Step-wise Explanations for Constraint Solving

Dissertation submitted in fulfilment of the requirements for the award of the joint degree of

Doctor of Business Economics: Business engineer, awarded by the VUB

and

Doctor of Engineering Science: Computer Science, awarded by the KU Leuven

by

Emilio Gamba

Supervisors

Prof. dr. Tias Guns
Prof. dr. Bart Bogaerts
Prof. dr. Vincent Ginis

VRIJE UNIVERSITEIT BRUSSEL

KU LEUVEN

Faculty of Social Sciences & Solvay Business School, Vrije Universiteit Brussel
Faculty of Engineering Science, KU Leuven

Members of the Examination Committee

Chairperson Prof. dr. Filip Van Droogenbroeck

Department of Business Technology and Operations, Data Analytics Lab, Vrije Universiteit Brussel (VUB), Belgium

Jury member Prof. dr. Gerda Janssens

Department of Computer Science, Declarative Languages and Artificial Intelligence (DTAI), Katholieke Universiteit Leuven (KU Leuven), Belgium

Jury member Prof. dr. María García de la Banda

Department of Data Science & AI, Monash University, Australia

Jury member Prof. dr. Christopher Jefferson

School of Computer Science, University of Saint Andrews, United Kingdom

Supervisor Prof. dr. Tias Guns

Department of Computer Science, Declarative Languages and Artificial Intelligence (DTAI), Katholieke Universiteit Leuven (KU Leuven)
Department of Business Technology and Operations, Data Analytics Lab, Vrije Universiteit Brussel (VUB)

Supervisor Prof. dr. Bart Bogaerts

Department of Computer Science, Artificial Intelligence Lab, Vrije Universiteit Brussel (VUB), Belgium

Supervisor Prof. dr. Vincent Ginis

Department of Business Technology and Operations, Data Analytics Lab, Vrije Universiteit Brussel (VUB), Belgium

Abstract

Constraint solving techniques provide decision support in many everyday and industrial applications, ranging from the scheduling of nurses in a hospital to sequencing jobs on machines under resource constraints. AI practitioners harness the power of highly efficient solvers by building a model-based representation of these constraint satisfaction problems (CSPs). Advances in constraint solving techniques and hardware improvements allow such systems to consider millions of alternatives in a short period of time. However, this complex reasoning makes it increasingly difficult to understand why certain decisions are made, i.e., which parts of the problem constraints were used to obtain the solution.

In this thesis, we study the problem of *explaining the inference steps* leading to a solution, in a *human interpretable* way, agnostic of the 'inner working' of the underlying solver used. The main challenge is to explain the solution by means of a *sequence* of *simple* explanation steps. Each explanation step should aim to be as cognitively easy as possible for a human to *understand*.

In our first contribution, we formalise the problem of step-wise explaining the maximal consequence of constraint satisfaction problems. We propose algorithms for generating a sequence of explanation steps. Each *explanation step* combines constraints with variable assignments (facts), to infer new information from the solution. We propose the use of a cost function to estimate the difficulty of an individual explanation step. This cost function captures, for example, the interaction between constraints and facts, or the type of constraints present in the explanation. Our algorithms for explanation generation exploit a one-to-one correspondence between so-called *non-redundant explanations* and Minimal Unsatisfiable Subsets (MUSs) of a derived unsatisfiable formula. Since some generated explanations are still difficult to understand, we propose to decompose a complex explanation into smaller *nested explanations* using reasoning by contradiction, thus providing the ability to *zoom-in* on explanations.

Unfortunately, MUS extraction techniques do not guarantee the optimality of an explanation with respect to such a cost function. Therefore, we address how to generate explanations by introducing *Optimal Constrained Unsatisfiable Subsets* (OCUS), that satisfy certain structural constraints and that are provably *optimal* (with respect to a given objective function), e.g., ensuring that we explain one fact of the solution at each step. We introduce a hitting–set-based algorithm that leverages the strengths of Mixed Integer Programming (MIP) and Boolean Satisfiability (SAT) for efficient OCUS computation.

So far, we have assumed the existence of a cost function that estimates the difficulty of an explanation. However, we do not currently know what a *good* function is. Furthermore, at each step, there are many candidate explanations for a single fact, and we do not currently know how to characterise what is the *most helpful explanation* for a user. It may even vary from one user to another. To address this challenge, we study *learning* preferences for explanations directly from users. In our third contribution, we investigate whether we can learn a preference function over a feature representation of explanations using learning-to-rank techniques, and its integration into an explanation generation workflow. We showcase on sudoku that we can learn functions that score well on choosing between two smallest MUSs. Furthermore, by using an iterative learning setup, we can directly learn a linear scoring function that can be integrated to directly generate preferred explanations. Finally, we conduct a small-scale user study to evaluate how well they perform when generating explanations directly versus selecting from a pre-generated set of explanations.

Overall, our results show that we are able to step-wise explain the maximal consequence of constraint satisfaction problems, with a focus on human interpretability, by providing the most helpful explanation to the user at each step.

Acknowledgement

This manuscript is the result of an intense journey that would not have been possible without the support of many people. I would like to begin by thanking my jury members Prof. dr. Gerda Janssens, Prof. dr. Christopher Jefferson, Prof. dr. María García de la Banda and Prof. dr. Filip Van Droogenbroeck for their time, comments and suggestions to help improve this dissertation. Christopher, although in the end we only met virtually, I enjoyed every discussion we had about the many surprising ways in which people solve puzzles.

When people ask me how the PhD is going, I always say that it is challenging, but I am surrounded by great supervisors. Each of them has complemented the others in so many ways. I would like to express my gratitude to the supervisors who gave me the opportunity to start a PhD. Tias, just over 4 years ago you gave me the opportunity to start an exciting journey as a teaching assistant. It was the perfect balance between research and teaching the basics of computer science to economics students. I remember the many hours spent preparing tutorials and your suggestions on how to make the material as easy to understand as possible. Besides teaching activities, you were also a great mentor. You gave me the tools to develop my skills as a researcher, but also as a person. I cannot count the many sleepless nights close to the deadlines of conference papers, but I can confidently say that I loved every single one of them. Bart, thank you for the intense but fun discussions. Your jokes would always lighten up the mood in the meetings. Your thoughtful comments and attention to detail helped me shape my critical thinking. Thank Prof. dr. Vincent Ginis for agreeing to be my supervisor and for supporting me during the last months of my PhD.

Throughout the years I have seen our team grow. At the beginning, the dragonflock consisted of Lize, Rocs, Jay, and Maxime bundled together in PL5.4.32. I enjoyed every minute we spent in our own office, and I will keep such great memories of us discussing chocolate, how to say things in French, helping each other with presentations. As our team grew, I had the chance to meet many bright minds including: Ahmed, Nicholas, Ignace, Senne, Bastian, Wout, Kostis, Thomas, Wout. Thank you guys for the fun coffee breaks! I am grateful to all the kind postdocs I met: Marjolein, Victor, Jo, Helene, Dimos and Irfan. I am indebted to you for all the time you have spent helping me become the person I am today. Finally, I would like to thank all my colleagues at DataLab, Buto, Mobi and DTAI at VUB and KULeuven as well as the department secretariat for their help in .

Tout ce que j'ai accompli aujourd'hui n'aurait pas été possible sans l'aide de ma famille et mes amis. Merci Jan et Anne de m'avoir accueilli comme un de vos enfants dans family Beheme. Merci Thomas, Clara, Alexia et Joseph pour tous ces chouettes moments passés et tous les bons souvenirs que j'en retiens. Dankje Geert voor jouw nieuwsgierigheid en de interesse die je in mijn werk toont. Merci tonton Marco pour les nombreuses discussions autour d'un café, et Tati pour tout le soutien que tu m'apportes.

Lisa, quel parcours ce doctorat a été pour toi et pour moi. Merci de m'avoir accompagné dans cette aventure. Je n'aurais pas été en mesure de terminer cette thèse sans ton soutien surtout ces derniers mois, Tu m'as donné de la force dans les moments où je doutais le plus de moi-même. Aujourd'hui, notre petite famille s'est agrandie. Lisa et Oscar, je vous aime tous les deux de tout mon cœur et, aujourd'hui, je ne peux pas imaginer ma vie autrement.

List of Publications

Part of the work reported in this thesis was published in the following publications:

- Bart Bogaerts, Emilio Gamba, Jens Claes, and Tias Guns. Step-wise explanations of constraint satisfaction problems. In *24th European Conference on Artificial Intelligence (ECAI)*, 2020. doi: 10.3233/FAIA200149
- Bart Bogaerts, Emilio Gamba, and Tias Guns. A framework for step-wise explaining how to solve constraint satisfaction problems. *Artificial Intelligence*, 300:103550, 2021. ISSN 0004-3702. doi: 10.1016/j.artint.2021.103550
- Emilio Gamba, Bart Bogaerts, and Tias Guns. Efficiently explaining csps with unsatisfiable subset optimization. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence, IJCAI*, pages 1381–1388, 2021. doi: 10.24963/ijcai.2021/191
- Emilio Gamba, Bart Bogaerts, and Tias Guns. Efficiently explaining csps with unsatisfibale subset optimization. *Journal of Artificial Intelligence Research*, 78:709–746, 2023. doi: 10.1613/jair.1.14260

Other publications which are not part of the thesis:

- Jens Claes, Bart Bogaerts, Rocsildes Canoy, Emilio Gamba, and Tias Guns. Zebratutor: Explaining how to solve logic grid puzzles. In *Proceedings of the 31st Benelux Conference on Artificial Intelligence (BNAIC 2019) and the 28th Belgian Dutch Conference on Machine Learning (Benelearn 2019), Brussels, Belgium, November 6-8, 2019*, 2019. URL `http://ceur-ws.org/Vol-2491/demo96.pdf`
- Tias Guns, Emilio Gamba, Maxime Mulamba, Ignace Bleukx, Senne Berden, and Milan Pesa. Sudoku assistant – an ai-powered app to help solve pen-and-paper sudokus. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(13):16440–16442, Sep. 2023. doi: 10.1609/aaai.v37i13.27072
- Ignace Bleukx, Jo Devriendt, Emilio Gamba, Bart Bogaerts, and Tias Guns. Simplifying Step-Wise Explanation Sequences. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280

of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:20, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-300-3. doi: 10.4230/LIPIcs.CP.2023.11

- Rocsildes Canoy, Jayanta Mandi, Victor Bucarey Lopez, Emilio Gamba, Maxime Mulamba, and Tias Guns. Tsp with learned zone preferences for last-mile vehicle dispatching. In *Amazon Last-Mile Routing Research Challenge*, pages XXX–1. MIT Libraries, 2021

Contents

List of Tables

List of Figures

List of Open Science Contributions

The codes, data and experiments developed for this thesis are open-source and available at:

- Codes and experiments for Step-wise explanations of Logic Grid Puzzles using IDP of Chapter 3:
  - `https://zenodo.org/records/4982025`
- Codes and experiments for unsatisfiable subset optimisation of Chapter 4:
  - `https://github.com/ML-KULeuven/ocus-explain`
- Codes, datasets and experiments for learning preferences over explanations for constraint satisfaction of Chapter 5 will be made available upon acceptance.

Abbreviations

NLP	Natural Language Processing. 1, 2, 43
OCUS	Optimal Constrained Unsatisfiable Subset. iii, iv, 14, 58, 62–69, 71–75, 77–79, 83, 86, 89
OUS	Optimal Unsatisfiable Subset. 14, 57, 58, 72, 74, 84, 93
POS	Part-Of-Speech tagging. 43, 44
QBF	Quantified Boolean Formulas. 58, 63
SAT	Boolean Satisfiability. iv, 5, 7, 8, 58
SMT	Satisfiability Modulo Theories. 5
SMUS	Smallest Minimal Unsatisfiable Subset. 63, 65
XAI	eXplainable Artificial Intelligence. 7
XAIP	eXplAInable Planning. 22

CHAPTER 1

Introduction

In this thesis, we explore methods that help bring explainable agency to constraint solving. We first sketch the general context of Artificial Intelligence (AI) and how constraint solving fits within the AI landscape. In Section 1.2, we discuss initiatives related to explainability in AI. Next, in Section 1.3, we give a brief overview of how explainability has been approached in constraint solving and the gap that this work aims to fill. Finally, for each chapter, we motivate, starting from the problem statement, how each contribution helps close the gap to explainable constraint solving.

1.1 Artificial Intelligence: Machine Learning, Automated Reasoning, and more

Artificial Intelligence is the field of computer science that aims to augment artificial systems with intelligence to perform tasks that typically require human intelligence. Due to a number of success stories, much of the attention in AI has shifted to the field of Machine Learning (ML). In machine learning, a system is trained to acquire knowledge by automatically extracting patterns from raw data [79]. Deep Learning [88], a subfield of ML, combines artificial neurons in multi-layer networks, called deep neural networks (DNN), to simulate the complex decision making of a human brain [88, 72].

Artificial Intelligence research encompasses more than just the development of data-driven learning methods (ML). Artificial intelligence also includes:

Natural Language Processing Natural Language Processing (NLP) enables machines to process, understand and generate human language, with applications in chatbots, translation systems and sentiment analysis.

Knowledge Representation and Reasoning (KRR) Knowledge representation involves structuring information in a way that a computer can use to reason. Examples of knowledge representation include semantic networks, (first-order) logic theories, frames, ontologies, and constraint satisfaction problems. In this thesis, we focus on *first-order logic* (FOL) theories and on *constraint satisfaction problems* (CSPs). In Chapter 2, we formally introduce the background knowledge related to FOL and CSPs, as well as the techniques we used to reason over this formal representation.

Constraint Solving Constraint solving involves formulating constraints on the values that can be assigned to variables in the problem, in order to restrict the solutions allowed by the problem.

Machine Perception Machine perception, often associated with *Computer Vision* (CV), enables AI systems to interpret and understand sensory data such as images or speech.

Robotics The design and programming of machines that can autonomously perform physically demanding and labour-intensive tasks.

Planning Planning algorithms allow systems to strategise actions given a current state, desired goals and resources.

A clear distinction between these research fields is fading as many now integrate data-driven techniques to improve and accelerate their performance. Large Language Models (LLMs), for example, rely on large neural networks to learn how to process and generate text directly from vast amounts of data using supervised and semi-supervised learning (ML).

In this thesis, we develop techniques in constraint solving (or constraint reasoning) that can be used to explain constraint satisfaction problems CSP. These problems can range from simple logic puzzles such as the Sudoku or a logic grid puzzle, to complex real-world problems such as scheduling, resource allocation, and optimisation. In the following, we present two examples of AI systems that showcase the explanation techniques developed in this thesis: (1) the *Zebra Tutor* developed in [19] and (2) the *Sudoku Assistant* [65].

Example 1 (Logic Grid Puzzle)**.** *A logic grid puzzle (also known as "Zebra puzzle" or "Einstein puzzle") consists of natural language sentences (from here on referred to as "clues") over a set of* entities *occurring in those sentences. The logic grid shown in Figure 1.1 is a puzzle about people having dinner in a restaurant and ordering different types of pasta. The right side of Figure 1.1 shows the clues that were given in the problem statement, as well as the reasoning techniques (Transitivity and Bijectivity) that can be used to solve it. The left-hand side corresponds to the logic grid that needs to be filled in by matching entities of different entity types with each other. In practice, the set of entities is sometimes left implicit if it can be derived from the clues, but it is often given in the form of a grid.*

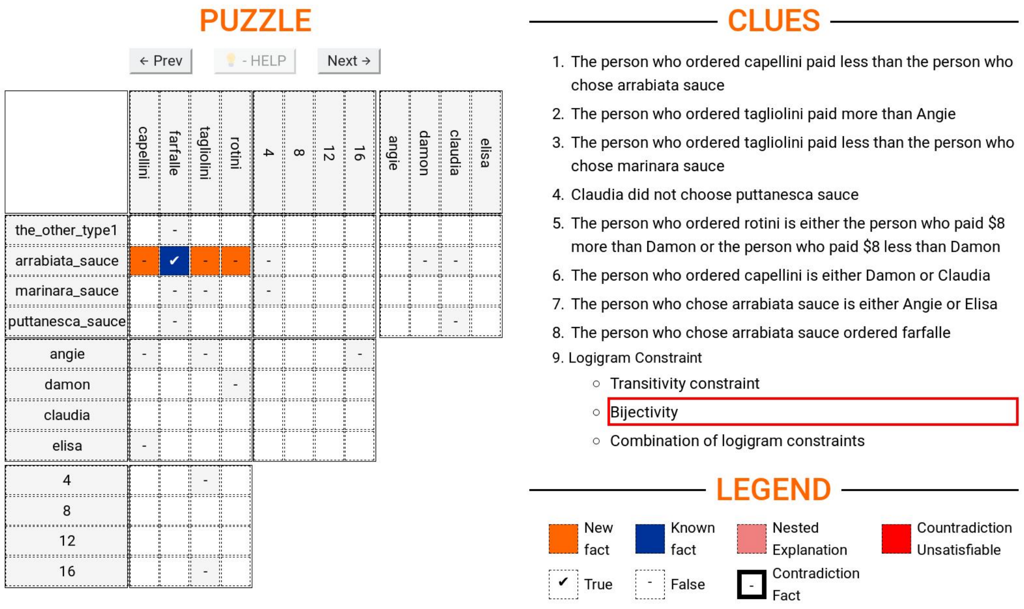
Furthermore, in such a puzzle the set of entities is partitioned into equally-sized groups (corresponding to types*); in our example, "person" and "sauce" are two such types. In short, the goal of the puzzle is to find relations between each two types such that*

- *Each clue is respected;*

- *Each entity of one type is matched with exactly one entity of each other type, e.g., each person chose exactly one sauce and each sauce is linked to one person (this type of constraint will be referred to as* bijectivity*); and*

- *The relations are logically linked, e.g., if angie chose arrabiata sauce and arrabiata sauce was paired with farfalle, then angie must also have eaten farfalle (from now on called* transitivity*).*

The logic grid puzzle shown is a puzzle about people (angie, elisa, claudia, damon) having dinner in a restaurant and ordering different types of pasta (rotini, taglioni, farfalle and capellini), with a unique sauce (arrabiate, marinara, puttanesca, and a last one that was not mentioned in any of the clues), and unique price. The logic grid already contains 1 (known) fact that has been derived using clue 8, i.e. farfalle (a type of pasta) has been associated with arrabiata sauce (a pasta sauce).



	capellini	farfalle	tagliolini	rotini	4	8	12	16	angie	damon	claudia	elisa
the_other_type1		-										
arrabiata_sauce	🟧 -	🟦 ✓	🟧 -	🟧 -	-					-	-	
marinara_sauce		-	-		-							
puttanesca_sauce		-									-	
angie	-		-					-				
damon				-								
claudia												
elisa	-											
4			-									
8												
12												
16			-									

CLUES

1. The person who ordered capellini paid less than the person who chose arrabiata sauce
2. The person who ordered tagliolini paid more than Angie
3. The person who ordered tagliolini paid less than the person who chose marinara sauce
4. Claudia did not choose puttanesca sauce
5. The person who ordered rotini is either the person who paid $8 more than Damon or the person who paid $8 less than Damon
6. The person who ordered capellini is either Damon or Claudia
7. The person who chose arrabiata sauce is either Angie or Elisa
8. The person who chose arrabiata sauce ordered farfalle
9. Logigram Constraint
   - Transitivity constraint
   - Bijectivity
   - Combination of logigram constraints

LEGEND

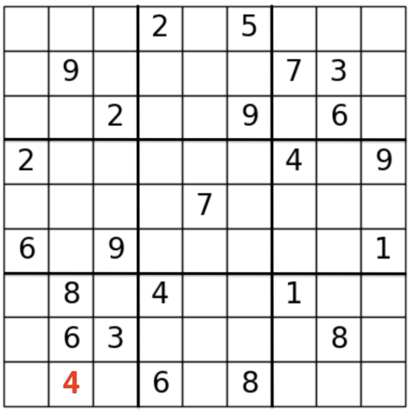
🟧 New fact	🟦 Known fact	▨ Nested Explanation	🟥 Countradiction Unsatisfiable
✓ True	- False	☐ Contradiction Fact	

Figure 1.1: Zebra Tutor: AI system to solve a logic grid puzzle (also known as a Zebra puzzle) given the clues in natural language and a list of the entities in the puzzle.[19]

For instance, our running example in Figure 1.1 contains as second clue "The person who chose arrabiata sauce is either Angie or Elisa" and (among others) the entities "arrabiata sauce", "Angie" and "Elisa".

Example 2 (Zebra Tutor)**.** *The* Zebra Tutor *is an AI system that illustrates how NLP techniques and Logic Reasoning can be combined to interpret and solve a logic grid puzzle specified in natural language (English), and subsequently, step-wise explain it in a human understandable way. Figure 1.1 shows an example of an explanation using the implicit 'Bijectivity' axiom present in each logic grid puzzle which is used to derive the following: since* arrabiata sauce *was eaten with* farfalle*, it was not eaten with any of the other pasta types. In Chapter 3, we provide the details of how this is done.*

Example 3 (Sudoku)**.** *The 9 x 9 Sudoku problem is a logic puzzle that requires filling in a set of digits from 1 to 9 into a rectangular grid, such that each column, each row and each 3 x 3 block contains exactly one occurrence of each digit. A valid Sudoku should have a unique solution. Figure 1.2 shows an example of a partially filled Sudoku that must be completed using the rules (constraints). However, there seems to be a mistake. In Section 1.3.2, we highlight some approaches to help identify, explain, and resolve this mistake.*

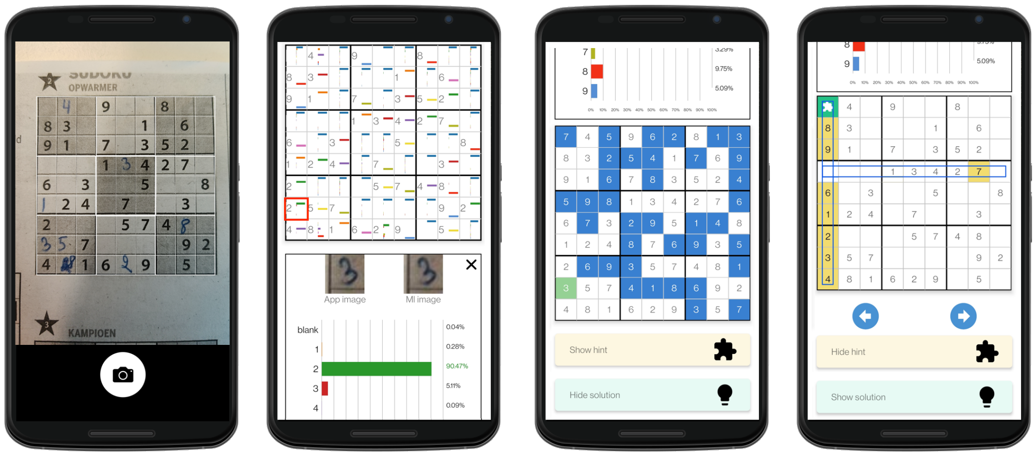


			2		5			
	9					7	3	
		2			9		6	
2						4		9
				7				
6		9						1
	8		4			1		
	6	3					8	
	4		6		8			

Figure 1.2: Partially filled Sudoku puzzle

Example 4 (Sudoku Assistant)**.** *The* Sudoku Assistant *[65] illustrates how computer vision, machine learning, and constraint reasoning techniques are combined in order to solve a pen-and-paper Sudoku puzzle from an image captured by a smartphone camera. An overview of the pipeline is shown in Figure 1.3.*

First, the image of the 9x9 Sudoku grid (Figure 1.3a) is split into 81 squares containing either the empty cell or a digit and a machine learning model is used to predict the probability for each label, i.e., 1 to 9 or the empty class. Next, a most likely satisfiable Sudoku is extracted by using a joint inference over the (Sudoku) constraints and the predicted probabilities, correcting potential



(a) Picture of Sudoku grid taken by a smartphone camera.

(b) Digit recognition, and prediction of most-likely satisfiable Sudoku grid.

(c) Solution of Sudoku.

(d) Step-wise explanation (hint) of the next digit to fill.

Figure 1.3: Sudoku Assistant: AI system combining computer vision, machine learning, and constraint reasoning techniques to interpret, solve a pen-and-paper Sudoku Puzzle taken by a smartphone camera. Once solved, the Sudoku Assistant assists the user by showing a hint of which digit to fill in next.

mistakes during prediction (Figure 1.3b). Once the puzzle has been solved (Figure 1.3c), the Sudoku Assistant helps the user by showing a hint as to which number to fill in next (Figure 1.3d).

These two examples of AI systems illustrate the complexity of artificial intelligence, and how AI aims to mimic human-like cognitive abilities.

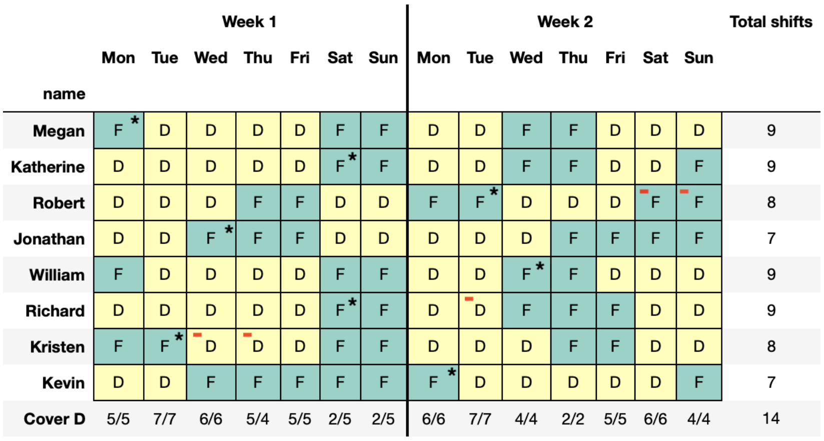
1.1.1 Constraint Solving

Many real-world applications benefit from constraint solving techniques to support decision making. In healthcare, scheduling nurses in a hospital requires finding an allocation of shifts that, for example, takes into account nurses' preferences and ensures that each shift is covered by enough nurses. Job shop scheduling is another example, but in an industrial setting. In job shop scheduling, jobs are scheduled to run on different machines in a factory under resource constraints.

These two complex problems are examples of constraint satisfaction problems (CSPs) where constraints express the restrictions or preferences that need to be satisfied in a solution. When an objective is introduced, such as minimising the number of resources used, the corresponding class of problems is called Constraint Optimisation Problems (COPs). By building a model-based representation of these problems, AI practitioners harness the power of highly efficient constraint solvers.

Healthcare and Industrial applications Consider two examples of real-world applications:

Example 5 (Healthcare)**.** *In healthcare, the nurse scheduling problem consists of finding an allocation of shifts and holidays that satisfies all hospital constraints and nurses' preferences.* [1,2]



	Week 1							Week 2							Total shifts
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun	
name															
Megan	F*	D	D	D	D	F	F	D	D	F	F	D	D	D	9
Katherine	D	D	D	D	D	F*	F	D	D	F	F	D	D	F	9
Robert	D	D	D	F	F	D	D	F	F*	D	D	D	-F	-F	8
Jonathan	D	D	F*	F	F	D	D	D	D	D	F	F	F	F	7
William	F	D	D	D	D	F	F	D	D	F*	F	D	D	D	9
Richard	D	D	D	D	D	F*	F	D	-D	F	F	F	D	D	9
Kristen	F	F*	-D	-D	D	F	F	D	D	D	F	F	D	D	8
Kevin	D	D	F	F	F	F	F	F*	D	D	D	D	D	F	7
Cover D	5/5	7/7	6/6	5/4	5/5	2/5	2/5	6/6	7/7	4/4	2/2	5/5	6/6	4/4	14

Figure 1.4: Nurse scheduling problem: shifts (D) and holidays (F) are assigned to nurses based on nurse's specific days off (*), and shifts on which they do not want to work (-).

For example, each nurse can work at most 5 consecutive shifts and has their own constraints, e.g., specific days off (*), and preferences, e.g. request not to work a specific shift (-). In addition, the hospital requires a minimum number of nurses to be present on each shift (Cover).

Figure 1.4 shows an example of a shift allocation for 8 nurses. Note that the proposed solution violates Richard's preference not to work on Tuesday of week 2, as he still has a shift assigned.

Example 6 (Industrial applications)**.** *Senthooran et al. [126] presents three industrial use cases that benefit from constraint solving: (1) Finding a satisfiable Plant Equipment Layout while en-*

[1]Examples of benchmark instances can be found at `http://www.schedulingbenchmarks.org/` including instances from various sources including industry collaborators and scientific publications.

[2]A more detailed description of different types of explanations (including their visualisation) for the nurse scheduling problem is available at `https://github.com/CPMpy/XCP-explain/`.

suring the best routes for connecting pipes; (2) Developing a Water Management System that connects a city's water supply and transfer lines to meet water demand, network constraints, and more; (3) Finding the number, location, and size of Hydrogen Production Facilities to meet demand, flow, storage, and turn-down constraints while minimising the production costs.

Toy problem Finally, throughout the thesis we will use the Sudoku puzzle as a pedagogical example of a CSP.

In the subsequent section, we present a high-level overview of the model-and-solve workflow typically employed by AI experts when tackling such complex problems.

Model and Solve

The typical workflow for tackling a new problem using Constraint Programming is divided into 2 key steps: modelling and solving [1].

During the *modelling* process, a real-world problem is described in a declarative way, typically as an instance of a constraint satisfaction problem (CSP) or as a Constraint Optimisation Problem (COP). An instance represents a specific instantiation of the problem consisting of a finite set of variables, their domains, constraints, and (if applicable) an objective function to be minimised or maximised. A constraint defines the relations between the variables involved in that constraint, called the scope of a constraint, and restricts the values that can be assigned to the variables in a solution.

In the *solving* process, a solver combines different techniques to explore the search space of variable-value assignments to find a solution (see Figure 1.5) that satisfies the problem constraints and, if applicable, optimise an objective.

Formal representation of a CS(O)P During the modelling process, an abstract representation of the problem, also called a model, is formulated to capture a family of problem instances by referring to some parameters representing the data, e.g. the number of variables or the domains of the variables. For the 9 x 9 Sudoku, this corresponds to providing the initial state of the grid. The model can then be represented in a preferred formalism. There are many formalisms and a suitable one should be chosen depending on the use-case at hand. To name a few:

- **Boolean Satisfiability** (SAT) is a paradigm that encodes satisfiability problems as a com-

[1]A more detailed tutorial is available at: `http://www.cril.univ-artois.fr/~lecoutre/teaching/constraints/ch2Modeling.pdf`

bination of 'and'-operators called *conjunctions* ($\wedge$),'or'-operators called *disjunctions* ($\vee$), 'not'-operators called a negations ($\neg$), and boolean variables called atoms (x_1, x_2, x_3), that are assigned either True or False [13]. An example of a logical formula encoded in SAT is:

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

For the formula to evaluate to true, x_1 must be true and, either x_2 must be *true* or x_3 must be *false*.

- **Satisfiability Modulo Theories** (SMT) extends Boolean satisfiability with predicates representing mathematical operations on non-binary variables [38]. For example, find an assignment of values to the variables $x, y \in \mathbb{N}$ such that the following constraint is satisfied:

$$x > 2, \quad y < 10, \quad x + 2y = 7$$

- **Logic Programming** is a paradigm for expressing logical relations, including the use of quantifiers, i.e. 'for all' $\forall$ and 'there exists' $\exists$. For example, *'every sauce must be paired with a type of pasta'* can be expressed as first-order logic (FOL) theory:

$$\forall\, s \in \mathit{sauce} : \exists\, p \in \mathit{pasta} : \mathit{pairedwith}(s, p)$$

Common logic programming languages include Prolog [31], and ASP [22].

- **Constraint Programming** (CP) is a computer science paradigm for solving combinatorial problems in which users declaratively state constraints on decision variables with a finite domain [119]. For example, by finding an assignment of values for the variables $x, y \in [2..5]$ such that the following constraint is satisfied:

$$x > 2, \quad y < 10, \quad x + 2y = 7$$

Model and solve The next step is to encode a mathematical formulation of the CSP in a *modelling language* such as CPMpy [64], MiniZinc [109], Essence [53], or as first-order logic theories using IDP [26]. Once encoded, the model is transformed to fit the input format of the underlying solver being used. Often, preprocessing techniques are applied to the input model to speed up the solving time when the solver is called to search for a solution, e.g. to detect symmetries in the formulation or to propagate individual constraints.

The constraint solver is then called to find an assignment of values to the variables that satisfies the constraints. If a solution is found, the problem is said to be *satisfiable*. However, if no solution can be found, i.e. not all constraints can be satisfied, then the problem is said to be *unsatisfiable*. This means that either the problem is inherently infeasible, there has been a (human) error in the modelling process, an error in the data provided by the instance or in the worst case there is a bug in the solver or compiler.

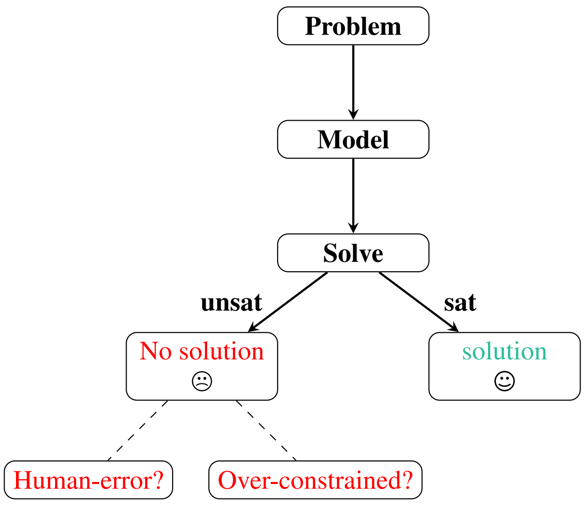


Figure 1.5: Model and solve pipeline

1.1.2 Solving using Inference and Search

The main strategies used by a constraint solver for reasoning are *inference* and *search*.

Search methods explore the space of variable value pairs to find solutions. Backtracking is an example of a search method that completes a partial solution by successively assigning values to variables until a constraint is violated. Once a failure is reached, the last variable assignment is undone, another value is selected from the variable's domain, and the search continues.

Inference is a reasoning technique used to restrict the values that can be taken by variables. *Constraint Propagation* is an example of inference used to enforce the constraints, individually or in combination, and propagate their effects through the solution space.

The interleaving of inference and search helps to efficiently explore the space of possible assignments. In this case, inference can ensure that large subspaces containing only non-solutions can be omitted from the search.

1.2 Explainable Artificial Intelligence

As AI systems employ more advanced reasoning mechanisms and computational power, it has become increasingly difficult to understand why certain decisions are made. Explainable AI (XAI) research aims to address the need for trustworthy AI systems to understand why the system has made a decision, in order to verify the correctness of the system and to check for biased or systematically unfair decisions. In the research community, this need is manifested by the emergence of (interdisciplinary) workshops and conferences on this topic [130, 69] and American and European incentives to stimulate research in this area [63, 67, 47]. On the **legislative** side, there is also increased attention to explainability, such as in the EU's GDPR 2016/679 [116].

The main focus of XAI research has been on explaining black-box systems such as neural networks, where the aim is to provide insights into which part(s) of the input are important in the *learned* model. These insights (or local approximations thereof) can justify *why* certain predictions are made. In this context, a number of techniques have been developed, ranging from local explanations of predictions at the *feature level* [118, 97] to **visual explanations** with *saliency maps* [125]. [1, 62, 9] provide an overview of the latest trends and main research directions in this area.

Model-based systems are typically considered to be more transparent. However, they are also in need of explanation mechanisms. For example, Vassiliades et al. [132] survey the important methods that use argumentation [108] to provide explainability in AI, with applications for example in medical diagnosis by Obeid et al. [112]. Abstract argumentation frameworks introduce an abstract formalism to explain argument acceptance [127, 92, 131]. Description logics [5], on the other hand, aim at explaining logical proofs [4, 85], i.e. 'why does phi follow from psi?'.

The main focus of this thesis is to provide explainable agency [86] to Constraint Programming (CP) [119] and Boolean Satisfiability (SAT) [14] systems. Advances in solving techniques and hardware improvements allow such systems to handle large models with complex objectives and many constraints. This complexity raises the question of how to generate *human interpretable* explanations of the conclusions they make.

In the following we explore how explainability is addressed in constraint reasoning at both the solving and modelling levels.

1.3 Explainability in Constraint Solving

Within constraint solving, there is already a rich literature on explanations. An important line of research there focuses on explanations that are useful to the *solver*, rather than the *user*.

1.3.1 Explanations at the Solving Level

In lazy clause generation solvers, explanations of a constraint are studied in the form of an implication of low-level Boolean literals that encode the result of a propagation step of an individual constraint [48]. Also, no-goods (learned clauses) in conflict-driven clause learning SAT solvers can be seen as explanations of failure during search [101]. These are *not* intended to be *human interpretable*, but rather to *propagate efficiently*, in other words to speed up solving.

1.3.2 Model-based Explanations

At the modelling level, the explicit *model-based representation* of the problem provides an opportunity to explain the inference steps directly in terms of this representation [119]. However, model-based systems, which are typically considered to be more transparent, are also in need of explanation mechanisms. The model itself can no longer be called "an explanation", and model-based systems also need explanation mechanisms.

In the following, we explore model-based explanations in the context of debugging unsatisfiable problems, how to restore their feasibility, and finally, inference-based explanations of satisfiability.

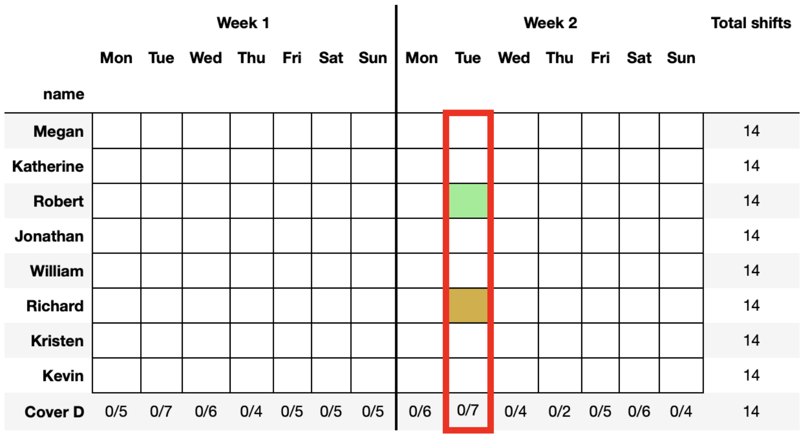
Explaining Unsatisfiability

In the CP and SAT communities, there has been a strong focus on explaining why **over-constrained**, and hence *unsatisfiable* problems have no solutions [80]. For example, QuickXplain [80] uses a dichotomous approach that recursively partitions the constraints to find a minimal conflict set. Many other papers consider the same goal and search for explanations of over-constrainedness [89, 135]. A small conflicting subset of the constraints, often called a *minimal unsatisfiable subset* (MUS) or *minimal unsatisfiable core* [94], is provided as an explanation of why the problem is unsatisfiable.

Feasibility Restoration

Feasibility restoration in constraint solving involves finding minimal changes to the model specification to 'repair' an unsatisfiable CSP to achieve feasibility, for example by extracting a minimal correction subset (MCS). An MCS is a minimal set of constraints that, when removed or relaxed, transforms an unsatisfiable problem into a satisfiable one [93, 89, 87, 126].

Example 7 (Nurse scheduling (continued))**.** *In Example 5, Robert has a day off and Richard prefers not to work on Tuesday of week 2. However, Tuesday in week 2 needs to be covered by 7 nurses, meaning that all 3 constraints cannot be satisfied together. This explanation is an example of a MUS that is extracted from the problem constraints.*



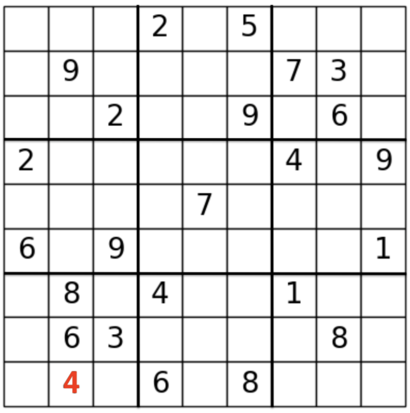
	Week 1							Week 2							Total shifts
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun	
name															
Megan															14
Katherine															14
Robert									🟩						14
Jonathan															14
William															14
Richard									🟨						14
Kristen															14
Kevin															14
Cover D	0/5	0/7	0/6	0/4	0/5	0/5	0/5	0/6	0/7	0/4	0/2	0/5	0/6	0/4	14

Figure 1.6: Explanation of unsatisfiability using a Minimal Unsatisfiable Subset (MUS).

To make our running example satisfiable, we can extract an MCS consisting of the following constraints:

1. *(1) Robert has a day off on Tuesday of week 2;*
2. *(2) Richard requests not to work the shift on Tuesday of week 2;*
3. *(3) The shift on Saturday of week 1 must be covered by 5 nurses out of the 8 nurses and;*
4. *(4) the shift on Sunday of week 1 must be covered by 5 nurses out of the 8 nurses.*

Example 8 (Sudoku Puzzle (continued))**.** *Consider the following example of a partially filled Sudoku that needs to be completed using the rules (constraints). However, there seems to be a mistake when filling in the 4 on the last line.*



			2		5			
	9					7	3	
		2			9		6	
2						4		9
				7				
6		9						1
	8		4			1		
	6	3					8	
	4		6		8			

Figure 1.7: Partially filled unsatisfiable Sudoku puzzle

A Minimal Unsatisfiable Subset (MUS) can then be extracted to explain why this Sudoku cannot be solved:

1. *2 cannot be in the first;*
2. *2 cannot be in the third column;*
3. *4, 6 and 8 are in the middle column of the block.*

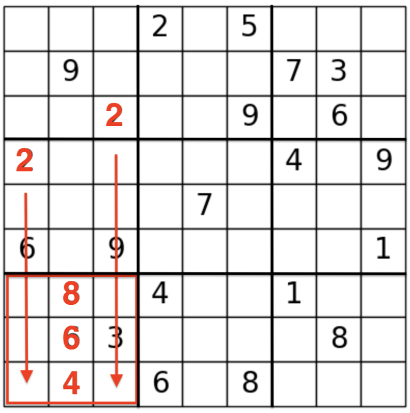
Therefore 4 cannot be in the second column of the last row.

In this case, either removing the 4 in the second column of the last row of Figure 1.8b or changing it to the correct digit (2), restores the feasibility of the Sudoku.

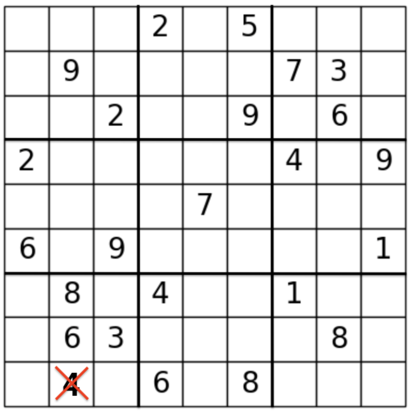
Explaining Satisfiability

In decision support systems, explaining satisfiability can enhance the trust and transparency of the system used in complex decision making problems. Furthermore, stakeholders are more likely to trust and accept solutions when they can understand and verify why the solutions found are correct. Despite its importance, explaining the satisfiability of a CSP remains underexplored in current research.

A solution can be extracted to explain why the problem is satisfiable. However, it does not provide *fine-grained information* about which constraint or combination of constraints led to which variable assignment in the solution.



(a) Explanation of infeasibility (Minimal Unsatisfiable Subset).



(b) Restoring feasibility (Minimal Correction Subset).

Figure 1.8: Examples of techniques for debugging an unsatisfiable Sudoku

Sqalli and Freuder [129] investigate logic grid puzzles and develop a number of problem-specific inference rules that allow solving (most, but not all) such puzzles without search. These inference rules are equipped with explanation templates such that each propagation event of an inference rule also has a templated explanation, and hence an explanation of the solution process is obtained. In [51], Freuder et al. study the implications of user choices to understand why a given solution was found, why choices led to a conflict (unsatisfiability), and more importantly why a value was assigned to a variable. An explanation in this context is defined as an inference step for which a set of problem features entail the result to be explained. For example, finding out which previous assignments (facts) form a sufficient basis to entail a new assignment (fact).

Holy Grail Challenge In 2019, the 'Holy Grail Challenge' [52] was organised, with the aim of providing *automated* processing of logic grid puzzles, ranging from natural language processing, to solving, and explaining. To address that challenge, one of the contestants [43] proposed an algorithm that uses 3 types of inference rules in a specific order: first clues, then basic consistency rules, advanced consistency rules. The specific order proposed in [43] is said to mimic the way that a human would solve a logic grid puzzle. Each time a new fact is propagated, a natural language explanation is generated using an inference rule based on the textual description of the constraints.

1.4 Research Gap

This thesis is motivated by the problem of explaining the maximal consequence of constraint satisfaction problems. We aim to provide *meaningful* and *short* explanations that are similar to human reasoning so that the user can understand the rationale behind it. More specifically, we aim to explain the process of propagation in a constraint solver, *independent of the consistency level of the propagation* and *without augmenting the solver and propagators with explanation capabilities*.

For problems that can — given a strong enough propagation mechanism — be solved without search, e.g., problems such as logic grid puzzles with a unique solution, this means explaining the entire problem-solving process. For problems involving search, this means explaining all variable assignments entailed by the constraint specification in one search node. This can be particularly useful if something unwanted is being propagated in a node of a deep search tree that involves many decisions. The user may need an explanation as to why the propagated information does not correspond to what the user had in mind. Since most of our work is illustrated on constraint satisfaction problems with unique solutions. For ease of exposition, we will refer to explaining the solution of a CSP when we talk about explaining the maximal consequence of a *CSP*.

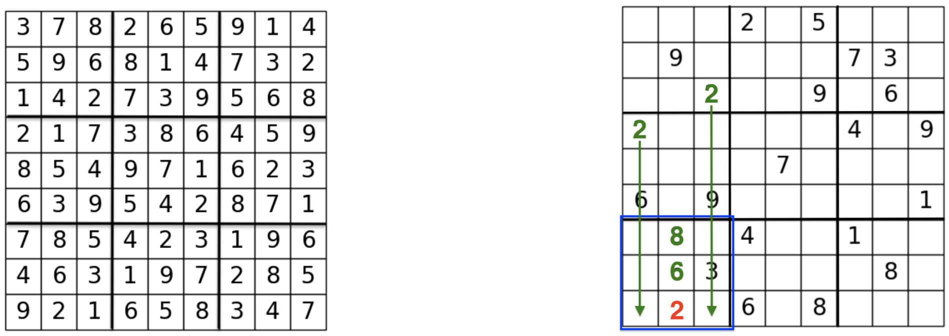
It deserves to be stressed that we are not interested in the computational cost of performing an expensive form of propagation, but in explaining all consequences of a given assignment to the user in a way that is as understandable as possible. More specifically, we aim to develop an explanation-producing system that is *complete* and *interpretable*. By *complete* we mean that it finds a *sequence* of small reasoning steps that, starting from the given problem specification and a partial solution, derives all consequences. Gilpin et al. [60] define *interpretable* explanations as:

> 'Descriptions that are simple enough for a person to understand, using a vocabulary that is meaningful to the user'

Our guiding principle is that of *simplicity*, where *smaller* and *simpler* explanations are better. Simplicity is approximated by means of a cost function which measures how difficult it would be for a human to understand the explanation. Figure 1.9b of Example 9 depicts an example of such an explanation, where as few constraints and already derived facts are used to infer a new fact.

Example 9 (Sudoku Puzzle (continued))**.** *Providing the full solution as in Figure 1.9a, does not help in understanding how each digit of the solution has been derived. Therefore, our explanation-producing system can be used to provide an explanation of which digit should be filled in next.*

Figure 1.9b illustrates such an explanation step on the Sudoku puzzle to explain how to derive 2 *using the problem specification, i.e. exactly one occurrence of a digit in every column, block and row, as well as the partial solution, i.e. the digits filled in the gird.*



(a) Solution of Sudoku.

(b) ‘Simple’ explanation step.

Figure 1.9: Example of step-wise explaining the solution of a Sudoku using a *short* and *meaningful* explanation that is similar to human reasoning.

A complete *sequence of such explanations allows for explaining a complete solution step-wise manner in a way that is* (human-)interpretable.

1.5 Research Questions

The main challenge we consider is how to explain the solution to a constraint satisfaction problem with a focus on human interpretability.

This leads to our main research question:

> *How to explain the maximal consequence of a constraint satisfaction problem step-by-step in a human understandable way?*

We propose to decompose the main research question into the following research questions:

RQ1 *How to decompose the solution to a constraint satisfaction problem into stepwise explanations, where each explanation should be as simple as possible for a human to understand?*

RQ2 *How can we break down difficult explanation steps into finer explanation steps?*

We propose the use of a cost function to approximate the simplicity of an explanation step. This leads to our next research question:

RQ3 *How to efficiently compute the easiest next explanation step that is guaranteed to be optimal with respect to a given cost function quantifying human interpretability?*

It is not clear what a *good* cost function is for characterising how *helpful* an explanation step is to a user. Ideally, such a function should be learned directly from users, especially since preferences may depend on the person to whom the explanation is given. However, there is no a dataset of explanation preferences labelled by users. This leads us to the following research questions:

RQ4 *How to collect a dataset of user preferences over explanation?*

RQ5 *How to learn a cost function that characterises which explanation is most helpful to a user?*

RQ6 *How to integrate a learned cost function into algorithms for explanation generation?*

1.6 Outline and Contributions

In the next chapter (Background), we introduce the formal concepts related to logic and constraint programming that are required for the remainder of the thesis. In the subsequent chapters, we present how to explain step-by-step solutions to constraint satisfaction problems with a focus on *human interpretability*. Our contributions in this thesis and the research questions they aim to answer can be summarised as follows, listed by chapter.

1.6.1 Chapter 3: A Framework for Step-wise Explaining How to Solve Constraint Satisfaction Problems

Chapter 3 aims to answer research questions RQ1 and RQ2 by proposing a framework for step-wise explaining how to solve constraint satisfaction problems, and the use of MUS-based algorithms for explanations generation.

The contributions to answer research question RQ1 are as follows:

- We formalise the problem of step-wise explaining the propagation of a constraint solver through a sequence of small inference steps;

- We propose an algorithm that is agnostic to the propagators and the consistency level used, and that can provide explanations for inference steps involving arbitrary combinations of constraints;
- Given a cost function quantifying human interpretability, our method uses an optimistic estimate of this function to guide the search to low cost explanations. For this, we make use of Minimal Unsatisfiable Subset extraction;
- We experimentally demonstrate the quality and feasibility of the approach in the domain of logic grid puzzles.

The contribution to answer research question RQ2 is as follows:

- We introduce nested explanations to provide additional explanations of complex inference steps using reasoning by contradiction;

Chapter 3 has been published as

> Bart Bogaerts, Emilio Gamba, Jens Claes, and Tias Guns. Step-wise explanations of constraint satisfaction problems. In *24th European Conference on Artificial Intelligence (ECAI)*, 2020. doi: 10.3233/FAIA200149
>
> Bart Bogaerts, Emilio Gamba, and Tias Guns. A framework for step-wise explaining how to solve constraint satisfaction problems. *Artificial Intelligence*, 300:103550, 2021. ISSN 0004-3702. doi: 10.1016/j.artint.2021.103550

1.6.2 Chapter 4: Efficient Explanation Generation with Unsatisfiable Subset Optimisation

Chapter 4 proposes the use of unsatisfiable subset optimisation to guarantee the generation of optimal explanations with respect to a given objective in order to answer research question RQ3.

The contributions to answer research question RQ3 are as follows:

- We develop algorithms that compute (cost-)Optimal Unsatisfiable Subsets (from now on called OUSs) based on the well-known hitting-set duality, which is also used for computing cardinality-minimal MUSs [74, 122].
- We observe that in the explanation setting, many of the individual calls for MUSs (or OUSs) can actually be replaced by a single call that searches for an optimal unsatisfiable subset *among subsets satisfying certain structural constraints*. We formalise and generalise this observation by introducing the *Optimal Constrained Unsatisfiable Subsets (OCUS)* problem. We then show how $O(n^2)$ calls to MUS/OUS can be replaced by $O(n)$ calls to an OCUS oracle, where n denotes the number of facts to explain.
- We develop techniques for *optimising* the O(C)US algorithms further, exploiting domain-specific information coming from the fact that we are in the *explanation generation context*. Such optimisations include
    1. the development of methods for *information re-use* between consecutive O(C)US calls; as well as
    2. an explanation-specific version of the OCUS algorithm.
- We extensively evaluate our approaches on a large number of CSP problems.

Chapter 4 has been published as:

> Emilio Gamba, Bart Bogaerts, and Tias Guns. Efficiently explaining csps with unsatisfiable subset optimization. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence, IJCAI*, pages 1381–1388, 2021. doi: 10.24963/ijcai.2021/191
>
> Emilio Gamba, Bart Bogaerts, and Tias Guns. Efficiently explaining csps with unsatisfibale subset optimization. *Journal of Artificial Intelligence Research*, 78:709–746, 2023. doi: 10.1613/jair.1.14260

1.6.3 Chapter 5: User-preferred Explanations

Chapter 5 proposes the integration of machine learning methods into our explanation algorithms to generate user-preferred explanations to answer RQ4, RQ5 and RQ6.

The contribution to answer RQ5 and RQ6 are as follows:

- We identify *domain-agnostic* as well as *domain-specific* features for MUSs by counting over different subsets of constraints how many are present in the MUS.
- We compare feature choices and use different point-wise and pair-wise learning-to-rank methods, showing that in the Sudoku domain, domain-specific features are essential to accurately discriminate between MUSs, and multiple learning techniques can do this well.
- We demonstrate that we can not only learn how to *discriminate* between two smallest MUSs, but that a few iterations of iterative learning with linear predictors allows learning a scoring function that can be directly used to *generate* good MUSs.

This chapter is currently under submission at a conference.

Finally, we conclude the thesis with a summary of the results, and discuss future directions for explanations in constraint programming, as well as some recommendations for AI practitioners in generating stepwise explanations.

CHAPTER 2

Constraint Satisfaction Problems: Background and Notation

This chapter is largely based on the preliminaries of the following journal papers:

> Bart Bogaerts, Emilio Gamba, and Tias Guns. A framework for step-wise explaining how to solve constraint satisfaction problems. *Artificial Intelligence*, 300:103550, 2021. ISSN 0004-3702. doi: 10.1016/j.artint.2021.103550

> Emilio Gamba, Bart Bogaerts, and Tias Guns. Efficiently explaining csps with unsatisfibale subset optimization. *Journal of Artificial Intelligence Research*, 78:709–746, 2023. doi: 10.1613/jair.1.14260

In this chapter, we introduce the terminology and concepts related to our explanation techniques.

2.1 Constraint Satisfaction Problems

Constraint Satisfaction Problems (CSP) are a class of combinatorial problems that involve a set of variables, each with a domain of possible values, and a set of constraints that restrict the values that can be assigned to the variables.

Definition 1 (CSP Rossi et al. [119]). *A constraint satisfaction problem (CSP) instance is a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ with*

- $\mathcal{X}$ *a set of* variables*;*
- $\mathcal{D}$ *a set of* domains D_x *of allowed* values *for each variable x of $\mathcal{X}$.*
- $\mathcal{C}$ *a set of* constraints*, over a subset of the variables.*

A *constraint* c is typically described by a formula (e.g., $x + y \geq 1$), and restricts the values that can be assigned to the variables. The scope $scope(c)$ of a constraint c corresponds to the variables appearing in a constraint, e.g. $scope(x + y \geq 1) = \{x, y\}$.

A *(partial) assignment* is a (partial) mapping of variables to values from their domain. An assignment *satisfies* (or falsifies) a constraint if the constraint evaluates to true (or false). Slightly abusing notation, if I is a partial assignment, we will identify it with the set of equalities (also called *facts*). A *solution* is a satisfying assignment of values to all variables in the problem.

Example 10 (Sudoku)**.** *Recall the rules for the Sudoku puzzle introduced in Example 3. The* 9×9 *Sudoku problem can be modelled as a CSP with an* 9×9 *array of variables* `cells`*, each over domain* $\{1..9\}$*. Each cell in the grid at column i and row j is associated with a variable* $\texttt{cells}[i,j]$*. The variables* `cells` *are subject to the following constraints:*

$\texttt{alldifR}_\texttt{i} = \texttt{alldifferent}(\texttt{cells}[\texttt{i},\texttt{j}] \mid \texttt{j} \in 1..9)$ *(for each* $i \in 1..9$*):*

- *all digits in one row* i *must be different.*

$\texttt{alldifC}_\texttt{j} = \texttt{alldifferent}(\texttt{cells}[\texttt{i},\texttt{j}] \mid \texttt{i} \in 1..9)$ *(for each* $j \in 1..9$*)*

- *all digits in the same column* j *must be different.*

$\texttt{alldifB}_{\texttt{i},\texttt{j}} = \texttt{alldifferent}(\texttt{cells}[\texttt{i}...\texttt{i}+2, \texttt{j}...\texttt{j}+2])$ *for all* $i, j \in \{1,4,7\}$*:*

- *all digits in the same block must be different.*

A partial assignment here corresponds to a partially filled Sudoku grid.

2.2 Propositional Logic and Boolean Satisfiability

Let Σ be a set of propositional symbols, also called *atoms*; this set is implicit in the rest of the thesis. A *literal* is an atom p or its negation $\neg p$. A clause is a disjunction ($\vee$) of literals, i.e. $c_1 = x_1 \vee \neg x_2$. A formula $\mathcal{F}$ in conjunctive normal form (CNF) is a conjunction ($\wedge$) of clauses. Slightly abusing notation, a clause is also viewed as a set of literals, and a formula is viewed as a set of clauses.

A (partial) interpretation I is a consistent (not containing both p and $\neg p$) set of literals. Satisfaction of a formula Fby an interpretation is defined as usual [14]: an interpretation I *satisfies* Fif I contains at least one literal from each clause in $\mathcal{F}$. A *model* of Fis an interpretation that satisfies F; $\mathcal{F}$ is said to be *unsatisfiable* if it has no models.

A literal l is a *consequence* of a formula Fif l holds in all $\mathcal{F}$'s models. The *maximal consequence* of a formula F, denotes a consequence that holds in all models. If I is a set of literals, we write $\overline{I}$ for the set of negated literals $\{\neg\ell \mid \ell \in I\}$.

Example 11. *Let $\mathcal{C}_1$ be the CNF formula over atoms x_1, x_2, x_3 with the following four clauses:*

$$c_1 := \neg x_1 \vee \neg x_2 \vee x_3 \qquad c_2 := \neg x_1 \vee x_2 \vee x_3 \qquad c_3 := x_1 \qquad c_4 := \neg x_2 \vee \neg x_3$$

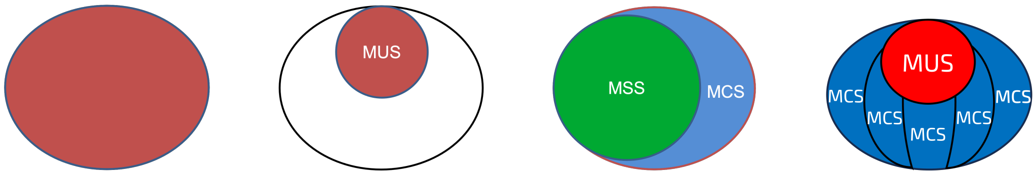
An example of an interpretation is $\{\neg x_2, x_3\}$. The maximal consequence *of formula $\mathcal{C}$ is $\{x_1, \neg x_2, x_3\}$.*

Definition 2. *A* Minimal Unsatisfiable Subset *(MUS) of Fis an unsatisfiable subset S of $\mathcal{F}$ for which every strict subset of S is satisfiable. $MUSs(\mathcal{F})$ denotes the set of MUSs of F.*

Definition 3. *A set $S \subseteq \mathcal{F}$ is a* Maximal Satisfiable Subset *(MSS) of $\mathcal{F}$ if S is satisfiable and for all S' with $S \subsetneq S' \subseteq \mathcal{F}$, S' is unsatisfiable.*

Definition 4. *A set $S \subseteq \mathcal{F}$ is a* correction subset *of Fif $\mathcal{F} \setminus S$ is satisfiable. Such a subset S is a* minimal correction subset *(MCS) of Fif no strict subset of S is also a correction subset. $MCSs(\mathcal{F})$ denotes the set of minimal correction subsets of F.*

Each MCS of Fis the complement of an MSS of Fand vice versa.



(a) Unsatisfiable model. (b) Minimal Unsatisfiable Subset (MUS). (c) Minimum Correction Subset (MCS) and Maximal Satisfiable Subset (MSS). (d) Hitting-set duality between MCSs and MUSs.

Figure 2.1: Visualisation of concepts related to (un)satisfiability.

2.2.1 First-order Logic

First-order Logic (FOL) is an extension of Propositional Logic augmented with the following elements:

Symbolic terms A symbolic term is either a variables (*x*, *y*, . . .), a name (*damon*, *arrabiata_sauce*) or an arbitrary term.

Predicates A *predicate(term$_1$, term$_2$, ...term$_n$)* of arity n describes the property of n symbolic terms. For example, predicate $chose(p, s)$ means that object p chose sauce s. For our running logic grid puzzle, $chose(damon, arrabiata\_sauce)$ means that damon chose arrabiata sauce.

functions A *function(*$term_1, term_2, ...term_n$*)* of arity n combines n symbolic terms to form another symbolic term. For example, $PriceOf(x)$ corresponds to the price of an object x, $PriceOf(arrabiata\_sauce)$ corresponds to the price of arrabiata sauce.

Quantifiers $\forall$ and $\exists$ allow us to quantify and reason about many objects at once.

The syntax of FOL should adhere to the following definitions:

Definition 5. *An arity n function combined with n symbolic terms is a symbolic term. An arity n predicate combined with n symbolic terms is a well-formed formula.*

Definition 6. *If there are no quantifiers in $\Phi(x)$ then x is a free variable in Φ. If for that formula Φ we write $\forall x \Phi(x)$ or $\exists x \Phi(x)$, we say that x is now bound in Φ. A variable that is bound is not free. A well-formed formula with no free variables is a sentence.*

Definition 7. *If Φ and Ψ are well-formed formulas, then the following logical operators applied to Φ and Ψ are also well-formed formulas (And if Φ and Ψ are sentences, then the following are also sentences):*

Operator	Description
$\neg\Phi$	Φ is false
$\Phi \vee \Psi$	Φ or Ψ must hold
$\Phi \wedge \Psi$	Φ and Ψ must hold
$\Phi \Rightarrow \Psi$	Φ implies Ψ
$\Phi \equiv \Psi$	Φ and Ψ are equivalent
$\Phi \Leftrightarrow \Psi$	Φ implies Ψ
$\exists x : \phi(x)$	$\phi(x)$ must hold for at least one x
$\forall x : \phi(x)$	$\phi(x)$ must hold for all x

Table 2.1: Syntax of First-order Logic (FOL)

2.3 Satisfiability and Unsatifisfiability

In the following, we introduce key properties related to unsatisfiable subsets on which we build our algorithms.

Definition 8. *Given a collection of sets H, a hitting set of H is a set h such that $h \cap C \neq \emptyset$ for every $C \in H$. A hitting set is* minimal *if no strict subset of it is also a hitting set.*

The next proposition is the well-known hitting set duality [94, 117] between MCSs and MUSs that forms the basis of our algorithms, as well as algorithms to compute MSSs [34] and *cardinality-minimal* MUSs [74].

Proposition 1. *A set $S \subseteq \mathcal{F}$ is an MCS of $\mathcal{F}$ if and only if it is a* minimal hitting set *of $MUSs(\mathcal{F})$. A set $S \subseteq \mathcal{F}$ is a MUS of $\mathcal{F}$ if and only if it is a* minimal hitting set *of $MCSs(\mathcal{F})$.*

Consider the following examples to illustrate the concepts related to unsatisfiability introduced in the previous definitions:

Example 12. *Let $\mathcal{C}_2$ be the unsatisfiable CNF formula over atoms x_1, x_2, x_3 with the following clauses:*

$$c_1 := \neg x_1 \vee \neg x_2 \vee x_3 \qquad c_2 := \neg x_1 \vee x_2 \vee x_3 \qquad c_3 := \neg x_2 \vee \neg x_3 \qquad c_4 := x_1$$

$$c_5 := \neg x_1 \vee x_2 \qquad c_6 := \neg x_1 \vee \neg x_3$$

In Example 12, the subset of clauses $\{c_1, c_2, c_4, c_6\}$ is an example of a Minimal Unsatisfiable Subset (MUS) and $\{\underline{c_1}\}, \{\underline{c_4}\}, \{\underline{c_2}, c_5\}, \{c_5, \underline{c_6}\}, \{c_3, \underline{c_6}\}$ are examples of minimal correction subsets (MCSs). We observe that MUS $\{c_1, c_2, c_4, c_6\}$ hits at least one clause of each MCS.

CHAPTER 3

A Framework for Step-wise Explaining How to Solve Constraint Satisfaction Problems

In constraint satisfaction, a solution can be extracted to explain why a constraint satisfaction problem (CSP) is satisfiable. However, it does not provide fine-grained information about which combination of constraints led to which variable assignment of the solution. The aim of this chapter is to step-wise explain the maximal consequence of a CSP as a sequence of simple inference steps, each explanation should aim to be as cognitively easy as possible for a human to understand.

This chapter is largely based on the following scientific publications:

Bart Bogaerts, Emilio Gamba, Jens Claes, and Tias Guns. Step-wise explanations of constraint satisfaction problems. In *24th European Conference on Artificial Intelligence (ECAI)*, 2020. doi: 10.3233/FAIA200149

Bart Bogaerts, Emilio Gamba, and Tias Guns. A framework for step-wise explaining how to solve constraint satisfaction problems. *Artificial Intelligence*, 300:103550, 2021. ISSN 0004-3702. doi: 10.1016/j.artint.2021.103550

3.1 Introduction

In this chapter, we aim to explain all logical consequences of constraint satisfaction problem. To do so, we aim to develop a *complete* and *interpretable* explanation-producing system with simplicity as guiding principle, where smaller and simpler explanations are better. The main challenge is to find a sequence of *simple* explanations where each explanation should be as cognitively easy as possible for a human to verify and understand.

An open issue when presenting a sequence of explanations is the level of *abstraction* used. On top of this idea of explaining propagations by means of as simple as possible explanations steps, there are two ways to change the level of abstraction.

The first direction is that one could *group* multiple single steps into one larger step that can then be expanded on the fly. The second direction is to provide a *more detailed* explanation of harder steps which can not be broken into simpler steps with the standard mechanism. While the first direction can be useful when explanations get very long, from a theoretical perspective, it is less interesting. Note that implementing this for real-world problems, might require significant engineering effort to do this in a way that preserves meaningful explanations.

The second direction on the other hand is interesting for several reasons. First, we noticed that some generated steps are still too complicated to be understood easily and thus really require being explained in more detail. Secondly, breaking them up in smaller steps is an interesting challenge. Indeed, since we start from the idea that steps should be as simple as possible, it should not be possible to break them up further. To still break them up further, we took inspiration from methods people use when solving puzzles, and mathematicians use when proving theorems. That is, for explaining a single step in more detail we work using reasoning by contradiction: starting from the negation of a consequence, we explain how a contradiction is obtained. In essence, we assume the negation of the derived fact, and can then reuse the principle of explanation steps to construct a sequence that leads to an inconsistency. This novel approach allows us to provide a mechanism for *zooming in* on the most difficult explanation step.

This chapter and more specifically the use case of logic grid puzzles, is motivated by the 'Holy Grail Challenge'[1] which had as objective to provide *automated* processing of logic grid puzzles, ranging from natural language processing, to solving, and explaining. While our integrated system, ZEBRATUTOR, has the capability of solving a logic grid puzzle starting from the natural language clues (see Section 3.8), the focus in this chapter is on the *explanation-producing* part of the system.

Contributions Summarised, our main contributions are the following:

- We formalise the problem of step-wise explaining the propagation of a constraint solver through a sequence of small inference steps;
- We propose an algorithm that is agnostic to the propagators and the consistency level used, and that can provide explanations for inference steps involving arbitrary combinations of constraints;

[1] `https://freuder.wordpress.com/pthg-19-the-third-workshop-on-progress-towards-the-holy-grail/`

- Given a cost function quantifying human interpretability, our method uses an optimistic estimate of this function to guide the search to low cost explanations, thereby making use of Minimal Unsatisfiable Subset extraction;
- We introduce nested explanations to provide additional explanations of complex inference steps using reasoning by contradiction;
- We experimentally demonstrate the quality and feasibility of the approach in the domain of logic grid puzzles as pedagogical examples of constraint satisfaction problems.

This chapter is structured as follows. In Section 3.2, we discuss related work. Section 3.3, explains the rules of logic grid puzzles and presents background information. Sections 3.4 and 3.5, formalise the theory of the explanation-production problem on two abstraction levels, while Section 3.6 describes the algorithms needed to solve the explanation-production problem. In Section 3.7, we motivate our design decisions using observations from the ZEBRATUTOR use case. Section 3.8 describes the entire information pipeline of the ZEBRATUTOR integrated system. In Section 3.9, we experimentally study the feasibility of our approach. Finally, we conclude the chapter in Section 3.10.

3.2 Related work

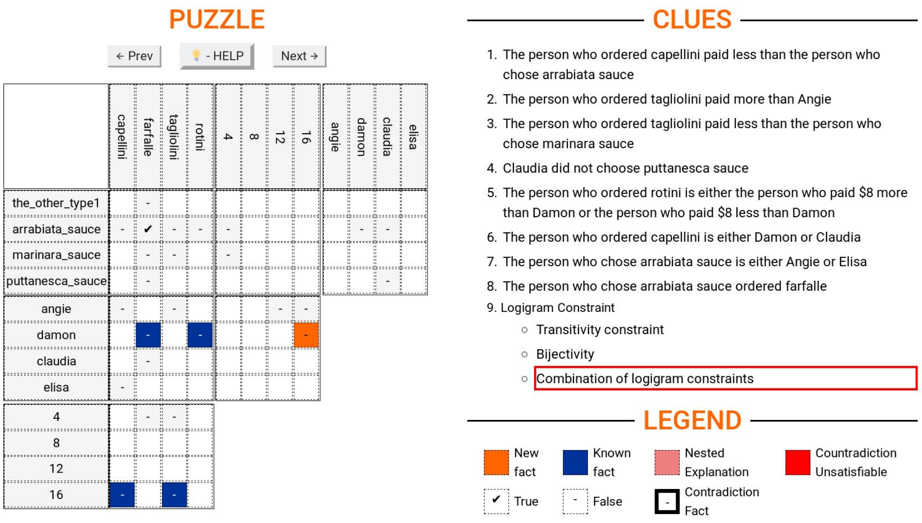
This research fits within the general topic of Explainable Agency [86], whereby in order for people to trust autonomous agents, the latter must be able to *explain their decisions* and the *reasoning* that produced their choices.

Explanations of constraint satisfaction problems (CSPs) have been studied most in the context of over-constrained problems, and hence *unsatisfiable* problems have no solutions [80]. As introduced in Section 2.3, a small conflicting subset of the constraints, often called a *minimal unsatisfiable subset* (MUS) or *minimal unsatisfiable core* [94], is provided as an explanation of why the problem is unsatisfiable. Despite the fact that we do not (specifically) aim to explain over-constrained problems, our algorithms will internally also make use of MUS extraction methods.

While explainability of constraint optimisation has received little attention so far, in the related field of *planning*, there is the emerging subfield of *eXplainable AI planning* (XAIP) [49], which is concerned with building planning systems that can explain their own behaviour. This includes answering queries such as ‘why did the system (not) make a certain decision?’, ‘why is this the best decision?’, etc. In contrast to explainable machine learning research [62], in explainable planning one can make use of the explicit *model-based representation* over which the reasoning happens. Likewise, we will make use of the constraint specification available to constraint solvers, more specifically typed first-order logic [133].

In contrast, our proposed method works for any type of constraint and any combination of constraints, and automatically infers a minimal set of facts and constraints that explain an inference step, without using any problem-specific knowledge. This powerful combination of constraints is able to automatically detect interesting consistency patterns that needed to be hand-coded in the Freuder's seminal work, and also in the solutions submitted by other contestants in the challenge [43].

Example 13. *Figure 3.1 shows an example of an explanation that our approach automatically generated for a given logic grid state. The facts highlighted by a blue and grey background are considered known and were previously derived by propagating the clues, so-called bijectivity and transitivity constraints individually.*



	capellini	farfalle	tagliolini	rotini	4	8	12	16	angie	damon	claudia	elisa
the_other_type1		-										
arrabiata_sauce	-	✔	-	-	-					-	-	
marinara_sauce		-	-		-							
puttanesca_sauce		-									-	
angie	-		-				-	-				
damon		-		-				-				
claudia		-										
elisa	-											
4		-	-									
8												
12												
16	-		-									

CLUES

1. The person who ordered capellini paid less than the person who chose arrabiata sauce
2. The person who ordered tagliolini paid more than Angie
3. The person who ordered tagliolini paid less than the person who chose marinara sauce
4. Claudia did not choose puttanesca sauce
5. The person who ordered rotini is either the person who paid $8 more than Damon or the person who paid $8 less than Damon
6. The person who ordered capellini is either Damon or Claudia
7. The person who chose arrabiata sauce is either Angie or Elisa
8. The person who chose arrabiata sauce ordered farfalle
9. Logigram Constraint
   - Transitivity constraint
   - Bijectivity
   - Combination of logigram constraints

LEGEND

🟧 New fact	🟦 Known fact	🟥 Nested Explanation	🟥 Countradiction Unsatisfiable
✔ True	- False	☐ Contradiction Fact	

Figure 3.1: Demonstration of explanation visualization for explaining why *damon* has not paid 16$ for his pasta. If *taglioni* and *capellini* do not cost 16$, then either *farfalle* and *rotini* must cost 16. If *damon* has not selected *farfalle* or *rotini*, then he must not have paid 16$.

The explanation shown in Figure 3.1 is non-trivial, it combines both bijectivity and transitivity constraints, which was hard-coded as the special-purpose 'path consistency' pattern in earlier logic-grid specific work [129].

There is a rich literature on automated and interactive theorem proving, recently focusing on providing proofs that are understandable for humans [57] and, e.g., on teaching humans – using interaction with theorem provers – how to craft mathematical proofs [134]. Our work fits into this line of research since our generated explanations can also be seen as proofs, but in the setting of finite-domain constraint solving.

In the context of interactive explanations, Caine and Cohen [23] introduce a *mixed-initiative* Intelligent Tutoring System (MITS). In that framework, the system maintains a model of the knowledge of the user (who is solving a sudoku), estimating how well certain strategies are understood. The tutor, based on its model of the student, decides whether the benefits outweigh the costs of interaction with the student (i.e. proposing a move or explaining a step) when the student lacks knowledge. The student also has the ability to ask for further guidance; in other words, the two involved parties can take the initiative to engage in interaction. In comparison, in our approach, we do not aim to model the user, but rather assume a cost-function is specified that quantifies difficulties of derivations. At each point in time, we then suggest the simplest next derivation.

3.3 Background

Our running example is a puzzle about people having dinner in a restaurant and ordering different types of pasta, which is the hardest logic grid puzzle we encountered; it was sent to us by someone who got stuck solving it and wondered whether it was correct in the first place. The entire puzzle can be seen in Figure 3.1; the full explanation sequence generated for it can be found online at `http://bartbog.github.io/zebra/pasta`.

In practice, we choose to present the constraints in natural language, which is an obvious choice for logic grid puzzles as they are given in the form of natural language *clues*. We represent the previously and newly derived facts visually, as can be seen in the grid in Figure 1.1.

In the following section, we introduce *typed first-order logic*, which we use as the language to express our constraint programs in. Here, it is important to stress that our definitions and algorithms work for any language with model-theoretic semantics, including typical constraint programming languages [119].

In Section 3.8, we explain how we obtain a vocabulary and first-order theory in a mostly automated way from the clues using Natural Language Processing. The result is a vocabulary with types corresponding to the groups of entities in the clues, and the names and types of the binary relations to find (e.g *chose(person, sauce)*, *paired(sauce, pasta)*, *eaten(person, pasta)*); as well as constraints (first-order sentences) corresponding to the clues, and the bijectivity and transitivity constraints.

3.3.1 Typed First-order Logic

Our constraint solving method is based on *typed first-order logic*. We refer to Section 2.2.1 for the syntax and semantics of first-order logic.Part of the input is a logical vocabulary consisting of a set of types, (typed) constant symbols, and (typed) relation symbols with associated type signature (i.e., each relation symbol is typed $T_1 \times \cdots \times T_n$ with n types T_i).[1] For our running example, constant symbol *Angie* of type *person* is linked using relation *chose(…, …)* with signature *person* $\times$ *sauce* to constant symbol *arrabiata sauce* of type *sauce*.

Definition 9. *A* first-order theory *is a set of sentences (well-formed variable-free first-order formulas [42] in which each quantified variable has an associated type), also referred to as constraints.*

Since we work in a fixed and finite domain, the vocabulary, the interpretation of the types (the domains) and the constants are fixed. This justifies the following definitions:

Definition 10. *A* (partial) interpretation *is a finite set of literals, i.e., expressions of the form $P(\overline{d})$ or $\neg P(\overline{d})$ where P is a relation symbol typed $T_1 \times \cdots \times T_n$ and $\overline{d}$ is a tuple of domain elements where each d_i is of type T_i.*

Definition 11. *A partial interpretation is* consistent *if it does not contain both an atom and its negation, it is called a* full *interpretation if it either contains $P(\overline{d})$ or $\neg P(\overline{d})$ for each well-typed atom $P(\overline{d})$.*

For instance, in the partial interpretation I_1,

$$I_1 = \{chose(angie, arrabiata),\ \neg chose(elisa, arrabiata)\}$$

it is known that $angie$ had *arrabiata sauce* while *elisa* did not. This partial interpretation does not specify whether or not *elisa* ate *farfalle*.

$I \models C$ defines a satisfaction relation between first-order theories C and full interpretations I as . If $I \models C$, we say that I is a model (or solution) of C.

Definition 12. *A partial interpretation I_1 is* more precise *than partial interpretation I_2 (notation $I_1 \geq_p I_2$) if $I_1 \supseteq I_2$.*

Informally, one partial interpretation is more precise than another if it contains more information. For example, the partial interpretation $I_2 = \{chose(angie, arrabiata, \neg chose(elisa, arrabiata), \neg chose(damon, arrabiata)\}$ is more precise than I_1 ($I_2 \geq_p I_1$).

[1] We here omit function symbols since they are not used in this thesis.

For practical purposes, since variable-free literals are also sentences, we will freely use a partial interpretation as (a part of) a theory in solver calls or in statements of the form $I \wedge C \models J$, meaning that everything in J is a consequence of I and C, or stated differently, that J is less precise than any model M of C satisfying $M \geq_p I$.

In the context of first-order logic, the task of finite-domain constraint solving is better known as *model expansion* [106]: given a logical theory C (corresponding to the constraint specification) and a partial interpretation I with a finite domain (corresponding to the initial domain of the variables), find a model M more precise than I (a partial solution that satisfies C).

Definition 13. *If P is a logic grid puzzle, we will use C_P to denote a first-order theory consisting of:*

- *One logical sentence for each clue in P.*
- *A logical representation of all possible bijection constraints.*
- *A logical representation of all possible transitivity constraints.*

For instance, for our running example, some sentences in C_P can represented as the following typed FOL sentences:

Typed First-order Logic Sentence	**Description**
$\neg chose(claudia, puttanesca)$	(Clue 4) Claudia did not choose puttanesca sauce.
$\forall\, s \in sauce : \exists\, p \in pasta : pairedwith(s,p)$	*(Bijectivity)* There exists a pasta for every sauce.
$\forall\, s \in sauce : \forall\, p1 \in pasta, \forall\, p2 \in pasta :$ $pairedwith(s,p1) \wedge pairedwith(s,p2) \Rightarrow p1 = p2.$	*(Bijectivity)* Every sauce can only be paired with one type of pasta.
$\forall\, s \in sauce : \forall\, p \in person, \forall\, c \in price :$ $chose(s,p) \wedge payed(p,c) \Rightarrow priced(s,c).$	*(Transitivity)* If a person pairs sauce s with pasta p, and pays price c, then sauce s costs (is priced at) c.

3.4 Problem definition

The overarching goal of this chapter is to generate a sequence of small reasoning steps, each with an interpretable explanation. We first introduce the concept of an explanation of a single reasoning step, after which we introduce a cost function as a proxy for the interpretability of a reasoning step, and the cost of a sequence of such steps.

3.4.1 Explanation of reasoning steps

We assume that a theory C and an initial partial interpretation I_0 are given and fixed.

Definition 14. *We define the* maximal consequence *of a theory C and partial interpretation I (denoted $max(I, C)$) as the most precise partial interpretation J such that $I \wedge C \models J$.*

Phrased differently, $max(I, C)$ is the intersection of all models of C that are more precise than I; this is also known as the set of *cautious consequences* of T and I, and corresponds to ensuring *global consistency* in constraint solving. Algorithms for computing cautious consequences without explicitly enumerating all models exist, such as for instance the ones implemented in clasp [58] or IDP [26] (in the latter system the task of computing all cautious consequences is called *optimal-propagate* since it performs the strongest propagation possible).

Weaker levels of propagation consistency can be used as well, leading to a potentially smaller maximal consequence interpretation $max_{otherConsistency}(I, C)$. The rest of this chapter assumes we want to construct a sequence that starts at I_0 and ends at $max(I_0, C)$ for some consistency algorithm, i.e., that can explain all computable consequences of C and I_0.

Definition 15. *A* sequence of incremental partial interpretations *of a theory C with an initial partial interpretation I_0 is a sequence*

$$\langle I_0, I_1, \ldots, I_n = max(I_0, C)\rangle$$

where $\forall\ i > 0,\ I_{i-1}\ \leq_p\ I_i$ (i.e., the sequence is precision-increasing).

The goal of our work is not just to obtain a sequence of incremental partial interpretations, but also for each incremental step $\langle I_{i-1}, I_i\rangle$ we want an explanation $(\mathcal{E}_i, \mathcal{S}_i)$ that justifies the newly derived information $\mathcal{N}$=$I_i \setminus I_{i-1}$. When visualised, such as in Figure 1.1, it will show the user precisely which information and constraints were used to derive a new piece of information.

Definition 16. *Let I_{i-1} and I_i be partial interpretations such that $I_{i-1} \wedge C \models I_i$. We say that $(\mathcal{E}_i, \mathcal{S}_i, \mathcal{N}_i)$* explains *the derivation of I_i from I_{i-1} if the following hold:*

- *$N_i = I_i \setminus I_{i-1}$ (i.e., N_i consists of all newly defined facts),*
- *$\mathcal{E}_i \subseteq I_{i-1}$ (i.e., the explaining facts are a subset of what was previously derived),*
- *$\mathcal{S}_i \subseteq C$ (i.e., a subset of the constraints is used), and*
- *$\mathcal{S}_i \cup \mathcal{E}_i \models \mathcal{N}_i$ (i.e., all newly derived information indeed follows from this explanation).*

The problem of simply checking whether $(\mathcal{E}_i, \mathcal{S}_i, \mathcal{N}_i)$ explains the derivation of I_i from I_{i-1} is in co-NP since this problem can be performed by verifying that $\mathcal{S}_i \wedge \neg\mathcal{N}_i$ has no models more precise than $\mathcal{E}_i$. It is hence an instance of the negation of a first-order model expansion problem [84].

Part of our goal of finding easy to interpret explanations is to avoid redundancy. That is, we want a non-redundant explanation $(\mathcal{E}_i, \mathcal{S}_i, \mathcal{N}_i)$ where none of the facts in $\mathcal{E}_i$ or constraints in S_i can be removed while still explaining the derivation of I_i from I_{i-1}; in other word: the explanation must be *subset-minimal*.

Definition 17. *We call $(\mathcal{E}_i, \mathcal{S}_i, \mathcal{N}_i)$ a* non-redundant explanation of the derivation of I_i from I_{i-1} *if it explains this derivation and whenever $\mathcal{E}' \subseteq \mathcal{E}_i; \mathcal{S}' \subseteq \mathcal{S}_i$ while $(\mathcal{E}', \mathcal{S}', \mathcal{N}')$ also explains this derivation, it must be that $\mathcal{E}_i = \mathcal{E}', \mathcal{S}_i = \mathcal{S}'$.*

Definition 18. *A* non-redundant explanation sequence *is a sequence*

$$\langle (I_0, (\emptyset, \emptyset, \emptyset)), (I_1, (\mathcal{E}_1, \mathcal{S}_1, \mathcal{N}_1)), \dots, (I_n, (\mathcal{E}_n, \mathcal{S}_n, \mathcal{N}_n)) \rangle$$

such that $(I_i)_{i \leq n}$ is a sequence of incremental partial interpretations and each $(\mathcal{E}_i, \mathcal{S}_i, \mathcal{N}_i)$, with $\mathcal{E}_i \subseteq I_{i-1}$ and $\mathcal{S}_i \subseteq C$, explains the derivation of I_i from I_{i-1}.

3.4.2 Interpretability of reasoning steps

While subset minimality ensures that an explanation is non-redundant, it does not quantify how *interpretable* an explanation is. This quantification is *problem-specific* and is often *subjective*. In this chapter, we will assume the existence of a cost function $f(\mathcal{E}_i, \mathcal{S}_i, \mathcal{N}_i)$ that quantifies the interpretability of a single explanation in line with the goal of 'simple enough for a person to understand' and Occam's Razor.

3.4.3 Interpretability of a sequence of reasoning steps

In its most general form, we would like to optimise the understandability of the entire sequence of explanations. While quantifying the interpretability of a single step can be hard, doing so for a sequence of explanations is even harder. For example, is it related to the *most difficult step* or the *average difficulty*, and how important is the *ordering* within the sequence? As a starting point, we here consider the total cost to be an aggregation of the costs of the individual explanations, e.g. the average or maximum cost.

Definition 19. *Given a theory C and an initial partial interpretation I_0, the* explanation-production problem *consists of finding a non-redundant explanation sequence*

$$\langle (I_0, (\emptyset, \emptyset, \emptyset)), (I_1, (\mathcal{E}_1, \mathcal{S}_1, \mathcal{N}_1)), \ldots, (I_n, (\mathcal{E}_n, \mathcal{S}_n, \mathcal{N}_n)) \rangle$$

such that a predefined aggregate over the sequence $(f(\mathcal{E}_i, \mathcal{S}_i, \mathcal{N}_i))_{i \leq n}$ *is minimised.*

Example aggregation operators are $max()$ and $average()$, which each have their peculiarities: the $max()$ aggregation operator will minimise the cost of the most complicated reasoning step, but does not capture whether there is one such step used, or multiple. Likewise, the $average()$ aggregation operator will favor many simple steps, including splitting up trivial steps into many small components if the constraint abstraction allows this. Even for a fixed aggregation operator, the problem of holistically optimising a sequence of explanation steps is much harder than optimising the cost of a single reasoning step, since there are exponentially more sequences.

3.5 Nested Explanations

If each explanation in the sequence is non-redundant, this means the steps cannot be further broken up in smaller sub-steps. Yet, in our earlier work we noticed that some explanations were still quite hard to understand, mainly since a clue had to be combined with implicit constraints and a couple of previously derived facts. All these things *together* implied a consequence, and they had to be taken into account at once. Such steps turned out to be too complicated to be understood easily and thus require being explained in more detail.

An example of a more difficult step is depicted at the top of Figure 3.2. It uses a disjunctive clue ('The person who ordered Rotini is either the person who paid \$8 more than Damon or the person who paid \$8 less than Damon'), in combination with three previously derived facts to derive that Farfalle does not cost \$8. This derivation is non-trivial when only looking at the highlighted clue. It turns out that the derivation can be explained in a step-wise manner using only implicit constraints with the help of reasoning by contradiction:

- If Farfalle did cost \$8, then (since Damon did not eat Farfalle), Damon did not pay \$8;
- If Farfalle costs \$8, then it does not cost \$16;
- Since Farfalle does not cost \$16 and neither does Capellini or Tagliolini, Rotini must cost \$16;
- However, the fact that Rotini costs \$16, while Damon did not pay \$8 is in contradiction with the clue in question;

- Hence, Farfalle can not cost \$8.

We hence wish to provide a further, *nested* explanation of such difficult reasoning steps. We believe that an explanation using contradiction is a good tool for this for two reasons: *(i)* it is often used by people when solving puzzles, as well as by mathematicians when proving theorems; and *(ii)* adding the negation of a derived fact such as 'Farfalle does not cost \$8', allows us to generate a new sequence of non-redundant explanations up to inconsistency, hence reusing the techniques from the previous section. This novel approach allows us to provide a mechanism for *zooming in* into the more difficult explanation steps of the explanation sequence.

3.5.1 Nested explanation of a reasoning step

We propose the following principles for what constitutes a meaningful and simple nested explanation, given a non-trivial explanation $(\mathcal{E}, \mathcal{S}, \mathcal{N})$:

- a nested explanation starts from the explaining facts $\mathcal{E}$, augmented with the counterfactual assumption of a newly derived fact $n \in \mathcal{N}$;
- at each step, it only uses clues from $\mathcal{S}$;
- each step is easier to understand (has a *strictly* lower cost) than the parent explanation which has cost $f(\mathcal{E}, \mathcal{S}, \mathcal{N})$;
- from the counterfactual assumption, a contradiction is derived.

Note that if an explanation step derives multiple new facts, e.g. $|\mathcal{N}| > 1$, then we can compute a nested explanation for each $n_i \in \mathcal{N}$.

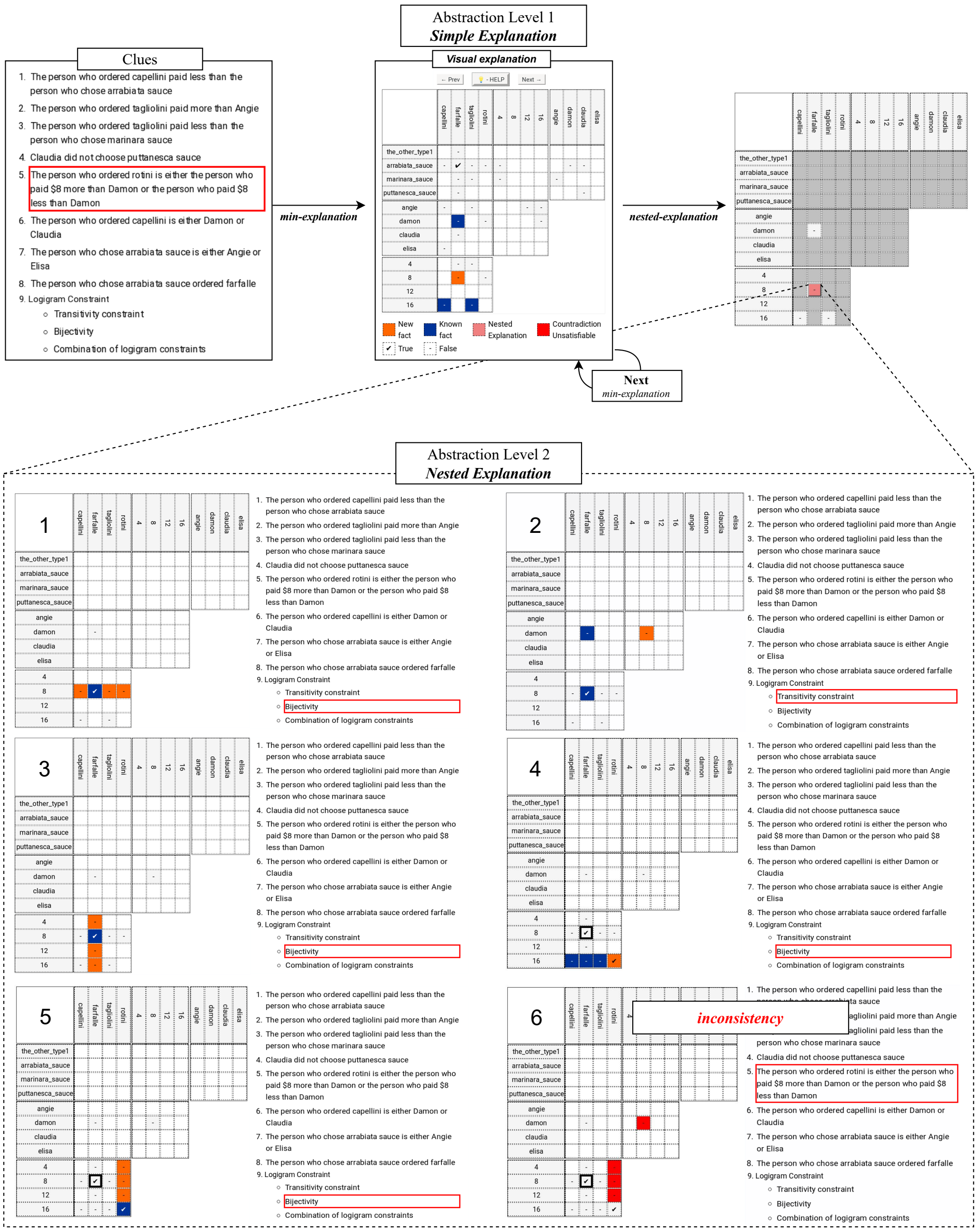
More formally, we define the concept of *nested explanation* as follows:

Definition 20. *The* nested explanation *problem consists of — given a non-redundant explanation $(\mathcal{E}, \mathcal{S}, \mathcal{N})$, and a newly derived fact $n \in \mathcal{N}$ — finding a non-redundant explanation sequence*

$$\langle\ (I'_0, (\emptyset, \emptyset, \emptyset)),\ (I'_1, (\mathcal{E}'_1, \mathcal{S}'_n, \mathcal{N}'_1)), \ldots,\ (I'_n, (\mathcal{E}'_n, \mathcal{S}'_n, \mathcal{N}'_n))\ \rangle$$

such that:

- *I'_0 is the partial interpretation $\{\neg n_i \wedge \mathcal{E}\}$;*
- *$\mathcal{S}'_i \subseteq \mathcal{S}$ for each i;*



1. The person who ordered capellini paid less than the person who chose arrabiata sauce
2. The person who ordered tagliolini paid more than Angie
3. The person who ordered tagliolini paid less than the person who chose marinara sauce
4. Claudia did not choose puttanesca sauce
5. The person who ordered rotini is either the person who paid $8 more than Damon or the person who paid $8 less than Damon
6. The person who ordered capellini is either Damon or Claudia
7. The person who chose arrabiata sauce is either Angie or Elisa
8. The person who chose arrabiata sauce ordered farfalle
9. Logigram Constraint
   - Transitivity constraint
   - Bijectivity
   - Combination of logigram constraints

Figure 3.2: A difficult explanation step, including its nested explanation

- $f(\mathcal{E}'_i, \mathcal{S}'_i, \mathcal{N}'_i) < f(\mathcal{E}, \mathcal{S}, \mathcal{N})$ *for each i;*
- I'_n *is inconsistent; and*
- *a predefined aggregate over the sequence* $(f(\mathcal{E}'_i, \mathcal{S}'_i, \mathcal{N}'_i))_{i \leq n}$ *is minimised.*

We can hence augment each explanation $(\mathcal{E}, \mathcal{S}, \mathcal{N})$ with a set of nested explanations, if they exist. We next discuss algorithms for computing explanations and nested explanations.

3.6 Explanation Generation

In this section, we tackle the goal of searching for a non-redundant explanation sequence that is as simple as possible to understand.

Ideally, we could generate all explanations of each fact in $max(I_0, C)$, and search for the lowest scoring sequence among those explanations. However, the number of explanations for each fact quickly explodes with the number of constraints, and is hence not feasible to compute. Instead, we will greedily construct the sequence, by generating candidates for a given partial interpretation and searching for the smallest one among those.

3.6.1 Sequence construction

We aim to minimise the cost of the explanations of the sequence, measured with an aggregate over individual explanation costs $f(\mathcal{E}_i, \mathcal{S}_i, \mathcal{N}_i)$ for some aggregate like $max()$ or $average()$. The cost function f could for example be a weighted sum of the cardinalities of $\mathcal{E}_i$, $\mathcal{S}_i$ and $\mathcal{N}_i$; in Section 3.7, we discuss a concrete cost function for the use case of logic grid puzzles.

Instead of globally optimising the aggregated sequence cost, we encode the knowledge that we are seeking a sequence of small explanations in our algorithm. Namely, we will greedily and incrementally build the sequence, each time searching for the lowest scoring next explanation, given the current partial interpretation. Such an explanation always exists since the end point of the explanation process $max(I_0, C)$ only contains consequences of I_0 and C.

Algorithm 3.1 formalises the greedy construction of the sequence, which determines $I_{end} = max(I_0, C)$ through propagation and relies on a call to
`ExplainOneStep`$(\mathcal{C}, f, I, I_{end})$ to find the next cost-minimal explanation.

Algorithm 3.1: greedy-explain(C, f, I_0)

```
Input  : A set of constraints $C$, a cost-function $f : S \rightarrow \mathbb{R}$ with $S \subseteq C \wedge I_{end}$, and an
         initial interpretation $I_0$.
Output : An explanation sequence $Seq$
1 $I_{end} \leftarrow$ propagate($C \wedge I_0$)
2 $Seq \leftarrow$ empty sequence
3 $I \leftarrow I_0$
4 while $I \neq I_{end}$ do
5     $(\mathcal{E}, \mathcal{S}, \mathcal{N}) \leftarrow$ ExplainOneStep($\mathcal{C}, f, I, I_{end}$)
6     append $(\mathcal{E}, \mathcal{S}, \mathcal{N})$ to $Seq$
7     $I \leftarrow I \cup \mathcal{N}$
8 end
```

3.6.2 Cost functions and cost-minimal explanations

We use Algorithm 3.3 to generate candidate explanations, but in general our goal is to find cost-minimal explanations.

To guide the search to cost-minimal MUSs, we use the observation that typically a small (1 to a few) number of constraints is sufficient to explain the reasoning. A small number of constraints is also preferred in terms of easy to understand explanations, and hence have a lower cost. For this reason, we will not call *candidate-explanations* with the full set of constraints C, but we will greedily grow the number of constraints using the powerset of C.

We make one further assumption to ensure that we do not have to search for candidates for all possible subsets of constraints. The assumption is that we have an optimistic estimate g that maps a subset S of C to a real number such that

$$\forall\, \mathcal{E}, \mathcal{S}, \mathcal{N} : g(\mathcal{S}) \leq f(\mathcal{E}, \mathcal{S}, \mathcal{N})$$

This is for example the case if f is an additive function, such as $f(\mathcal{E}, \mathcal{S}, \mathcal{N}) = f_1(\mathcal{E}) + f_2(\mathcal{S}) + f_3(\mathcal{N})$ where $g(\mathcal{S}) = f_2(\mathcal{S})$ assuming f_1 and f_3 are always positive.

We can then search for the smallest explanation among the candidates found, by searching among increasingly worse scoring $\mathcal{S}$ as shown in Algorithm 3.2. This is the algorithm called by the iterative sequence generation (Algorithm 3.1).

Every time ExplainOneStep($\mathcal{C}, f, I, I_{end}$) is called with an updated partial interpretation I the explanations should be regenerated. The reason is that for some derivable facts n, there may now be a much easier and cost-effective explanation of that fact. There is one benefit in caching

Algorithm 3.2: `ExplainOneStep`$(\mathcal{C}, f, I, I_{end})$

```
Input  : A set of constraints $C$, a cost-function $f$, a partial interpretation $I$, an
         interpretation to explain $I_{end}$.
Output : A non-redundant explanation $(\mathcal{E}, \mathcal{S}, \mathcal{N})$ with $\mathcal{E} \subseteq I$, $\mathcal{S} \subseteq C$, $\mathcal{E} \cup \mathcal{S} \models \mathcal{N}$
1  $\mathcal{N} \leftarrow I_{end} \setminus I$
2  $best \leftarrow (I, C, \mathcal{N})$
3  for $\mathcal{S} \subseteq p(C)$ ordered ascending by $g(\mathcal{S})$ do
4      if $g(\mathcal{S}) > f(best)$ then
5          break
6      end
7      cand ← best explanation from CandidateExplanations$(\mathcal{S}, I)$
8      if $f(cand) < f(best)$ then
9          $best \leftarrow cand$
10     end
11 end
12 return $best$
```

the *Candidates* across the different iterations, and that is that in a subsequent call, the cost of the most cost-effective explanation that is still applicable can be used as a lower bound to start from. Furthermore, in practice, we cache all candidates and when we (re)compute a MUS for a fact n, we only store it if it is more cost-effective than the best one we have previously found for that fact, across the different iterations.

3.6.3 Candidate generation

The main challenge is finding the lowest scoring (easiest) explanation, among all reasoning steps that can be applied for a given partial interpretation I. We first study how to *enumerate* a set of candidate non-redundant explanations given a set of constraints.

For a set of constraints C, we can first use propagation to get the set of new facts that can be derived from a given partial interpretation I and the constraints C. For each new fact n not in I, we wish to find a non-redundant explanation $(\mathcal{E} \subseteq I, \mathcal{S} \subseteq C, \{n\})$ that explains n (and possibly explains more). Recall from Definition 18 that this means that whenever one of the facts in $\mathcal{E}$ or constraints in C is removed, the result is no longer an explanation. We now show that this task is equivalent to finding a Minimal Unsat Core (or Minimal Unsat Subset, MUS). To see this, consider the theory

$$I \wedge C \wedge \neg n.$$

This theory surely is unsatisfiable since n is a consequence of I and C. Furthermore, under the assumption that $I \wedge C$ is consistent (if it were not, there would be nothing left to explain), *any* unsatisfiable subset of this theory contains $\neg n$. We then see that each unsatisfiable subset of this theory is of the form $\mathcal{E} \wedge \mathcal{S} \wedge \neg n$ where $(\mathcal{E}, \mathcal{S}, \{n\})$ is a (not necessarily redundant) explanation of the derivation of $\{n\}$ from I. Vice versa, each explanation of $\{n\}$ corresponds to an unsatisfiable subset. Thus, the *minimal* unsatisfiable subsets (MUS) of the above theory are in one-to-one correspondence with the non-redundant explanations of n, allowing us to use existing MUS algorithms to search for non-redundant explanations.

We must point out that MUS algorithms typically find *an* unsatisfiable core that is *subset-minimal*, but not *cardinality-minimal*. That is, the unsatisfiable core can not be reduced further, but there could be another minimal unsatisfiable core whose size is smaller. That means that if size is taken as a measure of simplicity of explanations, we do not have the guarantee to find the optimal ones. Optimality is certainly not guaranteed when a cost function is involved.

Algorithm 3.3 shows our proposed algorithm. The key part of the algorithm is on line 4 where we find an explanation of a single new fact n by searching for a MUS that includes $\neg n$. We search for subset-minimal unsatisfiable cores to avoid redundancy in the explanations. Furthermore, once a good explanation $(\mathcal{E}, \mathcal{S}, \mathcal{N})$ is found, we immediately explain all implicants of $\mathcal{E}$ and $\mathcal{S}$. In other words: we take $\mathcal{N}$ to be subset-maximal. The reason is that we assume that all derivable facts that use the same part of the theory and the same part of the previously derived knowledge probably require similar types of reasoning and it is thus better to consider them at once. Thus, we choose to generate candidate explanations at once for all implicants of $(\mathcal{E}, \mathcal{S})$ on line 7. Note that the other implicants $\mathcal{N} \setminus \{n\}$ may have simpler explanations that may be found later in the for loop, hence we do not remove them from $\mathcal{N}^*$.

We assume the use of a standard MUS algorithm, e.g., one that searches for a satisfying solution and if a failure is encountered, the resulting Unsat Core is shrunk to a minimal one [100]. While computing a MUS may be computationally demanding, it is far less demanding than enumerating all MUSs (of arbitrary size) as candidates.

Consider the following example in propositional logic to illustrate how `ExplainOneStep` supports the search for low cost explanation steps by generating candidate explanation steps with the help of `CandidateExplanations`.

Example 14. *Let $\mathcal{C}$ be the CNF formula over atoms x_1, x_2, x_3 with the following four clauses:*

$$c_1 := \neg x_1 \vee \neg x_2 \vee x_3 \qquad c_2 := \neg x_1 \vee x_2 \vee x_3 \qquad c_3 := x_1 \qquad c_4 := \neg x_2 \vee \neg x_3$$

Algorithm 3.3: `CandidateExplanations`(C, I)

```
Input  : A partial interpretation $I$ and a set of constraints $C$
Output : A set of candidate explanations $Candidates$
1  Candidates $\leftarrow \{\}$
2  $\mathcal{N}^* \leftarrow$ propagate$(C \wedge I) \setminus I$
3  for $n \in \mathcal{N}^*$ do
       // Minimal expl. of each new fact:
4      $X \leftarrow$ MUS$(C \wedge I \wedge \neg n)$
5      $\mathcal{E} \leftarrow I \cap X$                          // facts used
6      $\mathcal{S} \leftarrow C \cap X$                          // constraints used
7      $\mathcal{N} \leftarrow$ propagate$(\mathcal{S} \wedge \mathcal{E})$       // all implied facts
8      add $(\mathcal{E}, \mathcal{S}, \mathcal{N})$ to Candidates
9  end
10 return $Candidates$
```

The cost function f is defined as a linear sum

$$f(\mathcal{E},\mathcal{S},\mathcal{N}) = \sum_{c \in S} w_i \cdot c_i + |\mathcal{E}| + |\mathcal{N}| \tag{3.1}$$

over the following clause weights:

$$w_{c_1} = 10 \qquad w_{c_2} = 20 \qquad w_{c_3} = 50 \qquad w_{c_4} = 30$$

The optimistic estimated g is defined as:

$$g(\mathcal{S}) = \sum_{c \in S} w_i \cdot c_i \tag{3.2}$$

Let $I_0 = \{\neg x_2\}$ be the initial interpretation *with corresponding maximal consequence $I_{end} = \{x_1, \neg x_2, x_3\}$ computed by propagating constraints C and initial I_0, see Line 1 of Algorithm 3.1). In the following, we will consider one iteration of the main loop in $Algorithm$ 3.1 that corresponds to extracting an explanation step with `ExplainOneStep`. Algorithm 3.2 first identifies the set of remaining literals to explain $\mathcal{N} = I_{end} \setminus I = \{x_1, x_3\}$. Candidate explanation best $= (I, C, \mathcal{N}) = (\{\neg x_2\}, \{c_1, c_2, c_3, c_4\}, \{x_1, x_3\})$ is used to provide an upper bound on the cost of the best explanation so far.*

By greedily growing the subset of constraints S using the powerset of C (Line 3 of Algorithm 3.2), we aim to search for a better explanation step that is easier to understand than the current best *explanation best.*

Table 3.1 decomposes the intermediate steps of `ExplainOneStep` *in the search for a better explanation candidate computed in* `CandidateExplanations` *for a given subset of constraints $\mathcal{S}$, and its optimistic estimate $g(\mathcal{S})$. Every subset is propagated with interpretation I to check if new facts can be derived. The table shows that no new facts can be propagated until step 6. Propagating $c_3 \wedge \neg x_2$ leads to a new fact $\{x_1\}$.*

Step	$\mathcal{S}$	g($\mathcal{S}$)	$\mathcal{N}^*$	`CandidateExplanations`	*best*	$f(best)$
	—	—	—	—	$(\{\neg x_2\}, \{c_1, c_2, c_3, c_4\}, \{x_1, x_3\})$	103
1	$\{c_1\}$	10	$\emptyset$	$\emptyset$		
2	$\{c_2\}$	20	$\emptyset$	$\emptyset$		
3	$\{c_1, c_2\}$	30	$\emptyset$	$\emptyset$		
4	$\{c_4\}$	30	$\emptyset$	$\emptyset$		
5	$\{c_1, c_4\}$	40	$\emptyset$	$\emptyset$		
6	$\{c_3\}$	50	$\{\mathbf{x_1}\}$	$\{(\emptyset, c_3, \{x_1\})\}$	$(\emptyset, c_3, \{x_1\})$	51
7	$\{c_2, c_4\}$	50	$\emptyset$	$\emptyset$		
8	$\{c_1, c_3\}$	60	—	—	—	—

Table 3.1: Candidate explanation generation

For computing a non-redundant explanation of x_1 with given interpretation $I = \{\neg x_2\}$, we simply extract a

$$MUS(\mathcal{S} \wedge I \wedge \neg \ell) = MUS(\mathcal{S} \wedge \{\neg x_2\} \wedge \{\neg x_1\}) = \{c_3, \neg x_1\}$$

also written as

$$c_3 \Rightarrow x_1$$

with cost 51. The execution is `ExplainOneStep` *is interrupted at step 8 since the cost of $g(\mathcal{S}) > f(best)$, in this case 60 > 51.*

Note that all explanations in the examples are expressed in terms of this logical representation, however, the explanations can be translated back to the original input (e.g. natural language clues, alldifferent constraints, ...).

3.6.4 Searching nested explanations

We extend Algorithm 3.1 to generate new candidate explanations with support for nested explanations as introduced in Section 3.5. Fundamentally, the generated candidate explanations are further decomposed in a nested explanation sequence provided that the sequence is easier than the candidate explanation according to the defined cost function f. We assess the possibility for a nested explanation for every newly derived fact in the candidate explanation. Similar to Algorithm 3.1, Algorithm 3.4 exploits the `ExplainOneStep` function to generate the candidate nested explanations. The only difference is that after computing each explanation step, also a call to *nested-explanations* (which is found in Algorithm 3.5) is done to generate a nested sequence.

Algorithm 3.4: `GreedyExplainSequence`(C, f, I_0)

```
Input  : A set of constraints $C$, a cost function $f$, and an initial interpretation $I_0$
Output : An explanation sequence $Seq$
1 $I_{end} \leftarrow$ propagate$(C \wedge I_0)$
2 $Seq \leftarrow$ empty sequence
3 $I \leftarrow I_0$
4 while $I \neq I_{end}$ do
5     $(\mathcal{E}, \mathcal{S}, \mathcal{N}) \leftarrow$ ExplainOneStep$(\mathcal{C}, f, I, I_{end})$
6     $nested \leftarrow$ ExplainNested$(\mathcal{E}, \mathcal{S}, \mathcal{N})$
7     append $((\mathcal{E}, \mathcal{S}, \mathcal{N}), nested)$ to $Seq$
8     $I \leftarrow I \cup \mathcal{N}$
9 end
```

Given an explanation step $(\mathcal{E}, \mathcal{S}, \mathcal{N})$ and a newly derived fact $n \in \mathcal{N}$ for which we want a more detailed explanation, Algorithm 3.5 first constructs a partial interpretation I' formed by the facts in $\mathcal{E}$ and the negated new fact n. Then, we gradually build the sequence by adding the newly found explanations $(\mathcal{E}', \mathcal{S}', \mathcal{N}')$ as long as the interpretation is consistent, and the explanation is easier than explanation step $(\mathcal{E}, \mathcal{S}, \mathcal{N})$ (this is in line with Definition 20 and serves to avoid the nested explanation being a single-step explanation that is equally difficult as the original step).

The computation of a nested explanation, described in Algorithm 3.5, also reuses `ExplainOneStep`. The main differences with the high level explanation generating algorithm come from the fact that the search space for the next easiest explanation step is bounded by the input explanation: it can use only constraints (and facts) from the original explanation, and the cost of the parent explanation is an upper bound on the acceptable costs at the nested level.

While Algorithm 3.5 tries to find a nested explanation sequence for each derived fact, it will not find one for each fact due to the if-check at Line 9. This check can kick in for two different reasons. The first reason is that the explanation step at the main level is simply too simple to be further broken up in pieces. For instance, the explanation of Figure 1.1 is of that kind: it uses a single bijectivity constraint with a single previously derived fact. Breaking this derivation up in strictly smaller parts would thus not be helpful.

This phenomenon can also occur for difficult steps: sometimes the best nested explanation of a

Algorithm 3.5: `ExplainNested`$(\mathcal{E}, \mathcal{S}, \mathcal{N})$

```
Input  : Explanation facts $\mathcal{E}$, constraints $\mathcal{S}$ and newly derived facts $\mathcal{N}$
Output : A collection of nested explanation sequences nested_explanations
1  $nested\_explanations \leftarrow$ empty set
2  for $n \in \mathcal{N}$ do
3      $store \leftarrow true$
4      $nested\_seq \leftarrow$ empty sequence
5      $I' \leftarrow \mathcal{E} \wedge \neg\{n\}$
6      while consistent($I'$) do
7          $I'_{end} \leftarrow$ propagate($\mathcal{S} \wedge I'$)
8          $(\mathcal{E}', \mathcal{S}', \mathcal{N}') \leftarrow$ ExplainOneStep($S, f, I', I'_{end}$)
9          if $f(\mathcal{E}', \mathcal{S}', \mathcal{N}') \geq f(\mathcal{E}, \mathcal{S}, \mathcal{N})$ then
10             $store \leftarrow false$
11             break
12         end
13         append $(\mathcal{E}', \mathcal{S}', \mathcal{N}')$ to $nested\_seq$
14         $I' \leftarrow I' \cup \mathcal{N}'$
15     end
16     if $store$ then
           // only when all steps simpler than $(\mathcal{E}, \mathcal{S}, \mathcal{N})$
17         append $nested\_seq$ to $nested\_explanations$
18     end
19 end
20 return $nested\_explanations$
```

difficult explanation step contains a step that is as difficult or harder than the high-level step itself. In that case, this is a sign that reasoning by contradiction is not simplifying matters in this step and other methods should be explored to further explain it, for example counterfactual explanations that aim to answer the question: ‘what if Y instead of X?’.

3.7 Explanations for logic grid puzzles

We instantiated the above described algorithms in the context of logic grid puzzles. In that setting, for C, we take C_P for some puzzle P, as described in Definition 13 in Section 3.3. There are three types of constraints in C: *transitivity* constraints, *bijectivity* constraints and *clues*, where the first two follow the same structure in every puzzle and the clues are obtained in a mostly automatic way (see Section 3.8). Before defining a cost-function, and the estimation for g used in our implementation, we provide some observations that drove our design decisions.

Observation 1: Propagations from a single implicit constraint are very easy to understand Contrary to the clues, the implicit constraints (transitivity/bijectivity) are very limited in form and propagations over them follow well-specified patterns. For instance in the case of bijectivity, a typical pattern that occurs is that when $X - 1$ out of X possible values for a given function have been derived not to be possible, it is propagated that the last value should be true; this is visualised for instance in Figure 1.1. Hence, in our implementation, we ensure that they are always performed first. Stated differently, g and f are designed in such a way that $g(\mathcal{S}_1) \geq f(\mathcal{E}, \mathcal{S}_2, \mathcal{N})$ whenever $\mathcal{S}_2$ consists of only one implicit constraint and $\mathcal{S}_1$ does not.

Observation 2: Clues rarely propagate by themselves We observed that the automatically obtained logic representation of clues usually has limited propagation in isolation. This is not a property of the clues, but rather of the final obtained translation. As an example, consider the following sentence: 'The person who ordered capellini is either Damon or Claudia'. From this, a human reasoner might conclude that Angie did not order capellini. However, the (automatically) obtained logical representation is

$$\exists p \in \mathit{person} : \mathit{ordered}(p, \mathit{capellini}) \land (p = \mathit{damon} \lor p = \mathit{claudia}).$$

This logic sentence only entails that Angie did not order capellini *in conjunction with the bijectivity constraint on ordered*. In the natural language sentence, this bijectivity is implicit by the use of the article 'The' which entails that there is a unique person who ordered capellini.

We observed that there is rarely any propagation from sole clues, and that only few implicit constraints are active together with a clue at any time. Because of this last observation, in our implementation for logic grid puzzles we decided not to consider all subsets of implicit constraints in combination with a clue as candidate sets $\mathcal{S}$ in Line 3 in Algorithm 3.2. Instead, we combine each clue with the entire set of all implicit constraints, subsequently counting on the non-redundance of the explanation (the subset-minimality of the core) to eliminate most of the implicit constraints since they are not used anyway.

Observation 3: Clues are typically used independently from other clues A next observation is that in all the puzzles we encountered, human reasoners never needed to combine two clues in order to derive new information and that when such propagations are possible, they are quite hard to explain, and can be split up into derivations containing only single clues. The latter is of course not guaranteed, since one can artificially devise disjunctive clues that do not allow propagation by themselves. Our algorithms are built to handle this case as well, but it turned out to be not necessary in practice: in the puzzles we tested, we never encountered an explanation step that combined multiple clues.

Observation 4: Previously derived facts are easier to use than clues or implicit constraints Our final observation that drove the design of the cost functions is that using previously derived facts is often easier than using an extra clue or implicit constraint. This might be due to the fact that previously derived facts are of a very simple nature while, even implicit constraints contain quantification and are thus harder to grasp. An additional reason for this perceived simplicity is that the derived facts are visualised in the grid.

A cost function With these four observations in mind, we devised f and g as follows (where $nc(\mathcal{S})$ denotes the number of *clues* rather than implicit constraints in $\mathcal{S}$, while M and N are explanation parameters:

$$f(\mathcal{E},\mathcal{S},\mathcal{N}) = basecost(\mathcal{S}) + |\mathcal{E}| + N \cdot |\mathcal{S}|$$

$$g(\mathcal{S}) = basecost(\mathcal{S}) = \begin{cases} 0 & \text{if } |\mathcal{S}| = 1 \text{ and } nc(\mathcal{S}) = 0 \\ M & \text{if } |\mathcal{S}| > 1 \text{ and } nc(\mathcal{S}) = 0 \\ M \cdot nc(\mathcal{S}) & \text{otherwise} \end{cases}$$

The parameter M is taken here to be larger than any reasonable explanation size $|\mathcal{E}| + N \cdot |\mathcal{S}|$ and N to be relatively small in comparison, but slightly larger than a few facts $|\mathcal{E}|$. In our experiments, we use the combination $M = 100$ and $N = 5$ which provided good explanation sequences for the tested puzzles. The effect of this, is that we can generate our subsets $\mathcal{S}$ in Line 3 of Algorithm 3.2 in the following order:

- First all $\mathcal{S}$ containing exactly one implicit constraint.
- Next, all $\mathcal{S}$ containing exactly all implicit constraints and (optionally) exactly one clue.
- Finally, all clue pairs, triples etc. though in practice this is never reached.

Summarised, our instantiation for logic grid puzzles differs from the generic methods developed in the previous section in that it uses a domain-specific optimisation function f and does not consider all $\mathcal{S}$ in Line 3, but only promising candidates based on our observations.

For the complete non-redundant explanation sequence our tool produces on the running example using these scoring functions, we refer to `http://bartbog.github.io/zebra/past`

a. An example of the hardest derivation we encountered (with cost 108), as well as its nested explanation, is depicted in Figure 3.2. It uses several bijectivity constraints for uniqueness of persons, but also for reasoning on the relation between costs and types of pasta, in combination with a clue and three assumptions.

3.8 Logic grid puzzles: From natural language clues to typed first-order logic

The demo system we developed, called ZEBRATUTOR, is named after Einstein's zebra puzzle, which is an integrated solution for solving logic grid puzzles, and for explaining, *in a human understandable way*, how the solution can be obtained from the clues. The input to ZEBRATUTOR is a plain English language representation of the clues and the names of the *entities* that make up the puzzle, e.g. 'Angie', 'arrabiata'.

In typical logic grid puzzles, the entity names are present in the grid that is supplied with the puzzle. For some puzzles, not all entities are named or required to know in advance; a prototypical example is Einstein's Zebra puzzle, which ends with the question 'Who owns the zebra?', while the clues do not name the zebra entity and the puzzle can be solved without knowledge of the fact there is a zebra in the first place.

The complete specification undergoes the following steps, starting from the input:

- **A** **Part-Of-Speech tagging**: A Part-Of-Speech (POS) tag is associated with each word using an out-of-the-box POS tagger [99].
- **B** **Chunking and lexicon building**: A problem-specific lexicon is constructed.
- **C** **From chunked sentences to logic**: Using a custom grammar and semantics a logical representation of the clues is constructed
- **D** **From logic to a complete IDP specification**: The logical representation is translated into the IDP language and augmented with logic-grid-specific information.
- **E** **Explanation-producing search in IDP**: This is the main contribution of this chapter, as explained in Section 3.6.
- **F** **Visualization**: The step-by-step explanation is visualised.

The first three of these steps are related to Natural Language Processing (NLP) and are detailed in section 3.8.1 hereafter. Next, we explain in section 3.8.2 how the IDP specification formed in step *Step D* is used to generate the explanations and visualizations in steps *Step E* and *Step F* respectively. An online demo of our system can be found on `http://bartbog.github.io/zebra`, containing examples of all the steps (bottom of demo page).

3.8.1 Natural Language Processing

Step A. POS tagging

The standard procedure in Natural Language Processing is to start by tagging each word with its estimated Part-Of-Speech tag (POS tag). We use the standard English Penn treebank Pos tagset [99] together with NLTK's built-in perceptron tagger [1] as POS tagger. It uses a statistical inference mechanism trained on a standard training set from the Wall Street Journal. Since any POS-tagger can make mistakes, we manually verify and correct the assigned POS-tags to make sure that all of the puzzle's entities are tagged as noun.

Step B. Chunking and lexicon building

From a natural language processing point of view, the hardest part is step B: automatically deriving the lexicon and building the grammar. The lexicon assigns a role to different sets of words, while the grammar is a set of rules describing how words can be combined into sentences. The goal of this second step is to group the POS-tagged words of the clues into *chunks* that are tagged with lexical categories of which 3 are puzzle-specific: proper nouns for the individual entities that are central to the puzzle, other nouns to groups of entities (like *pasta, sauce*) and transitive verbs that link two entities to each other (e.g. Claudia did not *choose* puttanesca sauce). The other categories are determiner, number, preposition, auxiliary verb etc. and contain a built-in list of possible members. We refer to [29] for full details on the categories.

This process of grouping words is referred to as *chunking*. We use NLTK and a custom set of regular expressions for chunking the proper nouns and different types of transitive verbs. The result is a lexicon where each word or set of words (chunk) is assigned a role based on the POS tags. On top of these roles, we defined a puzzle-independent grammar in the Blackburn and Bos framework [15, 16]. The grammar itself was created based on 10 example training puzzles, and tested on 10 different puzzles to ensure genericity [29].

[1]`http://www.nltk.org/`

Step C. From chunked sentences to logic

Next in *Step C*, the lexicon, partly problem agnostic and partly puzzle-specific, is fed into a type-aware variant of the semantical framework of Blackburn & Bos [15, 16], which translates the clues into Discourse Representation Theory [81]. The typed extension allows us to discover the case where different verbs are used as synonyms for the same inherent relation between two types, e.g. *'ate(person, pasta)'* and *'ordered(person, pasta)'*.

In our system, this is a semi-automated method that suggests a lexicon and lets a user modify and approve it, to compensate for possible 'creativity' of the puzzle designers who tend to insert ambiguous words, or use implicit background knowledge such as using 'in the morning' when there is only one time slot before 12:00.

3.8.2 From logic to visual explanations

Once the types are built, we generate a complete specification which is used by the reasoner, the IDP system [26], to solve the problem.

Step D. From logic to a complete IDP specification

From the types built in *Step C*, we construct the IDP vocabulary containing: all the types and a relation for each transitive verb or preposition. For instance, if the clues contain a sentence 'Claudia did not choose puttanesca sauce', then the vocabulary will contain a binary relation chose($\ldots, \ldots$) with the first argument of type *person* and the second argument of type *sauce*.

After the vocabulary, we construct IDP theories: we translate each clue into the IDP language, and we add implicit constraints present in logic grid puzzles. The implicit constraints stem from the translation of the clues: our translation might generate multiple relations between two types. For instance, if there are clues 'The person who ate taglioni paid more than Angie' and 'The person who ordered farfalle chose the arrabiata sauce', then the translation will create two relations *ate* and *ordered* between persons and pasta. However, we know that there is only one relation between two types, hence we add a theory containing synonymy axioms; for this case concretely:

$$\forall\, x \in person\, \forall\, y \in pasta : ate(x, y) \leftrightarrow ordered(x, y)$$

Similarly, if two relations have an inverse signature, they represent inversion functions, for instance *liked_by* and *ordered* in the clues 'Taglioni is liked by Elisa' and 'Damon ordered capellini'.

In this case we add constraints of the form

$$\forall\, x \in \mathit{person}\, \forall\, y \in \mathit{pasta} : \mathit{liked\_by}(y, x) \leftrightarrow \mathit{ordered}(x, y)$$

Next, we refer back to the end of section 3.3 for examples of the bijectivity and transitivity axioms that link the different relations.

The underlying solver, IDP [26], uses this formal representation of the clues both to solve the puzzle and to explain the solution. We chose the IDP system as an underlying solver since it natively offers different inference methods to be applied on logic theories, including model expansion (searching for solutions), different types of propagation (we used optimal-propagate here to find $max(I, C)$), unsat-core extraction and offers a LUA [73] interface to glue these inference steps together seamlessly [26].

3.9 Experiments

Using logic grid puzzles as a use-case, we validate the feasibility of finding non-redundant explanation sequences and generating nested explanation sequences. As data, we use puzzles from Puzzle Baron's Logic Puzzles Volume 3 [120]. The first 10 puzzles were used to construct the grammar; the next 10 to test the genericity of the grammar. Our experiments below are on test puzzles only; we also report results on the *pasta* puzzle, which was sent to us by someone who did not manage to solve it himself.

As constraint solving engine, we use IDP [26] due to the variety of inference methods it supports natively. The algorithms themselves are written in embedded LUA, which provides an imperative environment inside the otherwise declarative IDP system. The code was not optimised for efficiency and can at this point not be used in an interactive setting, as it takes between 15 minutes to a few hours to fully explain a logic grid puzzle. Experiments were run on an Intel(R) Xeon(R) CPU E3-1225 with 4 cores and 32 Gb memory, running linux 4.15.0 and IDP 3.7.1. The code for generating the explanations of the puzzles in the experiments is available at [55].

3.9.1 Sequence composition

We first investigate the properties of the puzzles and the composition of the resulting sequence explanations. The results are shown in Table 3.2.

The first column is the puzzle identifier, where the puzzle identified as p is the pasta puzzle, our running example. The next 3 columns show the properties of each puzzle: $|type|$ is the number

p	|types|	|dom|	|grid|	# steps	$\overline{\text{cost}}$	1 bij.	1 trans.	1 clue	1 clue+i.	mult i.	mult c.
1	4	5	150	112	27.06	31.25%	50.00%	0.89%	17.85%	0%	0%
2	4	5	150	119	27.77	23.53%	57.14%	1.68%	17.64%	0%	0%
3	4	5	150	110	23.93	32.73%	51.82%	0%	15.46%	0%	0%
4	4	5	150	115	24.54	27.83%	55.65%	2.61%	13.92%	0%	0%
5	4	5	150	122	24.97	24.59%	59.02%	0.82%	15.58%	0%	0%
6	4	5	150	115	22.58	26.96%	58.26%	2.61%	12.18%	0%	0%
7	4	5	150	110	26.79	35.45%	46.36%	0.91%	17.27%	0%	0%
8	4	5	150	118	26.81	33.90%	47.46%	3.39%	15.25%	0%	0%
9	4	5	150	114	24.75	28.95%	54.39%	3.51%	13.16%	0%	0%
p	4	4	96	83	34.45	33.73%	40.96%	1.20%	21.69%	2.41%	0%

Table 3.2: Properties of the puzzles, explanation sequences and constraints used in the explanations.

of types (e.g. person, sauce) while $|dom|$ is the number of entities of each type and $|grid|$ is the number of cells in the grid, i.e. the number of literals in the maximal consequence interpretation $I_n = max(\emptyset, C)$. Coincidentally, almost all the puzzles have 4 types with domain size 5, hence 150 cells, except for the pasta puzzle which has a domain size of 4, thus 96 cells.

Columns 5 and 6 show the amount of steps ($\#steps$) in the explanation sequences found, and $\overline{cost}$ is the average cost of an explanation step in the puzzle. The number of inference steps is around 110-120 for all but the pasta puzzle, which is related to the grid size.

Table 3.2 investigates the proportion of inference steps in the explanation sequence, e.g. the trivial steps using just one bijection constraint (**1 bij.**), one transitivity constraint (**1 trans.**) or one clue (**1 clue**) and no other constraints; and the more complex inference steps using one clue and some implicit (bijectivity or transitivity) constraint (**1 clue+i**), multiple implicit constraints (**mult i.**).

	average nr. of facts used					**% of explanations with a clue that use # facts**				
p	all	Bij.	Trans.	Clue	multi i.	0 facts	1 facts	2 facts	3 facts	>3 facts
1	1.84	2.37	2.00	0.52	-	66.67%	28.57%	0%	0%	4.76%
2	1.85	2.50	2.00	0.61	-	47.83%	47.83%	0%	4.35%	0%
3	1.84	2.33	2.00	0.24	-	82.35%	11.76%	5.88%	0%	0%
4	1.89	2.50	2.00	0.47	-	68.42%	15.79%	15.79%	0%	0%
5	1.85	2.50	2.00	0.35	-	65.00%	35.00%	0%	0%	0%
6	1.84	2.35	2.00	0.29	-	76.47%	17.65%	5.88%	0%	0%
7	1.88	2.31	2.00	0.75	-	55.00%	25.00%	10.00%	10.00%	0%
8	1.86	2.58	2.00	0.18	-	81.82%	18.18%	0%	0%	0%
9	1.85	2.45	2.00	0.32	-	78.95%	15.79%	0%	5.26%	0%
p	1.73	2.07	2.00	0.53	4.00	68.42%	21.05%	0%	10.53%	0%

Table 3.3: Statistics on number of previously derived facts $|E|$ used in the explanation steps.

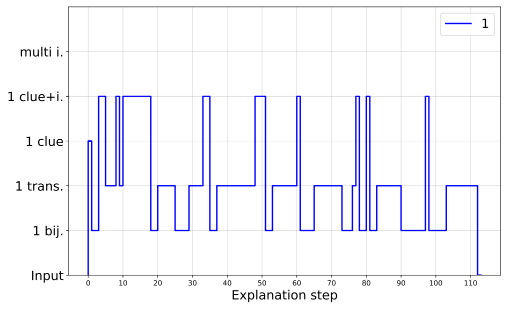
We can observe (see Section 3.7) around 30% of steps are simple bijectivity steps (e.g. completing a row or column in one relation), around 50% are transitivity steps (except for the pasta puzzle) and up to 3% use just a single clue (see Section 3.7). The majority of the steps involving clues use a more complex combination of a clue with other constraints. We see that while our method can combine multiple clues, the explanations generated never require combining multiple clues in one inference step (**mult c.** always 0, see Section 3.7), that is, the method always find simpler steps involving just one clue. Also notably, the puzzles from the booklet never require combining implicit constraints, while the harder pasta puzzle does. In general, less than $1/5$th of the explanations actually need to use a clue or a combination of a clue and implicit constraints. Note that however, for puzzle 3, it is not possible to find new facts based on clue information only, it has to be combined with 1 or multiple constraints.

3.9.2 Sequence progression

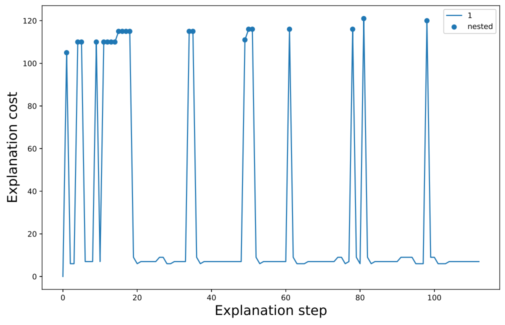
The left side of Figure 3.3 shows a visualization of the type of explanation used in each of the explanation steps for the hardest puzzles 1,2 and p (puzzles with the highest average step cost). We can see that typically at the beginning of the sequence, individual clues (3rd line) and some individual bijectivity (1st line) and transitivity (2nd line) constraints are used, i.e., trivial ones. This is then followed by a series of clues that also involve bijectivity/transitivity constraints, after which a large fraction of the table can be completed with bijectivity/transitivity, followed by a few last clue/implicit constraint combinations and another round of completion. The exception to this is the pasta puzzle. We can see that after around 20 steps where mostly clues have been used, twice a combination of implicit logigram constraints must be used to derive a new fact, after which the table can be easily completed with bijectivity/transitivity and twice the use of a clue.

3.9.3 Explanation size

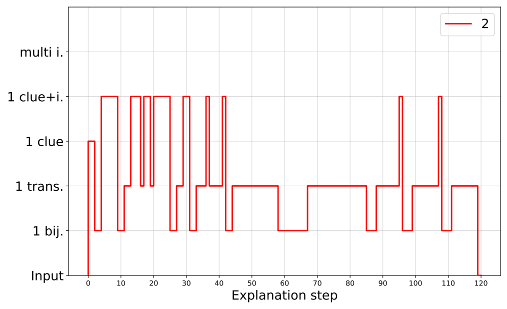
Our cost-function is constructed to favor few (if any) clues and constraints in the explanations, and a small number of previously derived facts $|E|$. Table 3.3, first 5 columns, shows the average number of facts used per type of constraints used in the explanation. We can observe that the average number of facts used is indeed low, less than two (column ‘all’). The breakdown per type of constraint shows that bijectivity typically uses more facts: it either uses three ‘negative’ facts in one row to infer a ‘positive’ fact, as in Figure 1.1 or it uses one ‘positive’ fact to infer three negative facts. Note that for such an intuitive constraint, the number of facts used does not matter much. Transitivity, by nature, always uses two previously derived facts. When an explanation involves a clue, few facts are involved on average.



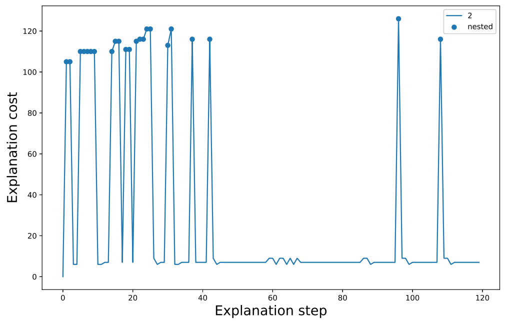
(a) Puzzle 1.



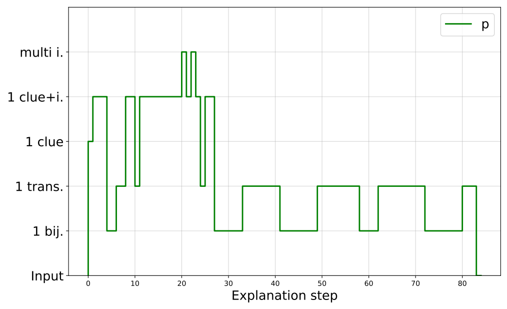
(b) Puzzle 1.



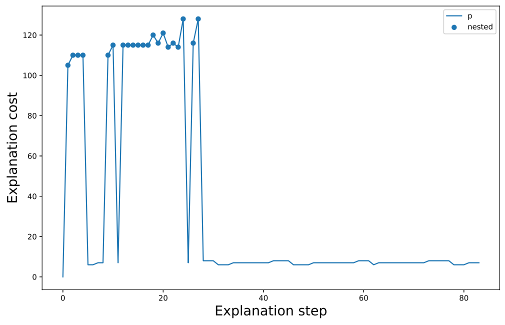
(c) Puzzle 2.



(d) Puzzle 2.



(e) Pasta puzzle.



(f) Pasta puzzle.

Figure 3.3: Side-by-side comparison of puzzle composition (left) and puzzle complexity with nested explanation steps highlighted (right).

The rest of the columns take a closer look at the number of facts used when the explanation involves a clue. We can see that our approach successfully finds small explanations: many clues (the trivial ones) use no facts, while some use 1 fact and only occasionally 2 or more facts are needed. The puzzles 1, 2, 9 and pasta require the use of 3 facts or more together with a clue corresponding to the puzzles with the highest amounts of clues with implicit constraints. Only puzzle 1 requires clues to be combined with more than 3 facts. Even though puzzle 9 has a lower cost, the fact it uses more than 3 facts combined with a clue is linked to a complex clue formulation.

Notably, the amount of facts for explanations requiring clues, is equal to 0 half of the time, which means that the new information can be derived independently of other new facts. Part of the clues with 1 fact are linked to the clue involving constraints, namely clues combined with bijectivity as they can derive 3 new facts from only 1 fact. Altogether clues with 0 or 1 facts form more than 80% of the explanations using clues with facts.

3.9.4 Nested explanations

We next investigate the explanation cost of the different steps in an explanation sequence, and which ones we can find a nested explanation for. The right side of Figure 3.3 shows the explanation cost of each explanation step for puzzles 1, 2 and p; for each step it is also indicated whether or not a nested explanation was found (those where one was found are indicated by a dot in the figure). The first observation we can draw from the side-by-side figures, is peaks with nested explanations (on the right) overlap with almost all peaks on the left. Simply put, for each of the non-trivial explanations, we are most often able to find a nested explanation that can provide a more detailed explanation using contradiction. Secondly, we observe that the highest cost steps involve more than one constraint: either a clue and some implicit constraints, or a combination of implicit constraints (only in the pasta puzzle). The harder pasta puzzle also has a higher average cost, as was already reported in Table 3.2.

As the explanation sequence progresses, the cost of difficult explanation steps also increases, meaning that we can find easy explanations in the beginning, but we will require more complex reasoning to find new facts that unlock more simpler steps afterwards. This is especially visible from the trend line that fits all each puzzle's nested explanation dots on the right side of Figure 3.3.

We now have a closer look at the composition of these nested explanations in Table 3.4. When looking at the percentage of reasoning steps that have a nested explanation, we see that only a fraction of all steps have a nested explanation. As the figures already indicated, when looking at the non-trivial steps, we see that for all of those, our method is able to generate a nested sequence that explains the reasoning step using contradiction.

		% steps with nested			average steps	Composition of nested explanation				
p	**# steps**	**of all**	**of clue+i.**	**of m-i**	**nested expl.**	**1 bij.**	**1 trans.**	**1 clue**	**1 clue+i.**	**mult i.**
1	112	14.29%	100%	-	2.94	36.99%	32.88%	12.33%	17.81%	0%
2	119	14.29%	100%	-	2.65	48.28%	10.34%	20.69%	20.69%	0%
3	110	7.27%	100%	-	2.00	50.00%	0%	0%	50.00%	0%
4	115	13.04%	100%	-	4.04	40.26%	29.87%	20.78%	9.09%	0%
5	122	13.11%	100%	-	2.29	47.83%	0%	8.70%	43.48%	0%
6	115	10.43%	100%	-	2.56	50.00%	8.00%	20.00%	22.00%	0%
7	110	15.45%	100%	-	3.29	37.50%	32.69%	18.27%	11.54%	0%
8	118	9.32%	100%	-	2.36	43.75%	9.38%	25.00%	21.88%	0%
9	114	10.53%	100%	-	3.50	31.25%	28.12%	20.31%	20.31%	0%
p	83	16.87%	100%	100%	3.07	48.44%	20.31%	7.81%	17.19%	6.25%

Table 3.4: Properties of the nested explanations.

When looking at the number of steps in a nested sequence, we see that the nested explanations are rather small: from 2 to 4 steps. The subsequent columns in the table show the composition of the nested explanations, in terms of the types of constraints used. The majority of those are simple steps involving just a single constraint (a single bijectivity, transitivity or clue). Interestingly, even in nested explanations which typically explain a '1 clue+i.' step, the nested explanation also has a step involving a clue and at least one other implicit constraint. Detailed inspection showed this was typically the final step, which is a contradiction in the clue as in Figure 3.2.

The composition of the nested sequence also shows that the nested explanations were so simple that they cannot be further decomposed into a second (deeper) level of nested explanations. This is due to the way our cost function is defined, i.e. promoting the use of *simple* constraints and heavily penalizing combinations of clues and/or constraints.

We can further look at the *cost* of the different steps in a nested explanation, which we use as a proxy for difficulty of the steps. Figure 3.4 displays a violin plot of the ratio of the cost of a nested step divided by the cost of the original parent step that it is contributing to explain; a wider region indicates a higher density of steps having that ratio. Note that by Definition 20, a step in a nested explanation can never be as costly or more costly than its parent step (always below the top orange line, though it often comes close). The figure shows us that there are many steps with a cost of only a fraction of the parent step (near 0-10% of the original cost), but also some steps closer but still a bit simpler than the parent step. Due to the construction of the cost function in Section 3.7 with each clue contributing a value of '100' and each fact only a value of '1', this means it typically involves fewer previously derived facts, which should make it easier to understand.

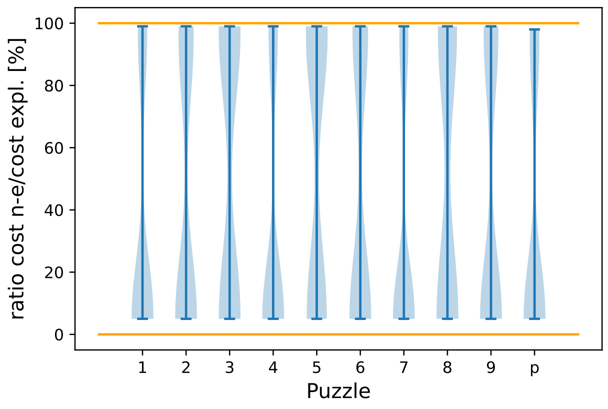


Figure 3.4: Distribution of the ratio of the cost of a nested explanation sequence step and the cost of original explanation step for all puzzles.

3.10 Discussion, future work, and conclusions

In this chapter, we formally defined the problem of step-wise explanation generation for constraint satisfaction problems, as well as presenting a generic algorithm for solving the problem. We extended the mechanism so that it can be used in a *nested* way, to further explain an explanation. We developed the algorithm in the context of logic grid puzzles where we start from natural language clues and provide a human-friendly explanation in the form of a visualization.

Observations

When investigating the nested explanation in Figure 3.2 as it is produced by the system, one can observe that this explanation does not entirely match an explanation that would be produced by a human reasoner for a couple of reasons:

- In the generation of the nested explanation, as well as in the high-level sequence, we used the greedy algorithm from section 3.6. While at the high level, this yields good results, at the nested level, this results in sometimes propagating facts that are not used afterwards. The very first propagation in the nested explanation is of this kind. While this would be easy to fix by post-processing the generated explanation, we left it in our example to highlight this difference between the nested and non-nested explanation.

- It sometimes happens that the system finds, as a minimal explanation, one in which $X - 1$ negative facts are used instead of the corresponding single positive fact. This can be seen in the last step. For human reasoners the positive facts often seem to be easier to grasp. A preference for the system towards these negative facts might be incidentally due to the formulation of the clues, or it can incidentally happen due to the way the MUS is computed (only subset-minimality is guaranteed there), or the granularity of the information that can be captured by the cost function such as a distinction between positive and negative facts. In general, observations of this kind should be taken into account when devising a cost function.

- A last observation we made (but that is not visible in the current figure) is that sometimes the generated nested explanations seem to be unnecessarily difficult to understand. In all cases we encountered where that was the case, the explanation was the same: the set of implicit constraints contains a lot of redundant information: a small number of them would be sufficient to imply all the others. Our cost function, and the subset-minimality of the generated MUS entails that in the explanation of a single step, implicit constraints will never be included if they follow from other included implicit constraints. However, when generating the nested explanations, it would actually be preferred to have those redundant constraints, since they allow breaking up the explanation in simpler parts, e.g., giving a simple step with a single bijectivity, rather than a complex step that uses a combination of multiple implicit constraints.

These observations suggest that further research into the question *what constitutes an understandable explanation for humans*. In Chapter 5, we propose learning how to characterise which is the most helpful explanation directly from users.

Future work

Bleukx et al. [17] noticed that there were a lot of redundant steps and unnecessarily complex parts when generating a sequence for explaining unsatisfiability. Similar to our nested explanations generated using reasoning by contradiction, Bleukx et al. [17] first build a greedy explanation sequence and filter out input facts and derived facts that are not necessary in reaching the contradiction. While that approach can be directly applied for improving the generation of nested explanations, there is no clear way of applying it to explaining satisfiability. More precisely, we are interested in explaining all facts of a solution, meaning no steps can be left out. However, as in [17], we build our sequence greedily, focussing on finding the next best explanation given an interpretation. Therefore, additional directions to produce easier-to-understand explanations would be to optimise the sequence as a whole, rather than only individual steps.

Additionally, on our puzzles, the nested explanations were so simple that no further refinement was needed. For more difficult problems, this may no longer be the case; as such, a possible avenue for future work is a generic approach to multi-level nested explanations. Another interesting direction is to research whether nesting can be related to existing notions of abstraction and refinement [90, 123, 105].

With respect to *efficiency*, the main bottleneck of the current algorithm is having to rely on many calls to MUS, which is a hard problem by itself, to ensure we find low cost explanations. In Chapter 4, we explore methods to perform *unsatisfiable subset optimisation* [54] to guarantee optimality of an explanation with respect to a given cost-function.We investigate how to take advantage of the explanation sequence generation context to make our approach (near) real-time, essentially allowing explanations to be generated *while* a user is solving the puzzle. This will become especially important when we implement our ideas in more critical domains, such as *interactive configuration*, where a human and a search engine cooperate to solve a configuration problem and the human can often be interested in understanding *why* the system did certain derivations [68, 25].

Finally, while most of the steps in the pipeline used to derive a typed first-order logic from natural language clues are largely automated, some steps still require human intervention. For instance, our system suggests a lexicon and lets a user modify and approve it, to compensate for possible "creativity" of the puzzle designers who tend to insert ambiguous words, or use implicit background knowledge such as using 'in the morning' when there is only one timeslot before 12:00. An possible avenue for future work is to whether recent large language models, LLMs fine-tuned for code generation or mathematical reasoning, can be leveraged to generate typed FOL specification directly from the natural language clues [77, 107].

CHAPTER 4

Efficiently Explaining CSPs with Unsatisfiable Subset Optimization

Explanations of inference steps are computed by extracting a Minimal Unsatisfiable Subset (MUS) of a derived formula. A cost function is used to evaluate the difficulty of understanding an explanation. However, since MUS extraction methods do not provide a guarantee of optimality for a given cost function, a heuristic must be used to ensure that low-cost explanations are found. This chapter explores how to efficiently generate explanations that are guaranteed to be optimal (MUSs), and how to exploit the fact that we are generating a sequence of explanations incrementally.

This chapter is largely based on the following scientific publications:

Emilio Gamba, Bart Bogaerts, and Tias Guns. Efficiently explaining csps with unsatisfiable subset optimization. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence, IJCAI*, pages 1381–1388, 2021. doi: 10.24963/ijcai.2021/191

Emilio Gamba, Bart Bogaerts, and Tias Guns. Efficiently explaining csps with unsatisfibale subset optimization. *Journal of Artificial Intelligence Research*, 78:709–746, 2023. doi: 10.1613/jair.1.14260

4.1 Introduction

In Chapter 3, we decompose explaining the maximal consequence of a CSP as a sequence of *simple* inference steps, where a cost function is used to approximate human understandability of the corresponding explanation step. The explanation generation algorithms in the previous chapter rely on extracting Minimal Unsatisfiable Subsets (MUSs) of a derived unsatisfiable formula, exploiting a one-to-one correspondence between so-called *non-redundant explanations* and MUSs.

Experiments have shown that such a MUS-based approach can easily take hours, especially when, at every explanation step, repeated MUS calls are performed for each remaining literal to explain (line 3 of Algorithm 3.3) to increase the chance of finding a low-cost MUS. Furthermore, MUS extraction algorithms do not provide any guarantee of $\subseteq$-minimality or optimality with respect to a given cost function f. Therefore, we handle the absence of $\subseteq$-minimality and optimality guarantees by heuristically considering increasingly larger subsets of the unsatisfiable formula ($\mathcal{C} \wedge I \wedge \neg l$), as shown on line 3 of Algorithm 3.2.

Hence, there is a need for algorithmic improvements to make it more practical. We see three main points of improvement, all of which will be tackled by our generic OCUS algorithm presented in Section 4.2.

- First, since the algorithm is based on `MUS` calls, there is no guarantee that the explanation found is indeed optimal (with respect to the given cost function). Performing multiple MUS calls is only a heuristic that is used to circumvent the restriction that *there are no algorithms for cost-based unsatisfiable subset **optimization***.
- Second, this algorithm uses `MUS` calls for every literal $\ell \in I_{end} \setminus I$ to explain separately. The goal of all these calls is to find a single unsatisfiable subset of $\mathcal{C} \wedge I \wedge \overline{(I_{end} \setminus I)}$ that contains *exactly one literal* from $\overline{(I_{end} \setminus I)} = \{\neg\ell \mid \ell \in I_{end} \setminus I\}$. This begs whether it is possible *to compute a single (optimal) unsatisfiable subset **subject to constraints***, where in our case, the constraint is to include exactly one literal from $\overline{(I_{end} \setminus I)}$.
- Third, the algorithm that computes an entire explanation sequence makes use of repeated calls to `ExplainOneStep` and hence will solve many similar problems. This raises the issue of ***incrementality**: can we re-use the computed data structures to achieve speed-ups in later calls?*

Contributions In this chapter, we tackle the limitations discussed above and improve the state of the-art in the following ways:

- We develop algorithms that compute (cost-)**Optimal** Unsatisfiable Subsets (from now on called OUSs) based on the well-known hitting-set duality that is also used for computing cardinality-minimal MUSs [74, 122].
- We observe that in the explanation setting, many of the individual calls for MUSs (or OUSs) can actually be replaced by a single call that searches for an optimal unsatisfiable subset *among subsets satisfying certain structural constraints*. We formalise and generalise this

observation by introducing the *Optimal* ***Constrained*** *Unsatisfiable Subsets (OCUS)* problem. We then show how $O(n^2)$ calls to MUS/OUS can be replaced by $O(n)$ calls to an OCUS oracle, where n denotes the number of facts to explain.

- We develop techniques for *optimizing* the O(C)US algorithms further, exploiting domain-specific information coming from the fact that we are in the *explanation-generation context*. Such optimizations include
  1. the development of methods for *information re-use* between consecutive O(C)US calls; as well as
  2. an explanation-specific version of the OCUS algorithm.
- Finally, we extensively evaluate our approaches on a large number of CSP problems.

Chapter structure. Section 4.2 introduces the OCUS problem, and a hitting set-based algorithm for computing OCUSs. In Section 4.3, we study 2 other methods for computing the next best explanation in the explanation sequence. Then, in Section 4.4 we present methods to improve the efficiency of our algorithms, e.g. by making them incremental or by using different methods to extract multiple correction subsets. Finally, in Section 4.5 we evaluate our approach on a large set of puzzle instances and conclude with some future perspectives.

4.2 Optimal Constrained Unsatisfiable Subsets

In this section, we address the first two challenges, namely

1. the *lack of optimality guarantees* when relying on MUS extraction methods (and heuristics) to compute an explanation step, and
2. while the optimal explanation of a single literal can be formalised as an *optimal* MUS (with respect to a given objective), finding the optimal next explanation step over all literals remains an open question.

To tackle these, we introduce the concept of an *Optimal Constrained Unsatisfiable Subset (OCUS)* and propose an algorithm for computing one.

Definition 21. *Let $\mathcal{F}$ be a formula, $f : 2^{\mathcal{F}} \rightarrow \mathbb{N}$ a cost function and p a predicate $p : 2^{\mathcal{F}} \rightarrow \{true, false\}$. We call $S \subseteq \mathcal{F}$ an* Optimal Constrained Unsatisfiable Subset (OCUS) *of F(with respect to f and p) if*

- *S is unsatisfiable,*
- *$p(S)$ is true*
- *all other unsatisfiable $S' \subseteq \mathcal{F}$ for which $p(S')$ is true satisfy $f(S') \geq f(S)$.*

The purpose of the predicate p is to allow additional structural constraints to be imposed on the set of solutions to the OCUS problem.

If no predicate p is involved, *Optimal Constrained Unsatisfiable Subset (OCUS)* is reduced to an *Optimal Unsatisfiable Subsets (OUS)*.

Definition 22. *Let $f : 2^{\mathcal{F}} \rightarrow \mathbb{N}$ be a cost function. We call $S \subseteq \mathcal{F}$ an* Optimal Unsatisfiable Subsets (OUS) *of F(with respect to f) if is S is unsatisfiable, and all other unsatisfiable $S' \subseteq \mathcal{F}$ for which $p(S')$ is true satisfy $f(S') \geq f(S)$.*

Proposition 2. *Let $\mathcal{F}$ be a CNF formula, p be a predicate specified as a CNF over (meta)-variables indicating the inclusion of clauses of* F*, and f be a cost-function obtained by assigning a weight to each such meta-variable, then the problem complexity of finding an OCUS is Σ_2^P-*complete.

Proof. If we assume that the predicate p is specified itself as a CNF over (meta)-variables indicating the inclusion of clauses of F, and f is obtained by assigning a weight to each such meta-variable, then the complexity of the problem of finding an OCUS is the same as that of the SMUS (cardinality-minimal or 'Smallest' MUS) problem [74]: the associated decision problem is Σ_2^P-complete. Hardness follows from the fact that SMUS is a special case of OCUS; containment follows - intuitively - from the fact that this can be encoded as an $\exists\forall$-QBF using a Boolean circuit encoding of the costs. □

The following properties hold for cost functions:

Definition 23. *Let f be a cost function assigning a cost to each set of constraints $S \subseteq \mathcal{F}$. f is* strictly monotonically increasing *if $S' \subsetneq S$, then $f(S') < f(S)$. A cost function f is called* linear *if it maps each S to $\sum_{c \in S} w_c$ for given weights $w_c \in \mathbb{R}$.*

Definition 24. *Let f be a cost function assigning a cost to each set of constraints $S \subseteq \mathcal{F}$, an O(C)US is also guaranteed to be a MUS, if the cost function f is* strictly monotonically increasing.

Recall the one-to-one correspondence between MUSs of $I \wedge C \wedge \neg n$ and non-redundant explanations of $(\mathcal{E} \subseteq I, \mathcal{S} \subseteq C, \{n\})$ introduced in Section 3.6.3.

Definition 25. *An optimal explanation step, with respect to a given cost function f, is said to be non-redundant if f is strictly monotonically increasing.*

In the next sections, we first explain how OCUS can be used to compute an explanation step and then propose an algorithm for computing OCUSs using the well-known hitting set duality between MUSs and MCSs [94, 117], see Proposition 1.

4.2.1 `ExplainOneStep` with OCUS

`ExplainOneStep` implicitly uses an "exactly one of" constraint on the set of literals to explain $\{\ell \in I_{end} \setminus I\}$, and the way this is done is by considering each literal ℓ from that set separately. It searches for a good, but not necessarily *optimal* CUS (an Unsatisfiable Subset that satisfies the Constraint in question). The goal is however to find an optimal one. Therefore, when considering the procedure `ExplainOneStep` from the perspective of OCUS defined above, the task of the procedure is to compute an OCUS of the formula $\mathcal{F} := \mathcal{C} \wedge I \wedge \overline{I_{end} \setminus I}$ with p the predicate that holds for subsets that contain *exactly one* literal of $\overline{I_{end} \setminus I}$. The pseudocode for this is shown in Algorithm 4.1.

Algorithm 4.1: `ExplainOneStep-OCUS`$(\mathcal{C}, f, I, I_{end})$

```
1 $\mathcal{F} \leftarrow \mathcal{C} \wedge I \wedge \overline{I_{end} \setminus I}$
2 $p \leftarrow (S \mapsto |S \cap \overline{I_{end} \setminus I}| = 1)$
3 $S^*, status \leftarrow$ OCUS$(\mathcal{F}, f, p)$
4 if $status = FAILURE$ then return $\emptyset$
  // Mapping to explanation step
5 $\mathcal{E} \leftarrow I \cap S^*$
6 $\mathcal{S} \leftarrow \mathcal{C} \cap S^*$
7 $\mathcal{N} \leftarrow$ propagate$(\mathcal{E} \wedge \mathcal{S})$
8 return $(\mathcal{E}, \mathcal{S}, \mathcal{N})$
```

Explaining 1 Literal per Step. We define the 'exactly one' constraint p as the equality constraint shown on line 2 of Algorithm 4.1. p evaluates to *true* if the size of the intersection of the negated literals to explain $I_{end} \setminus I$ with the subset S of $\mathcal{F}$ considered is equal to 1, otherwise p evaluates to *false*.

The call to `OCUS` at line 3 of `ExplainOneStep-OCUS` will never lead to the $FAILURE$ case, since an OCUS is guaranteed to exist for the input cost function f, predicate p, and formula $\mathcal{F}$ we consider. Failure can occur for a different predicate p; consider for example the predicate

$p : (S \mapsto f(S) \leq v)$ of Algorithm 4.5 that evaluates to true if a subset S has a value lower than a given bound v. In this case, it might lead to the failure case if no OCUS for a given bound v exists.

In the next section, we provide the details on how to compute an OCUS on line 3.

4.2.2 Computing an OCUS

In order to compute an OCUS of a given formula, we propose to build on the hitting set duality of Proposition 1. For this, we will assume to have access to a solver `CondOptHS` that can compute hitting sets of a given collection of sets that are *optimal* (w.r.t. a given cost function f) among all hitting sets *satisfying a condition* p. The choice of the underlying hitting set solver will thus determine which types of cost functions and constraints are possible.

Cost function In our implementation, we use a cost function f which is encoded as a linear term (weighted sum) similar to the cost function defined after Observation 4 in Section 3.7, where e.g. constraints are given a larger weight than already derived literals. For example, (unit) clauses representing previously derived facts can be given small weights, and regular clauses can be given large weights, so that explanations are penalized for including clauses when previously derived facts can be used instead.

Condition p can easily be encoded as a linear constraint (see Equation (4.8) for an example), thus allowing the use of highly optimised mixed integer programming (MIP) solvers for computing optimal hitting sets. In the following, we explain how the conditional optimal hitting set problem `CondOptHS` can be encoded into MIP to reason over combinations of clauses and literals (hitting sets) of the unsatisfiable formula.

Hitting Set Problem Given $\mathcal{F}$, we define a MIP decision variable d_c for every clause $c \in \mathcal{F}$ and write $D = \{d_c \mid c \in \mathcal{F}\}$ for the set of all such variables. We assume a given collection of sets-to-hit $\mathcal{H}$. The goal is to find a hitting set $h \subseteq D$ that hits every set-to-hit at least once (Equation (4.3)), that satisfies predicate p (Equation (4.2)) and minimises $f(D)$ (Equation (4.1)).

The `CondOptHS` formulation is as follows:

$$\underset{h \subseteq D}{\text{minimise}} \quad f(h) \tag{4.1}$$

$$s.t. \quad p(h) \tag{4.2}$$

$$\sum_{c \in H} d_c \geq 1, \qquad \forall H \in \mathcal{H} \tag{4.3}$$

$$d_c \in \{0,1\}, \qquad \forall d_c \in D \tag{4.4}$$

In the case of `OCUS`, every set-to-hit corresponds to a correction subset of $\mathcal{F}$.

OCUS Algorithm Our generic algorithm for computing OCUSs is depicted in Algorithm 4.3. It combines the hitting set-based approach for MUS of Ignatiev et al. [74] depicted in Algorithm 4.2[1] with the use of a MIP solver for (weighted) hitting sets as proposed for maximum satisfiability by Davies and Bacchus [34]. The key novelties are the ability to add structural constraints to the hitting set solver, without impacting the duality principles of Proposition 1, as we will show.

Algorithm 4.2: `SMUS`($\mathcal{F}$)

```
1 $\mathcal{H} \leftarrow \emptyset$
2 while true do
3     $h \leftarrow$ MinimumHS($\mathcal{H}$)
4     $\mathcal{S} \leftarrow \{c_i | e_i \in h\}$
5     if not $SAT(\mathcal{S})$ then
6         return $\mathcal{SMUS} \leftarrow S$
7     $K \leftarrow grow(S)$
8     $\mathcal{H} \leftarrow \mathcal{H} \cup K$
9 end
```

Algorithm 4.3: `OCUS`($\mathcal{F}, f, p$)

```
1  $\mathcal{H} \leftarrow \emptyset$
2  while true do
3      $S, status \leftarrow$ CondOptHS($\mathcal{H}, f, p$)
4      if $status = FAILURE$ then return ($\emptyset, status$)
5      if not $SAT(S)$ then
6          return ($S, status$)
7      end
8      $K \leftarrow$ CorrSubsets($S, \mathcal{F}$)
9      $\mathcal{H} \leftarrow \mathcal{H} \cup K$
10 end
```

Both the `SMUS` and the `OCUS` algorithm alternate calls to a hitting set solver with calls to a `SAT` oracle on a subset S of $\mathcal{F}$.

In case the `SAT` oracle returns true, i.e., subset S is satisfiable, the `grow` procedure of `SMUS` expands a satisfiable subset S of F further such that its complement, a single correction subset K is as small as possible. Shrinking the correction subset as a result of the `grow` finds stronger constraints on the sets-to-hit, since it restricts the choice on the clauses that need to be selected.

[1]Note that we have replaced variables $\mathcal{F}'$ and $\mathcal{C}$ in the original `SMUS` algorithm with variables S and C respectively to facilitate the comparison between the `SMUS` and the `OCUS` algorithm.

In `OCUS`, the `CorrSubsets` procedure extracts a non-empty set of subsets of $F \setminus S$, more precisely correction subsets, and added to the collection of sets-to-hit H. Furthermore, he calls for hitting sets will also take into account the cost (f), as well as the meta-level constraints (p); as such, it is not clear a priori which properties a good `CorrSubsets` function should have here. In Section 4.2.3, we propose different domain-specific methods for enumerating multiple correction subsets.

For *correctness* of the algorithm, all we need to know is that for a given satisfiable subset S, `CorrSubsets` returns a *non-empty* set of correction subsets K where $\forall\, C \in K : C \subseteq (F \setminus S)$. At any given step, if S is satisfiable, $\mathcal{H}$ is guaranteed to grow, since a non-empty set of correction subsets K is returned that is disjoint from S. Therefore, K cannot be present in $\mathcal{H}$. *Completeness* and *soundness* of the algorithm follow from the fact that the algorithm is guaranteed to terminate since there is a countable number of correction subsets of $\mathcal{F}$, and from Theorem 1, which states that what is returned is indeed a solution and that a solution will be found if it exists.

Theorem 1. *Let $\mathcal{H}$ be a set of correction subsets of F. If S is a hitting set of $\mathcal{H}$ that is f-optimal among the hitting sets of $\mathcal{H}$ satisfying a predicate p, and S is unsatisfiable, then S is an OCUS of $\mathcal{F}$. If $\mathcal{H}$ has no hitting sets satisfying p, then $\mathcal{F}$ has no OCUSs.*

Proof. For the first claim, it is clear that S is unsatisfiable and satisfies p. Hence, all we need to show is f-optimality of S. If there would exist some other unsatisfiable subset S' that satisfies p with $f(S') \leq f(S)$, we know that S' would hit every minimal correction set of F, and hence also every set in $\mathcal{H}$ (since every correction set is the superset of a minimal correction set). Since S is f-optimal among hitting sets of $\mathcal{H}$ satisfying p and S' also hits $\mathcal{H}$ and satisfies p, it must thus be that $f(S) = f(S')$.

The second claim immediately follows from Proposition 1 and the fact that an OCUS is an unsatisfiable subset of $\mathcal{F}$. □

Perhaps surprisingly, the correctness of the proposed algorithm does *not* depend on the monotonicity properties of f nor p. In principle, any (computable) cost function and condition on the unsatisfiable subsets can be used. In practice, however, one is bound by the limitations of the chosen hitting set solver.

As an illustration, we provide an example of one call to `ExplainOneStep-OCUS` (Algorithm 4.1) and the corresponding `OCUS`-call in detail (Algorithm 4.3) for our running example:

Example 15. *Consider the 4 clauses of our running example $\mathcal{C} := c_1 \wedge c_2 \wedge c_3 \wedge c_4$ with:*

$$c_1 := \neg x_1 \vee \neg x_2 \vee x_3 \qquad c_2 := \neg x_1 \vee x_2 \vee x_3 \qquad c_3 := x_1 \qquad c_4 := \neg x_2 \vee \neg x_3$$

Consider a call to `ExplainOneStep-OCUS` *with* $I = \emptyset$ *and* $I_{end} = \{x_1, \neg x_2, x_3\}$. *We add the following three new clauses, which represent the complement of literals to-be-derived* $\overline{I_{end} \setminus I}$*:*

$$\{\neg x_1\} \qquad \{x_2\} \qquad \{\neg x_3\}$$

The cost function f *is defined as a linear sum*

$$f(S) = \sum_{c \in S} w_i \cdot c_i \tag{4.5}$$

over the following clause weights:

$$\textbf{Clause weights: } w_{c_1} = 60 \qquad w_{c_2} = 60 \qquad w_{c_3} = 100 \qquad w_{c_4} = 100$$
$$I \wedge \overline{I_{end} \setminus I} \textbf{ weights: } w_{\neg x_1} = 1 \qquad w_{x_2} = 1 \qquad w_{\neg x_3} = 1$$

We encode the same cost function in the MIP encoding as a weighted sum using the corresponding decision variables as follows:

$$f(S) = \sum_{c \in S} w_c \cdot d_c \tag{4.6}$$

To ensure that we only explain one literal at a time, we add predicate p *as*

$$p(S) := |S \cap \{\neg x_1, x_2, \neg x_3\}| = 1 \tag{4.7}$$

The MIP encoding of predicate p *corresponds to*

$$p(h) := \sum_{c \in \overline{I_{end} \setminus I}} d_c = d_{\{\neg x_1\}} + d_{\{x_2\}} + d_{\{\neg x_3\}} = 1 \tag{4.8}$$

At Line 3 of `ExplainOneStep-OCUS`*,* F *is constructed, consisting of :*

$$\mathcal{F} = \mathcal{C} \wedge I \wedge \overline{(I_{end} \setminus I)} = c_1 \wedge \cdots \wedge c_4 \wedge \neg x_1 \wedge x_2 \wedge \neg x_3 \tag{4.9}$$

Finally, for generating an explanation step, we proceed to call `OCUS` *on formula* $\mathcal{F}$ *with given cost function* f *and* exactly-one *constraint* p*:*

$$\texttt{OCUS}(\mathcal{F}, f, p) \tag{4.10}$$

In this small example, the `CorrSubsets`$(S, \mathcal{F})$ *procedure simply returns* $\{F \setminus S\}$.

Step	S	`SAT`(S)	$\mathcal{H} \leftarrow \mathcal{H} \cup$ `CorrSubsets`$(S, \mathcal{F})$
			$\emptyset$
1.	$\{\neg x_3\}$	*true*	$\{\{c_1, c_2, c_3, c_4, \neg x_1, x_2\}\}$
2.	$\{\neg x_1\}$	*true*	$\{\dots, \{c_1, c_2, c_3, c_4, x_2, \neg x_3\}\}$
3.	$\{x_2\}$	*true*	$\{\dots, \{c_1, c_2, c_3, c_4, \neg x_1, \neg x_3\}\}$
4.	$\{c_1, x_2\}$	*true*	$\{\dots, \{c_2, c_3, c_4, \neg x_1, \neg x_3\}\}$
5.	$\{c_2, x_2\}$	*true*	$\{\dots, \{c_1, c_3, c_4, \neg x_1, \neg x_3\}\}$
6.	$\{c_1, \neg x_3\}$	*true*	$\{\dots, \{c_2, c_3, c_4, \neg x_1, x_2\}\}$
7.	$\{c_2, \neg x_1\}$	*true*	$\{\dots, \{c_1, c_3, c_4, x_2, \neg x_3\}\}$
8.	$\{c_1, \neg x_1\}$	*true*	$\{\dots, \{c_2, c_3, c_4, x_2, \neg x_3\}\}$
9.	$\{c_2, \neg x_3\}$	*true*	$\{\dots, \{c_1, c_3, c_4, \neg x_1, x_2\}\}$
10.	$\{c_4, x_2\}$	*true*	$\{\dots, \{c_1, c_2, c_3, \neg x_1, \neg x_3\}\}$
11.	$\{c_3, x_2\}$	*true*	$\{\dots, \{c_1, c_2, c_4, \neg x_1, \neg x_3\}\}$
12.	$\{\mathbf{c_3}, \neg \mathbf{x_1}\}$	*false*	

Table 4.1: Example 15 - Intermediate steps when computing an `OCUS` for `ExplainOneStep-OCUS` where `CorrSubsets`$(S, \mathcal{F}) = \{\mathcal{F} \setminus S\}$.

Table 4.1 breaks down the intermediate steps of algorithm 4.3 for generating an OCUS of given $\mathcal{F}$, f and p. First, the collection of sets-to-hit His initialized as the empty set. At each iteration, the hitting set solver searches for a cost-minimal assignment that hits all sets in Hand that contains exactly one of $\{\neg x_1, x_2, \neg x_3\}$ (due to p). If the hitting set is unsatisfiable, it is guaranteed to be an OCUS.

The first iterations show that the `OCUS` algorithm first hits all the literals of $\overline{I_{end} \setminus I}$ (steps 1-3 of Table 4.1) and then starts combining one literal of $\overline{I_{end} \setminus I}$ with the remaining clauses until an OCUS is found (step 12 of Table 4.1). Finally, lines Line 5 to Line 8 map the clauses of the unsatisfiable formula back to an explanation step. More precisely, OCUS $\{\mathbf{c_3}, \neg \mathbf{x_1}\}$ signifies that using clause c_3 we can derive x_1, and more formally $c_3 \Rightarrow x_1$.

Incremental MIP Solver The `OCUS` algorithm requires repeatedly computing hitting sets over an increasing collection of sets-to-hit. Initializing the MIP solver once and keeping it warm throughout the `OCUS` iterations allows it to reuse information from previous solver calls to solve the current hitting set problem. Similar to Davies and Bacchus [34], we notice a speed-up between

3 to 5 times by keeping the solver warm[1] compared to initializing a new MIP solver instance at every iteration.

As can be seen in Table 4.1, the `OCUS` algorithm requires a lot of intermediate steps in order to find an OCUS. Notice for instance that the computed correction subsets contain more than 1 literal to explain that are not relevant for explaining the literal in the current hitting set. Taking step 2 as an example, if $\{\neg x_1\}$ is a hitting set, its corresponding correction subset $\{c_1, c_2, c_3, c_4, x_2, \neg x_3\}$ contains $\{x_2, \neg x_3\}$ which cannot be taken.

Even though our running example has a rather small number of clauses, `OCUS` needs to combine an increasingly large number of literals and clauses in order to find an OCUS. Next, we investigate how to grow a given satisfiable subset efficiently in order to reduce the size of the corresponding correction subset. By imposing stronger restrictions on the hitting sets, we will be able to reduce the number of sets-to-hit.

4.2.3 Computing Correction Subsets

The `CorrSubsets` procedure generates a set of correction subsets starting from a given satisfiable subset $\mathcal{S}$ of an unsatisfiable formula $\mathcal{F}$.
However, calling `ExplainOneStep-OCUS` on our running example has shown that a naive 'No grow':

$$\texttt{CorrSubsets}(\mathcal{S}, \mathcal{F}) \leftarrow \{\mathcal{F} \setminus \mathcal{S}\} \tag{4.11}$$

drastically increases the number of sets-to-hit required compared to using the model provided by the `SAT` solver. This last observation suggests that the satisfiable subset S should be *efficiently grown* into a *larger satisfiable subset* before computing the complement similar to the observation in [74]:

$$\texttt{CorrSubsets}(\mathcal{S}, \mathcal{F}) \leftarrow \{\mathcal{F} \setminus \texttt{Grow}(\mathcal{S}, \mathcal{F})\} \tag{4.12}$$

Growing satisfiable subsets using domain-specific information

The goal of the `Grow` phase of the `CorrSubsets` procedure (see Equation (4.12)) is to turn S into a larger satisfiable subformula of $\mathcal{F}$. The effect of this is that the complement added to Hwill be smaller, and hence imposes stronger restrictions on the hitting sets.

There are multiple conflicting criteria that determine what makes an effective 'grow' procedure.

[1]We use the same setup as in the experiment section: a single core on a 10-core INTEL Xeon Gold 61482 (Skylake), a memory-limit of 8GB. The code is written on top of PySAT 0.1.7.dev1 [76], for MIP calls, we used Gurobi 9.1.2, and for the SAT calls MiniSat 2.2.

On the one hand, we want our subformula to be as large as possible (which ultimately would correspond to computing a maximal satisfiable subformula), but on the other hand, we also want the procedure to be very efficient as it is called in every iteration.

For the case of explanations, we make the following observations:

- Our formula at hand (using the notation from the `ExplainOneStep-OCUS` algorithm) consists of three types of clauses:
  1. (translations of) the problem constraints (this is C)
  2. literals representing the assignment found (this is I), and
  3. the negations of literals not-yet-derived (this is $\overline{I_{end} \setminus I}$).
- $\mathcal{C}$ and I together are satisfiable, with assignment I_{end}, and *mutually supportive*, by this we mean that making more clauses in Ctrue, more literals in I will automatically become true and vice versa.
- The constraint p enforces that each hitting set will contain **exactly** one literal of $\overline{I_{end} \setminus I}$

Since the restrictions on the third type of elements of $\mathcal{F}$ are already strong, it makes sense to search for a *maximal* satisfiable subset of $\mathcal{C} \cup I$ with as hard constraints that S should be satisfiable, using a call to an efficient (partial) `MaxSAT` solver.

Furthermore, we can initialize this call as well as any call to a `SAT` solver with the polarities for all variables set to the value they take in I_{end}.

We evaluate different grow strategies as part of the `CorrSubsets`$(S, \mathcal{F})$ procedure in the experiments section, including:

`SAT` extracts a satisfying model from the `SAT` solver to turn S into a larger satisfiable subset. This grow will be considered the baseline for comparing the other grow-variants.

`SubsetMax-SAT` extends the satisfiable subset computed by `SAT` by looping over every remaining clause $c \in \mathcal{F} \setminus S$. If $S \cup \{c\}$ is satisfiable, then the clause c is added to S as well as any other clause from $c' \in \mathcal{F} \setminus S \cup \{c\}$ that is satisfied in the model found by the SAT solver.

`Dom.-spec. MaxSAT` grows satisfiable subset S with a `MaxSAT` solver using only the previously derived facts and the original constraints.

`MaxSAT Full` grows satisfiable subset S with a `MaxSAT` solver using the full unsatisfiable formula $\mathcal{F}$.

In the experiments, we omit the 'No grow' procedure, where the complement $\{\mathcal{F} \setminus S\}$ is returned by `CorrSubsets`. Not growing produces large sets-to-hit as seen in Table 4.1 of the running example. Additional experiments comparing the effectiveness of the naive 'No grow' to growing with the `SAT` solver procedure shows that 'No grow' leads to significantly longer `OCUS` runtimes.

Finally, Table 4.1 showed that `OCUS` has to combine an increasingly large number of literals and clauses. In the next section, we analyse whether we can break the more general OCUS problem into smaller subproblems, similar to Algorithm 3.3, where instead of searching for a MUS, we search for an *Optimal Unsatisfiable Subset* (OUS) and select the best one.

4.3 Multiple Optimal Unsatisfiable Subsets

Preliminary experiments have shown that most of the time ($\sim 90\%$ of the time) is spent searching for hitting sets when generating an explanation step with `OCUS`. The main reason for this is that the hitting set solver needs to consider an increasingly large collection of sets-to-hit, potentially searching over an exponential number of literals and clauses (see Table 4.1 of Example 15).

In this section, we first analyse if instead of working OCUS-based, we can split up the `OCUS`-call into individual calls that compute Optimal Unsatisfiable Subsets (OUSs) for every literal by replacing a MUS call to OUS in Algorithm 3.3. In that case, the original `ExplainOneStep` introduced in Algorithm 3.2 no longer requires greedily growing the subset of constraints to guarantee finding low-cost (optimal) explanations. The resulting simplified algorithm is shown in Algorithm 4.4.

4.3.1 Bounded OCUS

Since OUS, without an additional predicate, is a special case of OCUS, we can take advantage of Proposition 1 and reuse the `OCUS` algorithm with a trivially true p, i.e., $\texttt{OUS}(\mathcal{F}, f) := \texttt{OCUS}(\mathcal{F}, f, \mathbf{t})$ for each $\mathcal{F}$ and f. However, the switch from `MUS` to `OUS` in Algorithm 3.3 still requires looping over every literal and computing the OUS, potentially introducing overhead compared to the single `OCUS` call. However, we can use the OUS obtained in one iteration, to infer a bound on the score that must be achieved in subsequent OUS calls.

Upper Bound Every MUS or OUS computed at Line 4 of Algorithm 3.3 provides an upper bound on the cost, which should be improved in the next iteration. By keeping track of the best candidate explanation, its corresponding cost can be considered the current best upper bound on

Algorithm 4.4: `ExplainOneStep-OUS`(C, f, I, I_{end})

```
1  $\mathcal{N} \leftarrow I_{end} \setminus I$
2  $best \leftarrow (I, C, \mathcal{N})$
3  for $n \in \mathcal{N}$ do
4      $X \leftarrow$ OUS$(C \wedge I \wedge \neg n, f)$
       // Mapping to explanation step
5      $\mathcal{E} \leftarrow I \cap X$
6      $\mathcal{S} \leftarrow \mathcal{C} \cap X$
7      $\mathcal{N} \leftarrow$ propagate$(\mathcal{E} \wedge \mathcal{S})$
8      if $f(\mathcal{E}, \mathcal{S}, \mathcal{N}) < f(best)$ then
9          $best \leftarrow cand$
10     end
11 end
12 return $best$
```

the cost of the OUSs of the remaining literals to explain.

Lower Bound Every hitting set computed inside the `OUS` algorithm produces a lower bound on the best cost that can be obtained, even the satisfying ones. Indeed, the candidate hitting set returned on line 3 of Algorithm 4.3 is guaranteed to be the lowest-cost one. Consequently, the cost of the best candidate explanation so far can be used as an early stopping criterion: if the cost of the current hitting set is larger than the cost of the best explanation so far, `OUS` will not be able to find a better (cheaper) unsatisfiable subset S for that literal.

In fact, such a *bounded OCUS* call is naturally obtained by doing an `OCUS` call with as constraint $p(S) := f(S) \leq f(S_{best})$.

Bounded OCUS-based Explanations `ExplainOneStep-OCUS-Bounded` in Algorithm 4.5 uses calls to the `OCUS` algorithm for every individual literal to compute the next best explanation step. The algorithm keeps track of the current best OCUS candidate S_{best}. This S_{best} is only updated if the `OCUS` algorithm is able to find an OCUS that is cheaper than the current upper bound $f(S_{best})$. The predicate $f(S) \leq f(S_{best})$ of the `OCUS`-call at line 4 of Algorithm 4.5 allows us to ensure that the cost of the hitting set does not exceed the upper bound. In case it does happen, the hitting set solver will return a failure message, meaning that a better candidate explanation cannot be computed for that literal given the current interpretation I.

Algorithm 4.5: `ExplainOneStep-OCUS-Bounded`$(\mathcal{C}, f, I, I_{end})$

```
1  $S_{best} \leftarrow nil$
2  $\mathcal{S}^{\ell}_{best} \leftarrow nil$ for each $\ell$        (or from previous iteration)
3  for $\ell \in \{I_{end} \setminus I\}$ sorted by $f(\mathcal{S}^{\ell}_{best})$ do
4      $\mathcal{S}^{\ell}_{best}, status \leftarrow$ OCUS$((\mathcal{C} \wedge I \wedge \neg l), f, f(S) \leq f(S_{best}))$
5      if $status \neq FAILURE$ then
6          $S_{best} \leftarrow S^{\ell}_{best}$
7      end
8  end
   // Mapping to explanation step
9  $\mathcal{E} \leftarrow I \cap S^{\ell}_{best}$
10 $\mathcal{S} \leftarrow \mathcal{C} \cap S^{\ell}_{best}$
11 $\mathcal{N} \leftarrow$ propagate$(\mathcal{E} \wedge \mathcal{S})$
12 return $(\mathcal{E}, \mathcal{S}, \mathcal{N})$
```

Literal Sorting Obtaining a good upper-bound quickly can further reduce runtime. We can heuristically aid this by keeping track of S^{ℓ}_{best} across explanation steps, and then use its score $f(S^{\ell}_{best})$ to sort the literals at line 3. The literal sorting ensures we first try the cheapest S^{ℓ}_{best} from a previous explanation step since these are more likely to provide a good upper bound on the cost of the next candidate OCUS.

4.3.2 Interleaving OCUS Calls for Different Literals: a Special-case OCUS Algorithm

The case to avoid for `ExplainOneStep-OCUS-Bounded` is that an `OCUS` call for a literal takes many hitting set iterations, and returns an 'expensive' OCUS with lower-cost OCUSs to be found for other literals.

Conceptually, one should only do hitting set iterations for the most promising literal, one that is most likely to produce an OCUS with the lowest cost. Indeed, this is what the original `OCUS` algorithm with an 'exactly one of' constraint is built for: to choose freely among all possible hitting sets across the different literals in order to find the globally optimal next candidate OUS.

For this special case, where the constraint p is that exactly one of a set of literals must be chosen, we can manually decompose the problem to iteratively search for the best hitting set across the independent problems. In such an approach, we do not repeatedly call (bounded) `OCUS` until optimality, but do one hitting-set iteration at a time; each time continuing with one hitting-set iteration of the most promising literal.

This is shown in Algorithm 4.6. Every literal to explain ℓ is associated with:

1. its current *collection of sets to hit* $\mathcal{H}_\ell$; and
2. corresponding *optimal hitting set* S_ℓ (initially the empty set for both) as well as
3. its corresponding *cost* $f(S_\ell)$.

These are stored in a priority queue, sorted by the cost.

Algorithm 4.6: `ExplainOneStep-OCUS-Split`$(\mathcal{C}, f, I, I_{end})$

```
1  $queue \leftarrow$ InitializePriorityQueue($(\ell, \emptyset, \emptyset) : 0 \mid \forall \ell \in I_{end} \setminus I$)
2  while $(\ell, S_\ell, \mathcal{H}_\ell) \leftarrow queue.pop()$ do
3      if $\neg SAT(S_\ell)$ then
           // Mapping to explanation step
4          $\mathcal{E} \leftarrow I \cap S^\ell$
5          $\mathcal{S} \leftarrow \mathcal{C} \cap S^\ell$
6          $\mathcal{N} \leftarrow$ propagate($\mathcal{E} \wedge \mathcal{S}$)
7          return $(\mathcal{E}, \mathcal{S}, \mathcal{N})$
8      $K \leftarrow$ CorrSubsets($\mathcal{S}_\ell, (\mathcal{C} \wedge I \wedge \neg\ell)$)
9      $\mathcal{H}_\ell \leftarrow \mathcal{H}_\ell \cup K$
10     $S_\ell \leftarrow OptHittingSet(\mathcal{H}_\ell, f)$
11     queue.push($(\ell, S_\ell, \mathcal{H}_\ell) : f(S_\ell)$)
12 end
```

`ExplainOneStep-OCUS-Split` repeatedly extracts the best literal-to-explain and corresponding hitting set out of the priority queue. Similar to Algorithm 4.3, if the corresponding hitting set is unsatisfiable, it is guaranteed to be the cost-minimal OUS. This is because the queue ensures that this hitting set is the lowest scoring hitting set across all literals, and because each hitting set is guaranteed to be an optimal hitting set of its collected sets-to-hit $\mathcal{H}_\ell$. If, on the other hand, the hitting set is satisfiable, a number of correction subsets are extracted from the literal-specific unsatisfiable formula and added to its respective collection of sets to hit. Finally, a new hitting set is computed, and this information is pushed back into the priority queue.

The intermediate steps of `OCUS` depicted in Table 4.1 of Example 15 shows that `OCUS` needs to consider *many combinations of clauses and literals* for *all literals to explain*. Whereas, `OCUS_Split` reasons over a smaller unsatisfiable formula containing literals relevant for the literal to explain, and only expands the most promising literal. In the experiments, we compare which

`ExplainOneStep-*` configuration (`OCUS`, `OCUS_Bound`, or `OCUS_Split`) is the fastest for computing explanations.

Example 16 (*continued*). *Consider the previously introduced clauses of our running example $\mathcal{C}$*

$$c_1 := \neg x_1 \vee \neg x_2 \vee x_3 \qquad c_2 := \neg x_1 \vee x_2 \vee x_3 \qquad c_3 := x_1 \qquad c_4 := \neg x_2 \vee \neg x_3$$

In this example, we detail the execution of one call to `ExplainOneStep-OCUS-Bounded` and one call to `ExplainOneStep-OCUS-Split` for $I = \emptyset$ and $I_{end} = \{x_1, \neg x_2, x_3\}$.

The intermediate steps for `ExplainOneStep-OCUS-Bounded` are depicted in Table 4.2. Furthermore, we consider that there does not exist an initial ordering to the literals to explain that can be computed from previous iterations. Table 4.3 depicts the intermediate steps for `ExplainOneStep-OCUS-Split`.

Table 4.2 highlights that executing for a call to `OCUS` for each literal ℓ still requires many iterations even though an upper bound bound on the cost of the best explanation was found after 4 iterations.

The intermediate steps of Table 4.3 highlight the benefit of only executing 1 iteration of `OCUS` for each literal to explain. In fact, the number of iterations, until an explanation, is found is lower compared to both `ExplainOneStep-OCUS-Bounded` (Table 4.2) and `ExplainOneStep-OCUS` (Table 4.1), i.e. 10 instead of 15 and 12 respectively.

In the following section, we analyse how to exploit the fact that OCUS and its variants have to be called repeatedly on an unsatisfiable formula that is incrementally extended when generating a sequence of explanations. We then consider how to reduce the number and size of sets-to-hit, e.g. to include only information relevant to each literal-to-explain, and how to generate small (p-)disjoint correction subsets from a given hitting set.

4.4 Efficiently computing optimal explanations

Up until now, we have investigated how to speed up the generation of an explanation step from the perspective of OCUS as an oracle. In the following, we discuss optimizations applicable to the `O(C)US` algorithms that are specific to explanation sequence generation, though they can also be used when other forms of domain knowledge are present.

ℓ	Step	S	$\texttt{SAT}(S)$	$\mathcal{H} \leftarrow \mathcal{H} \cup \texttt{CorrSubsets}(S, \mathcal{F})$
$\mathbf{x_1}$				$\emptyset$
	1.	$\{\neg x_1\}$	*true*	$\{\{c_1, c_2, c_3, c_4\}\}$
	2.	$\{c_2, \neg x_1\}$	*true*	$\{\dots, \{c_1, c_3, c_4\}\}$
	3.	$\{c_1, \neg x_1\}$	*true*	$\{\dots, \{c_2, c_3, c_4\}\}$
	4.	$\{\mathbf{c_3}, \neg \mathbf{x_1}\}$	*false*	
$\neg\mathbf{x_2}$				$\emptyset$
	5.	$\{x_2\}$	*true*	$\{\{c_1, c_2, c_3, c_4\}\}$
	6.	$\{c_1, x_2\}$	*true*	$\{\dots, \{c_2, c_3, c_4\}\}$
	7.	$\{c_2, x_2\}$	*true*	$\{\dots, \{c_1, c_3, c_4\}\}$
	8.	$\{c_3, x_2\}$	*true*	$\{\dots, \{c_1, c_2, c_4\}\}$
	9.	$\{c_4, \neg x_3\}$	*true*	$\{\dots, \{c_1, c_2, c_3\}\}$
	10.	$\{c_1, c_2, \neg x_2\}$	***FAILURE***	
$\mathbf{x_3}$				$\emptyset$
	11.	$\{\neg x_3\}$	*true*	$\{\{c_1, c_2, c_3, c_4\}\}$
	12.	$\{c_1, \neg x_3\}$	*true*	$\{\dots, \{c_2, c_3, c_4\}\}$
	13.	$\{c_2, \neg x_3\}$	*true*	$\{\dots, \{c_1, c_3, c_4\}\}$
	14.	$\{c_3, \neg x_3\}$	*true*	$\{\dots, \{c_1, c_2, c_4\}\}$
	15.	$\{c_4, \neg x_3\}$	*true*	$\{\dots, \{c_1, c_2, c_3\}\}$
	16.	$\{c_1, c_2, \neg x_3\}$	***FAILURE***	

Table 4.2: Example 16 - Intermediate steps when computing an explanation step with `ExplainOneStep-OCUS-Bounded` where $\texttt{CorrSubsets}(S, \mathcal{F}) = \{\mathcal{F} \setminus S\}$.

4.4.1 Incremental OCUS computation

Inherently, generating a sequence of explanations still requires many O(C)US calls. Indeed, a greedy sequence construction algorithm calls an `ExplainOneStep` variant repeatedly with a growing interpretation I until $I = I_{end}$.

All of these calls to `ExplainOneStep`, and hence O(C)US, are done with very similar input (the set of constraints does not change, and the I slowly grows between two calls). For this reason,

ℓ	Step	S_ℓ	$\texttt{SAT}(S_\ell)$	$\mathcal{H}_\ell \leftarrow \mathcal{H}_\ell \cup \texttt{CorrSubsets}(S_\ell, (C \wedge I \wedge \neg\ell))$
				$\emptyset$
$\mathbf{x_1}$	1.	$\{\neg x_1\}$	*true*	$\{\{c_1, c_2, c_3, c_4\}\}$
$\neg\mathbf{x_2}$	2.	$\{x_2\}$	*true*	$\{\{c_1, c_2, c_3, c_4\}\}$
$\mathbf{x_3}$	3.	$\{\neg x_3\}$	*true*	$\{\{c_1, c_2, c_3, c_4\}\}$
$\mathbf{x_1}$	4.	$\{c_1, \neg x_1\}$	*true*	$\{\ldots, \{c_2, c_3, c_4, \}\}$
$\neg\mathbf{x_2}$	5.	$\{c_1, x_2\}$	*true*	$\{\ldots, \{c_2, c_3, c_4\}\}$
$\mathbf{x_3}$	6.	$\{c_1, \neg x_3\}$	*true*	$\{\ldots, \{c_2, c_3, c_4\}\}$
$\mathbf{x_1}$	7.	$\{c_2, \neg x_1\}$	*true*	$\{\ldots, \{c_1, c_3, c_4\}\}$
$\neg\mathbf{x_2}$	8.	$\{c_2, x_2\}$	*true*	$\{\ldots, \{c_1, c_3, c_4\}\}$
$\mathbf{x_3}$	9.	$\{c_2, \neg x_3\}$	*true*	$\{\ldots, \{c_1, c_3, c_4\}\}$
$\mathbf{x_1}$	10.	$\{\mathbf{c_3}, \neg\mathbf{x_1}\}$	*false*	

Table 4.3: Example 16 - Intermediate steps when computing an explanation step with `ExplainOneStep-OCUS-Split` where $\texttt{CorrSubsets}(S, \mathcal{F}) = \{\mathcal{F} \setminus S\}$.

it makes sense that information computed during one of the earlier stages can be useful in later stages as well. The main question is:

> Suppose two `OCUS` calls are done, first with inputs $\mathcal{F}_1$, f_1, and p_1, and later with $\mathcal{F}_2$, f_2, and p_2; *how can we make use as much as possible of the data computations of the first call to speed up the second call?*

The answer is surprisingly elegant. The most important data `OCUS` keeps track of is the collection Hof correction subsets that need to be hit.

Bootstrapping $\mathcal{H}$ with satisfiable subsets

This collection in itself is not useful for transfer between two calls, since – unless we assume that $\mathcal{F}_2$ is a subset of $\mathcal{F}_1$ – there is no reason to assume that a set in $\mathcal{H}_1$ is also a correction subset of $\mathcal{F}_2$ in the second call. However, each set H in $\mathcal{H}$ is the complement (with respect to the formula at hand) of a *satisfiable subset* of constraints, and each subset of a satisfiable subset is satisfiable as well. Thus, instead of storing $\mathcal{H}$, we can keep track of a set of *satisfiable subsets* **SS**s; as the intermediate results of calls to `CorrSubsets`.

When a second call to `OCUS` is performed, we can then initialize $\mathcal{H}$ as the complement of each of these satisfiable subsets with respect to $\mathcal{F}_2$, i.e.,

$$\mathcal{H} \leftarrow \{\mathcal{F}_2 \setminus S \mid S \in \mathbf{SSs}\}. \tag{4.13}$$

The effect of this is that we *bootstrap* the hitting set solver with an initial set $\mathcal{H}$.

Incrementality with MIP

For hitting set solvers that natively implement incrementality, such as modern Mixed Integer Programming (MIP) solvers, we can generalise this idea further: we know that all calls to $\texttt{OCUS}(\mathcal{F}, f, p)$ will be cast with $\mathcal{F} \subseteq C \cup I_{end} \cup \overline{I_{end} \setminus I_0}$, where I_0 is the start interpretation. To compute the conditional hitting set for a specific $\mathcal{C} \cup I \cup \overline{I_{end} \setminus I} \subseteq C \cup I_{end} \cup \overline{I_{end} \setminus I_0}$, we need to ensure that the hitting set solver only uses literals in $\mathcal{C} \cup I \cup \overline{I_{end} \setminus I}$. For incremental hitting set solvers, this means updating the constraint p at every explanation step to include (1) only literals from interpretation I at the current explanation step, and (2) the 'exactly-one' constraint for explaining one literal at a time.

Since our implementation uses a MIP solver for computing hitting sets (see Proposition 1 of Section 2.3), and we know the entire formula from which elements must be chosen,we initialize the MIP solver once with all relevant decision variables of $C \cup I_{end} \cup \overline{I_{end} \setminus I_0}$.

Bear in mind that retracting a constraint p to replace it with an updated one (in the next explanation call) is non-trivial for MIP solvers. Therefore, we assign an infinite weight in the cost function to all literals of $I_{end} \setminus I$ and update their weights as soon as they have been derived according to the given cost function. In this way, the MIP solver will automatically maintain and reuse previously found sets-to-hit in each of its computations.

Next, we investigate how to speed up the generation of an OCUS using an appropriate `CorrSubsets` method when domain-specific information is available. Through our running example, we will look at the impact of incremetality and a better `CorrSubsets` procedure.

Efficiently generating an explanation sequence with Incremental OCUS

In the following example, we illustrate the efficiency of *incrementality with MIP* together with the *`SAT` grow* to speed up generating an OCUS-based explanation sequence.

Example 17 (*continued*)**.** *Consider the previously introduced clauses of our running example $\mathcal{C}$:*

$$c_1 := \neg x_1 \vee \neg x_2 \vee x_3 \qquad c_2 := \neg x_1 \vee x_2 \vee x_3 \qquad c_3 := x_1 \qquad c_4 := \neg x_2 \vee \neg x_3$$

To define the input for the MIP-incremental `OCUS` *with initial interpretation $I = \emptyset$, we extend $\mathcal{C}$ with the new clauses representing the final interpretation*

$$I_{end} = \{\{x_1, \neg x_2, x_3\}\}$$

and the complement thereof

$$\overline{I_{end} \setminus I} = \{\{\neg x_1, x_2, \neg x_3\}\}$$

For the MIP-incremental variant of `OCUS`*, p remains the same. The cost function f_I is defined as a weighted sum over the following weights:*

Clause weights:	$w_1 = 60$	$w_2 = 60$	$w_3 = 100$	$w_4 = 100$
$I \wedge \overline{I_{end} \setminus I}$ **weights:**	$w_{\neg x_1} = 1$	$w_{x_2} = 1$	$w_{\neg x_3} = 1$	
$I_{end} \setminus I$ **weights:**	$w_{x_1} = \infty$	$w_{\neg x_2} = \infty$	$w_{x_3} = \infty$	

Note how the literals that haven't been derived yet ($I_{end} \setminus I$) are given an infinite weight according to Section 4.4.1 for incrementality purposes. Therefore, f_I will be updated at every explanation step whenever interpretation I changes. The incremental `OCUS`*-call is now:*

$$\texttt{OCUS}(\mathcal{C} \wedge I_{end} \wedge \overline{(I_{end} \setminus I)}, f_I, p) \tag{4.14}$$

In this example, the `Grow` *procedure uses the model provided by the* `SAT` *solver to grow a given subset S. The* `CorrSubsets` *procedure simply returns*

$$K = \texttt{CorrSubsets}(S, \mathcal{F}) = \{\mathcal{F} \setminus \texttt{Grow}(S, \mathcal{F})\} \tag{4.15}$$

The following tables (Tables 4.4 to 4.6) summarise the intermediate steps to compute an OCUS-based explanation sequence for our running example. The literals that cannot be selected by the hitting set solver have been struck out because they have not been derived yet.

Observation 1. ***(An effective grow)*** *The most striking aspect of Table 4.4 compared to Table 4.1 is the number of steps required to find an OCUS that is greatly reduced, i.e. from 12 steps to 6, as a result of choosing an effective grow.*

For the next explanation step, since x_1 has been explained, we adapt the weights of the clauses $\{x_1\}$ and $\{\neg x_1\}$ to $w_{x_1} = 1$ and $w_{\neg x_1} = \infty$ respectively.

Observation 2. ***(Incrementality)*** *In the intermediate* `OCUS` *steps of explanation step 2 (Table 4.5), we observe the effect of incrementality from the number of steps required to find an OCUS. Recall that in the MIP setting, Fis constructed overall literals of I_{end} and $\overline{I_{end} \setminus I}$, and*

Step	S	SAT(S)	*Grow*$(S, \mathcal{F})$	$\mathcal{H} \leftarrow \mathcal{H} \cup K$
				$\emptyset$
1.	$\{\neg x_3\}$	*true*	$\{c_1, c_2, c_4, \neg x_1, \neg x_2, \neg x_3\}$	$\{\{c_3, \cancel{x_1}, x_2, \cancel{x_3}\}\}$
2.	$\{x_2\}$	*true*	$\{c_1, c_2, c_3, x_1, x_2, x_3\}$	$\{..., \{c_4, \neg x_1, \cancel{\neg x_2}, \neg x_3\}\}$
3.	$\{\neg x_1\}$	*true*	$\{c_1, c_2, c_4, \neg x_1, \neg x_2, x_3\}$	$\{..., \{c_3, \cancel{x_1}, x_2, \neg x_3\}\}$
4.	$\{c_4, x_2\}$	*true*	$\{c_1, c_2, c_4, \neg x_1, x_2, \neg x_3\}$	$\{..., \{c_3, \cancel{x_1}, \cancel{\neg x_2}, \cancel{x_3}\}\}$
5.	$\{c_3, \neg x_3\}$	*true*	$\{c_1, c_3, c_4, x_1, \neg x_2, \neg x_3\}$	$\{..., \{c_2, x_2, \cancel{x_3}, \neg x_1\}\}$
6.	$\{c_3, \neg x_1\}$	*false*	-	-

Table 4.4: *Example (continued).* Explanation step 1 ($\mathbf{c_3} \Rightarrow \mathbf{x_1}$) of the explanation sequence generated with `ExplainOneStep-OCUS` with incremental MIP solving (see Section 4.4.1).

hence stays the same throughout all explanation steps. Therefore, the correction subsets of the previous explanation steps can be reused as is. Using the previously computed sets-to-hit ensures that the `OCUS` *algorithm starts from a good candidate OCUS. If the same* `OCUS`*-call is performed without bootstrapping the previous sets-to-hit, the number of intermediate steps is higher, i.e. 5 instead of 2.*

Step	S	SAT(S)	`Grow` $(S, \mathcal{F})$	$\mathcal{H} \leftarrow \mathcal{H} \cup K$
				$\{\{c_3, x_1, x_2, \neg x_3\},$ $\{c_3, x_1, x_2, \cancel{x_3}\},$ $\{c_4, \cancel{\neg x_1}, \cancel{\neg x_2}, \neg x_3\},$ $\{c_3, x_1, \cancel{\neg x_2}, \cancel{x_3}\},$ $\{c_2, x_2, \cancel{x_3}, \cancel{\neg x_1}\}\}$
1.	$\{c_2, x_1, \neg x_3\}$	*true*	$\{c_2, c_3, c_4, x_1, x_2, \neg x_3\}$	$\{..., \{c_1, \cancel{x_3}, \cancel{\neg x_2}\}\}$
2.	$\{c_1, c_2, x_1, \neg x_3\}$	*false*		

Table 4.5: *Example (continued).* Explanation step 2 ($\mathbf{c_1} \wedge \mathbf{c_2} \wedge \mathbf{x_1} \Rightarrow \mathbf{x_3}$) of the explanation sequence generated with `ExplainOneStep-OCUS` *incremental.*

Finally, for the last explanation step, we adapt the weights to reflect the current interpretation and that we only want to explain $\neg x_2$ *(*$w_{x_3} = 1$ *and* $w_{\neg x_3} = \infty$*).*

Observation 3. ***(Disjoint Correction Subsets)*** *Observe set-to-hit* $\{c_4, \cancel{\neg x_1}, \cancel{\neg x_2}, \cancel{\neg x_3}\}$ *in the collection of previously compute sets-to-hit of Table 4.6. Given the current interpretation and the literal* $\neg x_2$ *to explain, only* c_4 *can* and *has to be taken. The phenomenon of a set being disjoint from another with respect to* p *is what we call in Definition 26* p*-disjointness. In this case, subset*

Step	S	$\texttt{SAT}(S)$	$\texttt{Grow}\,(S,\mathcal{F})$	$\mathcal{H} \leftarrow \mathcal{H} \cup K$
				$\{\{c_3, x_1, x_2, \cancel{\neg x_3}\},$ $\{c_3, x_1, x_2, x_3\},$ $\{c_4, \cancel{\neg x_1}, \cancel{\neg x_2}, \cancel{\neg x_3}\},$ $\{c_3, x_1, \cancel{\neg x_2}, x_3\},$ $\{c_2, x_2, x_3, \cancel{\neg x_1}\},$ $\{c_1, x_3, \cancel{\neg x_2}\}\}$
1.	$\{c_4, x_2, x_3\}$	*false*		

Table 4.6: *Example (continued).* Explanation step 3 ($\mathbf{c_4} \wedge \mathbf{x_3} \Rightarrow \neg\mathbf{x_2}$) of the explanation sequence generated with `ExplainOneStep-OCUS` *incremental.*

$\{c_4, \cancel{\neg x_1}, \cancel{\neg x_2}, \cancel{\neg x_3}\}$ is p-disjoint from the other sets-to-hit for the hitting set solver. Therefore, it poses a stronger restriction on the sets-to-hit.

Definition 26. *Two sets S_1 and S_2 are p-disjoint if every set that hits both S_1 and S_2 and satisfies p contains $s_1 \in S_1$ and $s_2 \in S_2$ with $s_1 \neq s_2$.*

Example 17 shows that *incrementality with MIP* and *growing the satisfiable subset S* are effective at reducing the size and the number of sets-to-hit when computing a sequence of explanations using `ExplainOneStep-OCUS`. Next, in section 4.4.1, we take advantage of Definition 26 to enumerate multiple correction subsets that are p-disjoint of each other during the `CorrSubsets` procedure.

Correction subsets enumeration

Our `OCUS` algorithm repeatedly alternates between computing hitting sets and correction subsets. The increasingly large collection of sets-to-hit makes finding optimal hitting sets much more expensive compared to the `CorrSubsets` procedure, which solely relies on finding correction subsets from a given satisfiable subset. Additionally, in the last explanation step of Example 17, we saw that one of the sets-to-hit ($\{c_4, \cancel{\neg x_1}, \cancel{\neg x_2}, \cancel{\neg x_3}\}$) was p-disjoint to the others, imposing a strong restriction on the hitting set solver. The question is:

> Can we cheaply find *multiple*, ideally *p-disjoint*, correction subsets and thereby add multiple sets-to-hit in one go?

Inspired by Marques-Silva et al. [102], we depict a `CorrSubsets` procedure in Algorithm 4.7 that computes a collection of correction subsets K starting from a given subset of constraints S

(either the empty set or a computed hitting set). The `CorrSubsets` procedure will repeatedly compute a satisfiable subset $S' \supseteq S$, and add its complement to the collection of disjoint MCSs K and to S, ensuring that the constraints in the correction subset cannot be present in the next correction subset (disjoint), until no more satisfiable subsets can be found.

Algorithm 4.7: `CorrSubsets`$(\mathcal{S}, \mathcal{F})$

```
1 $K \leftarrow \emptyset$
2 $\mathcal{S}' \leftarrow \mathcal{S}$
3 while $\texttt{SAT}(S')$ do
4     $\mathrm{C} \leftarrow \mathcal{F} \setminus \texttt{Grow}(S', \mathcal{F})$
5     $S' \leftarrow S' \cup \mathrm{C}$
6     $K \leftarrow K \cup \{\mathrm{C}\}$
7 end
8 return $K$
```

For simple constraints p, Algorithm 4.7 is directly applicable and will be able to compute *disjoint* correction subsets. However, for more complicated p constraints, this easily degrades into computing only a single correction subset.

Correction subset enumeration with incremental MIP

If Algorithm 4.7 is directly applied to the MIP incremental `OCUS` variant that considers the whole formula $\mathcal{C} \wedge I_{end} \wedge \overline{(I_{end} \setminus I)}$, both the literal-to-explain and its negation will be present at line 5 leading to UNSAT. In a given explanation step, only the base constraints $\mathcal{C}$, the current interpretation I, and the literals of $\overline{(I_{end} \setminus I)}$ can be hit by the hitting set solver. Therefore, we project subset S' onto the base constraints, the current interpretation, and the negated literals to explain. This extra step is executed right after line 5 of Algorithm 4.7:

$$S' \leftarrow S' \cap (\mathcal{C} \cup I \cup \overline{I_{end} \setminus I})$$

By incorporating *explanation-specific* information, we are able to enumerate extra correction subsets that are p-disjoint.

Example 18 (Example 17 continued)**.** *Consider the same setting as in Example 17 with given initial interpretation $I = \emptyset$ and $I_{end} = \{x_1, \neg x_2, x_3\}$. Table 4.7 illustrates the efficiency of the incremental variant of the `CorrSubsets` algorithm starting from hitting set $S := \{\neg x_3\}$ as in the first step of our running example (Table 4.4 of Example 17). This example uses the `SAT`-based `Grow` of Section 4.2.3.*

Step	S'	$\texttt{SAT}(S')$	$S' \leftarrow \texttt{Grow}(S, \mathcal{F})$	$K \leftarrow K \cup \{\mathcal{F} \setminus S'\}$
				$\emptyset$
1.	$\{\neg x_3\}$	*true*	$\{c_1, c_2, c_4, \neg x_1, \neg x_2, \neg x_3\}$	$\{\{c_3, \cancel{x_1}, x_2, \cancel{x_3}\}\}$
2.	$\{c_3, x_2, \neg x_3\}$	*true*	$\{c_2, c_3, c_4, x_1, x_2, \neg x_3\}$	$\{..., \{c_1, \neg x_1, \cancel{\neg x_2}, \cancel{x_3}\}\}$
3.	$\{c_1, c_3, \neg x_1, x_2, \neg x_3\}$	*false*		

Table 4.7: *Example 17 (continued).* Correction Subset enumeration starting from hitting set $\{\neg x_3\}$ with *MIP-incremental* OCUS.

The MIP-based incremental variant of OCUS uses unsatisfiable formula Fdefined as $\mathcal{C} \wedge I_{end} \wedge \overline{(I_{end} \setminus I)}$. For a given interpretation $I = \emptyset$, the $\mathcal{F}$ corresponds $\mathcal{F} := \mathcal{C} \wedge \{x_1\} \wedge \{\neg x_2\} \wedge \{x_3\} \wedge \{\neg x_1\} \wedge \{x_2\} \wedge \{\neg x_3\}$.

*Having a closer look at step 1 of Table 4.7, $\neg x_3$ is present in S and x_3 is in the corresponding correction subset. If we were to add it directly to S, the algorithm would stop at the next iteration, since S would be unsatisfiable. However, x_3 (and x_1) cannot be selected by the hitting set solver, and therefore should not be added. By adding only the clauses from $\{\mathcal{F} \setminus S\}$ projected onto $(\mathcal{C} \cup I \cup \overline{I_{end} \setminus I})$, i.e. c_3 and x_2, we are able to enumerate **multiple** correction subsets that are p-disjoint for the hitting set solver.*

Example 18 depicts the effectiveness of enumerating multiple correction subsets that are p-disjoint by projecting the correction subset onto the unsatisfiable formula for the current interpretation $(\mathcal{C} \cup I \cup \overline{I_{end} \setminus I})$. In the rest of the chapter, we consider correction subset enumeration with a SAT-based grow as the baseline approach. We refer to it as *Multi-SAT*.

4.5 Experiments

In this section, we validate the *qualitative improvement* of computing explanations that are optimal with respect to a cost function, as well as the *performance improvement* of the different versions of our algorithms, in comparison with the explanation generation algorithms presented in the previous chapter.

Experimental Setup

Our experiments[1] were run on a compute cluster where each explanation sequence was assigned a single core on a 10-core INTEL Xeon Gold 61482 (Skylake) processor with a time limit of 60 minutes and a memory-limit of 8 GB. The code is written on top of PySAT 0.1.7.dev1 [76]. For the MIP calls, we used Gurobi 9.1.2, for SAT calls MiniSat 2.2 and for `MaxSAT` calls RC2 as bundled with PySAT. In the MUS-based approach, we used PySAT's deletion-based MUS extractor MUSX [100].

Benchmark dataset

Our benchmark dataset consists of generated Sudoku puzzles of increasing difficulty (different amount of given numbers), the Logic Grid puzzles of Chapter 3, as well as the instances from Espasa et al. [44].[2] In Table 4.8 of Section 4.5, we summarise the average number of clauses (avg. # clauses), the average number of literals to explain (avg. # lits-to-explain) as well as the number of puzzles (n) in the benchmark data set for each puzzle family.

When generating an explanation sequence for these puzzles, the unsatisfiable subsets identify which constraints and which previously derived facts should be combined to derive new information. For Logic Grid puzzles, we assign a cost of 60 for puzzle-agnostic constraints (bijectivity and transitivity constraints); 100 for puzzle-specific constraints (clues); and a cost of 1 for facts. For all other puzzles, we assign a cost of 60 when using a constraint and a cost of 1 for facts. In Table 4.8 of Section 4.5, we summarise the average number of clauses (avg. # clauses), the average number of literals to explain (avg. # lits-to-explain) as well as the number of puzzles (n) in the benchmark data set for each puzzle family.

Research Questions Our experiments are designed to answer the following research questions:

Q1 What is the effect of using an *optimal* unsatisfiable subset, on the quality of the generated step-wise explanations?

Q2 What is the impact of information **re-use** on the efficiency of `OCUS`?

Q3 What are the time-critical components of `OCUS`?

[1]The code for all experiments is made available at `https://github.com/ML-KULeuven/ocus-explain`.

[2]We express our gratitude towards *Matthew. J. McIlree* and *Christopher Jefferson* of St. Andrews University for their help with the extraction of CNF instances from Essence problem specifications [53] using Savile Row [110], as well as for supplying the problem instances.

			Properties of instances	
puzzle type	**instance**	n	avg. # clauses	avg. # lits-to-explain
logic	pasta	1	4572	96
	150	8	7535	150
	250	1	17577	250
sudoku	9x9-easy	25	14904	497
	9x9-simple	25	14904	497
	9x9-intermediate	25	14904	501
	9x9-expert	25	14904	505
demystify	binairo	156	16759	99
	garam	10	14486	756
	kakurasu	1	226	16
	kakuro	6	6961	240
	killersudoku	13	3609685	729
	miracle	1	59331	729
	nonogram	1	9813	36
	skyscrapers	16	17991	159
	star-battle	5	7533	82
	sudoku	76	34143	729
	tents	4	18644	476
	thermometer	1	1227	36
	x-sums	1	109717	729

Table 4.8: Characteristics of benchmark instances

Q4 How does more advanced extraction of correction subsets and extraction of multiple correction subsets affect performance?

Q5 What is the efficiency of a single step O(C)US

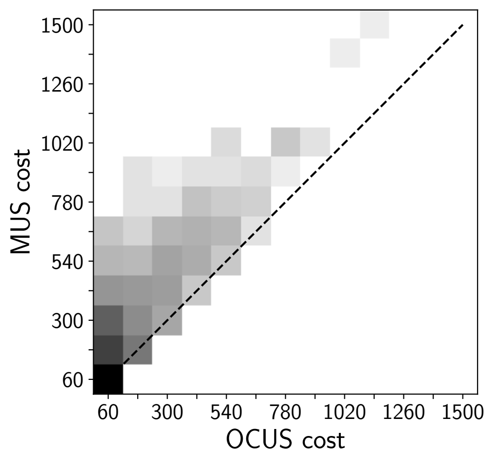
(a) from an instantaneous (time-to-first) explanation point of view?

(b) from a step-wise (single next explanation) solving point of view?

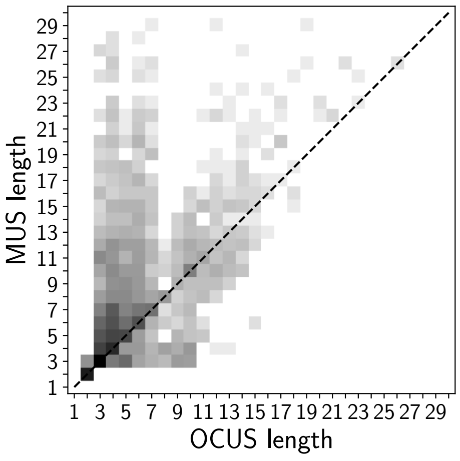
4.5.1 Explanation Quality

To evaluate the effect of optimality on the quality of the generated explanations, we reimplemented a MUS-based explanation generator based on Algorithm 3.3. Before presenting the results, we want to stress that this is *not* a fair comparison with the implementations of Chapter 3 and Espasa et al. [44], where a heuristic is used that relies on *even more* calls to `MUS` in order to avoid the quality problems we will illustrate below. While in both cases this would yield better explanations, it comes at the expense of computation time, thereby leading to several hours to generate the explanation of a single puzzle.

To answer **Q1**, we ran the OCUS-based algorithm as described in Algorithm 4.1 and compared at every step the cost (size) of the produced explanation with the cost (size) of the MUS-based explanation of Algorithm 3.3. These costs are plotted on a heatmap in Figure 4.1a, where the darkness represents the number of occurrences of the combination at hand. Figure 4.1b depicts the length of the MUS and OCUS for the same explanation pairs.



(a) Explanation cost comparison



(b) Explanation size comparison

Figure 4.1: Q1 - Explanation quality comparison of optimal versus subset-minimal explanations in the generated puzzle explanation sequences.

We see that the difference in quality is striking in many cases, with the MUS-based solution often missing very cheap explanations (as seen by the darkest squares in the column above cost 60), thereby confirming the need for a cost-based OUS/OCUS approach. The same observation can be made about the lower triangle of the diagram in Figure 4.1b where the size of the MUS-based explanation is smaller than the OCUS one. The MUS-based explanations use *more constraints and fewer facts* and therefore have a higher cost, compared to the OCUS-based explanations which use *more facts and fewer constraints* to derive new information and therefore have a lower cost.

4.5.2 Information Re-use

To answer **Q2**, we compare the effect of incrementality when generating a sequence of explanations. Next to

1. OCUS, we also include
2. the *bounded* OCUS (OCUS_Bound) algorithm, where we call the OCUS algorithm for every literal in every step, but we reuse information by giving it the current best bound $f(S_{best})$ and iterating over the literals that performed best in the previous call first; and
3. the *split* OCUS (OCUS_Split) approach, where we split up the computations by iteratively selecting the most promising literal and expanding only one hitting set for it, then re-evaluating the most promising literal and so forth.

Incremental versus non-incremental explanation generation

Preliminary experiments have shown that incrementality by keeping track of the hitting sets using the MIP solver significantly outperforms bootstrapping satisfiable subsets. Therefore, in the experiments, we only show results on incrementality using MIP solvers.

For OCUS, incrementality (*+Incr*) is achieved by reusing the same MIP hitting set solver throughout the explanation calls, as explained in Section 4.4.1. Methods OCUS_Bound and OCUS_Split use a separate hitting set solver for every literal to explain. Each hitting set solver can similarly be made incremental (with respect to its own literal) across the explanation calls (*+Incr.*), or not.

Figure 4.2 depicts the number of instances fully explained through time before the given 1-hour timeout. In each configuration depicted, we use a single SAT-based Grow in the CorrSubsets procedure. The same colour is used for the same OCUS algorithms, with a full line for the incremental version and a dashed line for the non-incremental one.

If we compare all configurations, we see that OCUS is faster than the plain MUS-based implementation for simpler instances. MUS is able to explain more instances than all OCUS versions, as it solves a simpler problem (with worse quality results, as shown in **Q1**). OCUS_Bound is faster than OCUS for easier instances but explains about the same number of instances, and OCUS_Split is considerably faster and solves the most instances out of all OCUS algorithms.

The effect of introducing incrementality produces a speed-up in OCUS. For OCUS_Bound it is negligible, and for OCUS_Split there is a speed-up in all but its most difficult instances. In general, we see that the curves of the incremental variants are located somewhat lower. The best

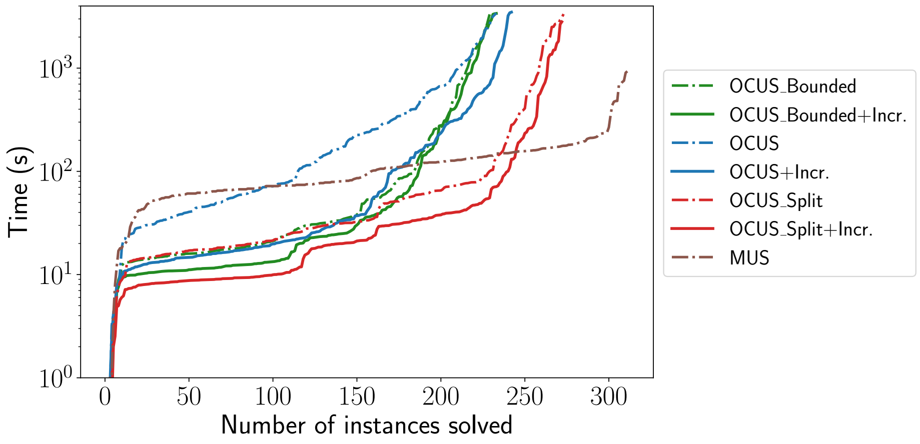


Figure 4.2: Time to generate a full explanation sequence with a single `SAT` grow call in the `CorrSubsets` procedure for incremental and non-incremental `OCUS` algorithms. Incrementality improves the number of instances solved and the time required to solve them for all O(C)US algorithms.

runtimes are obtained with `OCUS_Split+Incr.`, that is, using an incremental hitting set solver *for every individual literal-to-explain separately* and only expanding the literal that is most likely to provide a cost-minimal OCUS. Interestingly, the explanations for the instances solved by `MUS` but not by the other algorithms (between `OCUS_Split+Incr.` and `MUS`) have a higher average cost as well as a higher maximum cost. This means that the instances are more difficult to explain.

Instance-level speed-up with incrementality

Next, we analyse the speed-up of introducing incrementality per instance. Figure 4.3 compares the time to generate a full sequence for the incremental variant with the non-incremental version for each puzzle of every `OCUS` configuration. Results in the upper triangle signify an improvement using incrementality, and the lower triangle indicates worse performance with incrementality.

Having a closer look at the results for `OCUS`, the story is similar to the results of Figure 4.2. Many of the instances are explained quickly (around 10 to 20 seconds) by `OCUS+Incr.`, whereas non-incremental `OCUS` takes much longer to explain some instances, even leading to a timeout.

Incrementality with `OCUS_Split` and `OCUS_Bound` produce marginal improvements on the runtime. Most of the instances lie close to the black line. Only a few instances are present in the lower triangle, for the incremental variant of `OCUS_Bound`. For these instances, `OCUS_Bound+Incr.`

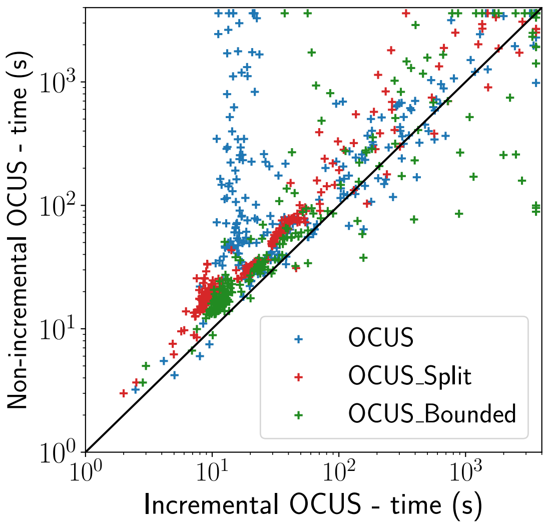


Figure 4.3: Q2 - Runtime comparison of introducing incrementality on the time to generate a whole explanation sequence with `SAT`-based `Grow`.

takes on average double the time to generate the first explanation step due to the overhead associated with introducing incrementality. The story is similar for the few instances where `OCUS_Split+Incr.` is slower than `OCUS_Split`.

4.5.3 Runtime Decomposition

To evaluate the time-critical components of `OCUS` (Q3), we decompose the time spent in each part of the `OCUS` algorithms in Table 4.9. We only included the runtimes of instances that do not time out for all methods to allow a fair comparison.

The table depicts for each configuration

1. the number of instances *explained*;
2. %`OPT` the percentage of time spent in the hitting set solver;
3. %`SAT` the percentage of time spent in the SAT solver:
4. %`CorrSS` the percentage of time spent in the `CorrSubsets` procedure; and

5. N_{sth} the average total number of sets-to-hit computed.

Each version of the OCUS algorithm uses the SAT-based grow in the CorrSubsets procedure (see Table 4.10).

Looking closer at the runtime decomposition, we observe that most of the time in all the OCUS algorithms is spent computing the optimal hitting sets. The results in the N_{sth} column (top 3 rows versus incremental bottom 3 rows), highlight the decrease in the amount of sets-to-hit that need to be computed when adding incrementality, in our case, re-using the MIP solver between explanation steps. Indeed, similar to the results of Example 17, the OCUS algorithms do not need to recompute the previously derived sets-to-hit, and will therefore start with a good candidate hitting set at the beginning of an explanation step.

config	explained	%OPT	%SAT	%CorrSS	N_{sth}
OCUS	[233 / 403]	96.65%	3.01%	0.35%	6245
OCUS_Bound	[229 / 403]	86.86%	12.31%	0.83%	9310
OCUS_Split	[273 / 403]	88.58%	10.41%	1.01%	7480
OCUS+Incr.	[242 / 403]	99.16%	0.78%	0.06%	405
OCUS_Bound+Incr.	[233 / 403]	95.95%	3.94%	0.1%	1538
OCUS_Split+Incr.	[272 / 403]	96.28%	3.42%	0.3%	1713
MUS	[311 / 403]	—	—	—	—

Table 4.9: Runtime decomposition of core parts of O(C)US with the SAT-based grow. On the left, most of the computational time is spent in computing more optimal (constrained) hitting sets, while on the right, the runtime is shifted towards higher quality correction subsets. Incrementality, by re-using the same MIP solver throughout the explanation steps, helps to drastically reduce the average number of sets-to-hit N_{sth}.

4.5.4 Correction subset extraction

As Table 4.9 shows, most time is spent computing the optimal hitting sets. Hence, there is potential in reducing its effort by computing better or more correction subsets, thereby having better collections of sets-to-hit. This induces a trade-off between the *efficiency* of the Grow strategy, the *quality* of the produced *satisfiable subset*, as well as the corresponding *sets-to-hit* generated by the CorrSubsets procedure. In this section, we answer **Q4** by determining whether an efficient Grow-call is able to balance the efficiency and the quality of the produced satisfiable subset. Second, we analyse whether the enumeration of multiple, ideally p-disjoint, correction subsets reduces the time spent in the hitting set solver. Finally, note that the average number of sets-to-hit is significantly lower for OCUS+Incr. than for OCUS_Split+Incr. and OCUS_Bound+Incr., due to the fact that OCUS+Incr. solves a single hitting set problem, whereas OCUS_Split+Incr. and OCUS_Bound+Incr. have to solve for each literal to explain.

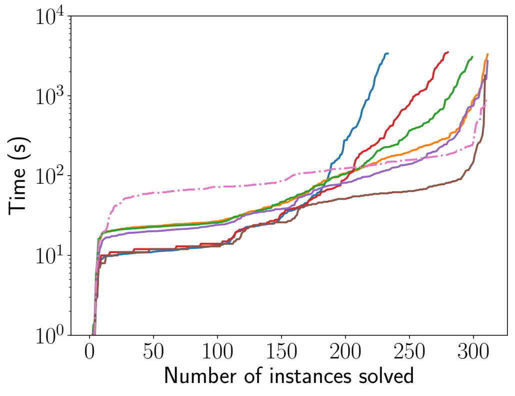
Thus, to answer **Q4**, we compare the incremental variants of OCUS, *bounded* OCUS, *split* OCUS, and only change the CorrSubsets strategy they use.

Efficient Correction Subset Enumeration Extending the observation of Section 4.2.3 that a call to an efficient MaxSAT solver helps in finding maximally satisfiable subsets, we propose three additional variants of Algorithm 4.7:

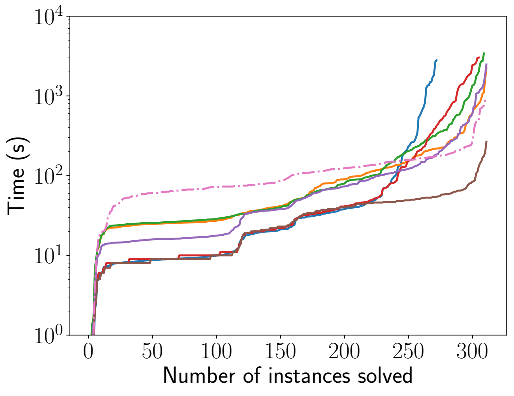
1. The first variant, *Multi-SAT*, repeatedly grows with SAT and blocks the corresponding correction subset until no more correction subsets can be extracted.
2. The second variant is *Multi-MaxSAT*. This correction subset enumerator repeatedly grows using the domain-specific MaxSAT introduced in Section 4.2.3 until the updated subset S' is no longer satisfiable.
3. The last variant *Multi-SubsetMax-SAT* repeatedly grows using the *SubsetMax-SAT* to balance the trade-off between efficiency and quality.

CorrSubsets	**Description**
SAT	*Extract a satisfiable model computed by the SAT solver.*
SubsetMax-SAT	*Extends the satisfiable subset computed by SAT by adding every remaining clause $c \in \mathcal{F} \setminus S$ if $S \cup \{c\}$ is satisfiable.*
Dom.-spec. MaxSAT	*Grow using an unweighted MaxSAT solver with domain-specific information, i.e. only previously derived facts and the original constraints (see Section 4.4.1).*
MaxSAT Full	*Grow with an unweighted MaxSAT solver using the full unsatisfiable formula $\mathcal{F}$.*
Multi SAT	*Repeatedly grow with* **SAT** *and block the corresponding correction subset until no more correction subsets can be extracted.*
Multi Dom.-spec. MaxSAT	*Repeatedly with* **Dom.-spec. MaxSAT** *and block the corresponding correction subset until no more correction subsets can be extracted.*
Multi SubsetMax-SAT	*Repeatedly grow with* **SubsetMax-SAT** *and block the corresponding correction subset until no more correction subsets can be extracted.*

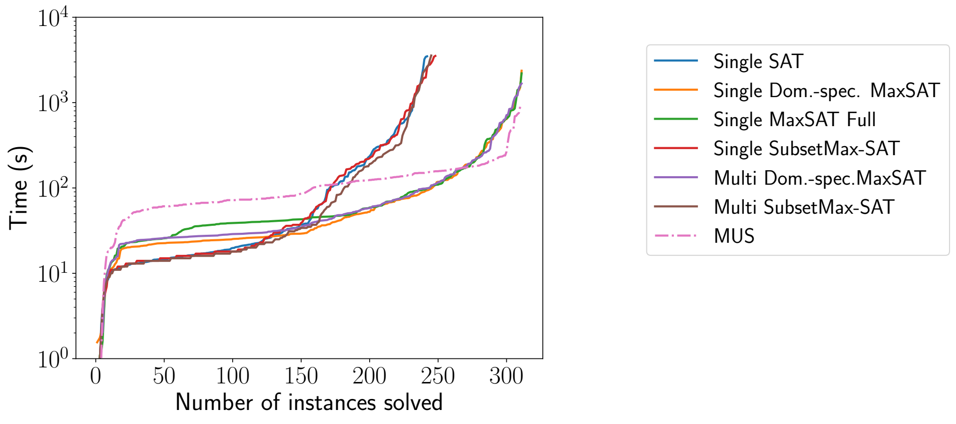
Table 4.10: Description of the correction subset procedures.



(a) `OCUS_Bound+Incr.`



(b) `OCUS_Split+Incr.`



(c) `OCUS+Incr.`

Figure 4.4: Average cumulative explanation time of correction subset configurations for the incremental O(C)US variants. The legends for each graph are ordered from the least instances explained (top) to the most instances explained (bottom).

Figure 4.4 depicts for each incremental `OCUS` algorithm, the number of instances solved across time for varying correction subset approaches. For `OCUS+Incr.`, all `MaxSAT`-based methods for enumerating correction subsets provide similar results and are able to explain more instances than the rest. When examining the instances that did not timeout in any correction subset procedure, the results for `OCUS+Incr.` show that 60% more sets-to-hit need to be extracted for all non `MaxSAT`-based methods compared to around 40 % for the `MaxSAT`-based grows. The consequence is that more time is spent in solving the hitting set problem: close to 90 % for the non MaxSATbased algorithms compared to around 40 % for the MaxSATones. The story for OCUS_Split+Incr. and OCUS_Bound+Incr. is different. The best method for enumerating correction subsets with OCUS_Split+Incr. and OCUS_Bound+Incr. is *Multi*-SubsetMax-SAT. *Multi*-SubsetMax-SAT is able to consistently beat all other correction subset enumeration methods and is substantially faster than the naive MUS approach.

To summarise, the best explanation sequence configuration corresponds to the iterated MIP-incremental approach of OCUS_Split+Incr. combined with *Multi*-SubsetMax-SAT.

Runtime Decomposition We now look at the runtime decomposition in Table 4.11 for the best performing CorrSubsets configuration, i.e. *Multi*-SubsetMax-SAT, also for the non-incremental OCUS variants. For the non-incremental variants, we observe that the shift in computation from the hitting set solver to correction subset enumeration clearly reaps its benefits.

	***Multi*-SubsetMax-SAT**				
config	**explained**	**%OPT**	**%SAT**	**%CorrSS**	$\mathbf{N_{sth}}$
OCUS	[291 / 403]	68.15%	2.78%	29.07%	3649
OCUS_Bound	[311 / 403]	39.87%	6.53%	53.6%	29747
OCUS_Split	[311 / 403]	45.24%	3.62%	51.14%	28320
OCUS+Incr.	[245 / 403]	87.38%	1.03%	11.59%	402
OCUS_Bound+Incr.	[309 / 403]	81.57%	5.68%	12.76%	1251
OCUS_Split+Incr.	[311 / 403]	79.64%	1.77%	18.58%	1274
MUS	[311 / 403]	—	—	—	—

Table 4.11: Runtime decomposition of core parts of OCUS. On the left, most of the computation time is spent in computing more optimal (constrained) hitting sets, while on the right, the runtime is shifted toward higher-quality correction subsets.

Compared to Table 4.9, more instances are explained, and the computation is more balanced both for incremental and non-incremental OCUS algorithms. The non-incremental OCUS variants require considerably more correction subsets to be extracted, and somewhat fewer in the incremental case. However, the number of instances explained is substantially higher for all OCUS configurations.

4.5.5 Efficiency of Single step O(C)US, Instantaneous and Step-wise

In this section, we analyse whether the algorithms we developed could be fit for an interactive context. By an interactive context, we consider two types of settings. First, a user may want an *immediate explanation step* for the given CSP, and second, a user is asking for the next step, and the next, and so forth.

Table 4.12 depicts *Multi*-SubsetMax-SAT the best performing CorrSubsets procedure of Figure 4.4. For each OCUS algorithm, we compute the average time to produce the first explanation step $\overline{t_1}$ (instantaneous), the average time to generate one step-wise explanation $\overline{t_{step}}$, and the explanation time quantiles 25 up to 100, where quantile 100 q_{100} symbols the most expensive step to generate over all the problem instances.

	***Multi*-SubsetMax-SAT**							
config	$\overline{\mathbf{t_1}}$	$\overline{\mathbf{t_{step}}}$	$\mathbf{q_{25}}$	$\mathbf{q_{50}}$	$\mathbf{q_{75}}$	$\mathbf{q_{95}}$	$\mathbf{q_{98}}$	$\mathbf{q_{100}}$
OCUS	3.41	2.54	0.51	0.77	1.39	8.37	18.50	523.67
OCUS_Bound	1.70	1.35	0.58	0.92	1.88	3.67	4.45	7.34
OCUS_Split	0.92	1.31	0.54	0.90	1.84	3.58	4.24	6.11
OCUS+Incr.	4.33	3.66	0.27	0.42	0.72	8.69	21.46	3452.20
OCUS_Bound+Incr.	3.22	0.58	0.39	0.51	0.67	1.07	1.57	17.58
OCUS_Split+Incr.	2.58	0.45	0.32	0.43	0.54	0.70	1.04	18.05

Table 4.12: **Non-timed-out instances:** Decomposition of runtime for individual explanations into time to generate the first explanation step (t_1), the average time to produce an additional explanation step $\overline{t_{expl}}$ and q_{xx} the quantiles of the explanation times.

For the non-incremental variants, in the upper part of Table 4.12, OCUS is able to compute part of the explanations faster than the other OCUS configurations. We see, however, that both incremental and non-incremental OCUS still take a lot of time for some explanations compared to the other OCUS-based algorithms.

In the case of *OCUS_Bound*, the time to the first explanation reflects how important the ordering of literals to explain is for quickly finding a good bound on the cost of the next best explanation.

The next step is to find out what causes instances to timeout before the full explanation sequence has been generated with *Multi*-SubsetMax-SAT for all OCUS configurations.

Timed out Instances Most of the instances that timed out had on average more literals to explain than those that did not, more precisely all *killersudoku* instances, the *miracle* problem, all *sudoku* problem instances of the demistify dataset, one of the *skyscrapers* instances, and the *x_sums* instance. Table 4.13 provides more context about the timed-out instances:

- *#timed-out* is the number of instances that have timed out.
- *#none* is the number of instances where no explanation step was generated.

config	#none	#timed-out	$\overline{\%\text{ explained}}$				
			killersudoku	miracle	skyscrapers	sudoku	x-sums
`MUS`	13	92	—	0.68%	15.09%	8.86%	—
`OCUS`	1	112	1.19%	0.96%	23.90%	5.96%	3.43%
`OCUS_Bound`	43	92	—	—	40.25%	2.20%	—
`OCUS_Split`	16	92	—	—	49%	7.79%	—
`OCUS+Incr.`	2	158	1.06%	0.82%	28.30%	21.96%	0.41%
`OCUS_Bound+Incr.`	43	94	—	—	55.34%	7.29%	—
`OCUS_Split+Incr.`	15	92	—	—	43.40%	15.75%	—

Table 4.13: **Timed-out instances:** *#none* is the number of instances where no explanations step was found before the time out, *#timed-out* is the number of instances that timed out, $\overline{\%\text{ expl}}$ is the average percentage of literals explained.

- $\overline{\%\text{ explained}}$ is the average percentage of literals that were explained before the instance timed out for each instance family.

Even though `OCUS` and `OCUS+Incr.` has the most time out, it is able to at least find an explanation for most of the difficult instances where no other configuration finds any.

In Figure 4.5 we specifically report the average time taken to generate a first explaining step ($\overline{\mathtt{t_1}}$) for all configurations.

Note from Table 4.13 that `OCUS`(+Incr.) has the smallest number of instances where no explanation was found. Unlike `OCUS_Bound`(+Incr.) and `OCUS_Split`(+Incr.), `OCUS`(+Incr.) does not need to iterate over the many literals to explain in order to find a good bound on the cost of the next best explanation step. Furthermore, we observe that `OCUS`(+Incr.) is the fastest at generating a first explanation step, with many outliers close to the timeout.

Similar to the observations in Section 4.5.2, incrementality has a high impact on the time to explain an instance for all `OCUS` configurations. Adding incrementality to all `OCUS` configurations results in more instances with no explanation found than without incrementality. However, incrementality helps to explain more literals, except for `OCUS_Bound`, which first requires computing many OUSs before the best one is found. A similar trend can be seen in Figure 4.5, where the introduction of incrementality increases the time to generate a first explanation step. Second, `OCUS_Bound` depends on a good ordering of the literals, which can change at any explanation step in the sequence.

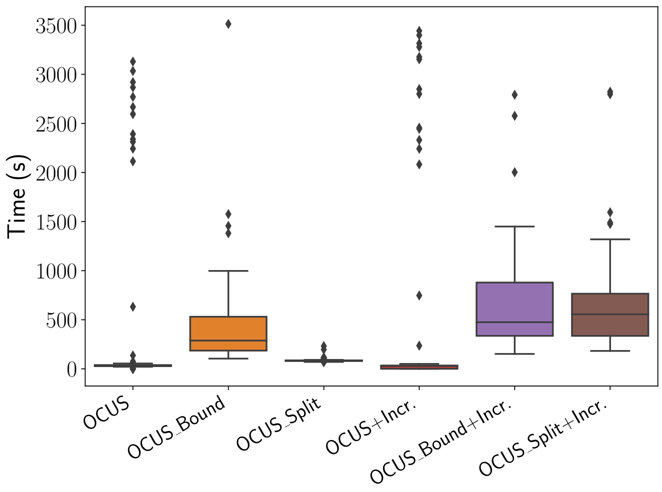


Figure 4.5: Average time to a first explanation step $\overline{\mathtt{t_1}}$.

To conclude, for an interactive context where the user asks for *an immediate explanation*, `OCUS_Split` will be the fastest to compute an explanation step. If the user *explores the sequence of explanations* and repeatedly *asks for additional explanations*, `OCUS_Split+Incr.` will be able to capitalise on incrementality to bring the average time close to real time.

4.6 Related Works

In Chapter 3, we have introduced *step-wise explanations* to explain the solution of satisfiable instances, exploiting a one-to-one correspondence between so-called non-redundant explanations and MUSs of a derived program. The focus of that chapter was on explaining Zebra puzzles; building on this work, Espasa et al. [44] investigated interpretable explanations using MUSs for a wide range of puzzles. Our current work is motivated by a concrete algorithmic need: to generate these explanations *efficiently*, we need algorithms that can find MUSs that are optimal with respect to a given cost function, where the cost function approximates human understandability of the corresponding explanation step.

The closest related work can be found in the literature on generating or enumerating MUSs [98, 7, 6, 95]. Various techniques are used to find MUSs, including manipulation of resolution proofs produced by SAT solvers [61, 59, 39], incremental solving to enable/disable clauses and branch-and-bound search [113], BDD-manipulation methods [71]. Some methods rely on *seed-shrink* algorithms [10, 11] which repeatedly start from unsatisfiable *seed* (an unsatisfiable core) and *shrink* the seed to a MUS. Other methods work by means of translation into a so-called Quantified MaxSAT [75], a field that combines the expressiveness of Quantified Boolean Formulas (QBF) [83] with techniques from Maximum Satisfiability (MaxSAT) [91], or by exploiting the so-called hitting set duality [74]. To the best of our knowledge, only few works have considered *optimizing* MUSs: the only criterion considered so far is cardinality-minimality [98, 74]. Extending some of the above mentioned to handle can be straightforward for cost functions that can be encoded as a weighted sum, for instance, by ordering (subsets of) constraints based on the weight of each constraint similar to Algorithm 3.2. However, modifying the above algorithms to handle more complex, e.g. non-linear, cost functions is non trivial.

An *abstract* framework for describing hitting set–based algorithms, including optimization, has been developed by Saikko et al. [122]. While our approach can be seen to fit within the framework, the terminology is focused on `MaxSAT` rather than MUS and would complicate our exposition. For instance, hitting set–based algortihms for `MaxSAT`work by extracting *unsatisfiable cores* until a `MaxSAT`solution is found. In contrast, our approach is based on extracting correction subsets.

4.7 Concluding Remarks

In this chapter, we tackled the problem of efficiently generating explanations that are optimal with respect to a cost function. For that, we introduced algorithms that replace the many calls to an unsatisfiable core extractor with a single call for an optimal constrained unsatisfiable subset (`OCUS`). Our algorithm for computing `OCUS`s uses the implicit hitting set duality between Minimum Correction Subsets (MCS) and Minimal Unsatisfiable Subsets (MUSs). We propose two additional variants for the 'exactly one of' constraint used in step-wise explanation generation. The main bottleneck of these approaches is having to repeatedly compute *optimal* hitting sets. To compensate, we have developed methods for enumerating correction subsets that explore the trade-off between efficiency and quality of the subsets generated. An open question in an interactive setting is whether there are other methods of enumerating correction subsets that are better suited to reducing the time to first explanation and the average explanation time.

To efficiently generate the *whole* explanation sequence, we introduced *incrementality*, which allows the reuse of computed information, i.e. satisfiable subsets remain valid from one explanation call to another. In the case of `OCUS` for explanation sequence generation, where the underlying hitting set solver is MIP-based, we instantiate the MIP solver once for all explanation steps and keep track of computed sets-to-hit. However, MIP solvers do not natively support non-linear cost functions. The current implementation of `OCUS` may therefore not be able to fully capture the complexity of what constitutes a good explanation due to the limitations of the cost functions we can encode. In the next chapter, we investigate machine learning techniques to characterise what understandable explanations directly from users, and their integration with our current explanation methods to generate user-preferred explanations.

Finally, for interactive settings such as interactive tutoring systems, we are able to generate explanations in near real time (less than a second) using `OCUS_Split`. Another question is how to distribute the computation across multiple cores to parallelize the computation to further speed up the generation of explanations.

CHAPTER 5

Learning Preferences over Explanations

For a given partial assignment of a CSP, there exist many possible next explanation steps, but only one is actually shown to the user. However, it is not clear *which explanation* would be *most helpful* to a user. This chapter investigates how to learn explanation preferences directly from users and integrate it into the current explanation techniques.

This chapter has been submitted to the 30th International Conference on Principles and Practice of Constraint Programming (CP2024).

5.1 Introduction

A prominent problem in *explainable constraint solving* is explaining why a model is unsatisfiable [95, 17, 80, 87]. Many methods tackle this problem by providing a deductive explanation in the form of a *minimal unsatisfiable subset* (MUS), which is a subset of the model's constraints that renders the problem unsatisfiable, and thus yielding an inconsistency. In Chapter 3, we have shown that MUS can also be used to explain why a value is assigned to a specific variable using a one-to-one correspondence between non-redundant explanations and Minimal Unsatisfiable Subsets (MUSs) of a derived unsatisfiable formula.

In general, given a partial assignment of a CSP, there are many possible next explanation steps (MUSs), but only one is actually shown to the user. [74] proposes the cardinality of the MUS, in other words the size of an explanation step, as a potential measure of explanation quality and, as such, compute the *(cardinality-)smallest* MUS (SMUS). In Chapter 3, we take a pragmatic approach, and search for a step that only uses few constraints and few previously derived facts. In fact, we take a handcrafted linear objective function (in terms of the number of constraints and facts used) and *heuristically* try to find a step with a low cost. In Chapter 4, we develop this approach further and propose algorithms that can find an *optimal* explanation step with respect to such a given weighted linear objective function, for a hand-crafted set of weights. The results of the experiments in Section 4.5.1 show that the introduction of a cost function helps to find explanation steps that are more understandable than taking size into account.

It can challenging for human users to explicitly specify weights that consistently lead to good solutions in a specific problem domain [103]. We aim to infer such weights indirectly from user feedback.. Furthermore, as Miller [104] noted:

> "It is fair to say that most work in explainable artificial intelligence uses only the researcher's intuition about what constitutes a *good* explanation".

Indeed, our step-wise explanation framework leaves open the question of what exactly a *good* objective function is, or how to obtain it.

Ideally, a good objective function should approximate human interpretability, so that the returned explanation step is helpful to the user in understanding what can be derived next. This, however, might well depend on the person the explanation is given to.

In this chapter, we investigate whether we can use *preference learning* to learn what a *helpful* explanation is directly from data:

> *Can we learn which explanation steps are most helpful to a user?*

We hence investigate whether we can learn a preference scoring function over explanation steps, and therefore MUSs, from labelled data.

For this, we will ask *pairwise preference* queries [2, 32, 66, 82]. That is, we will ask users to indicate whether one, the other, or neither of two explanations is preferred. We can then use machine learning methods to learn a *preference score* over a *feature representation* of explanation steps. This preference score can then be used to evaluate other, unseen, explanation steps and select the best scoring one among them.

An important concern to keep in mind here is whether the preference score over the feature representation can be reformulated as a linear objective function over the elements of the explanation. If so, it can be plugged directly into our existing algorithms to generate a most preferred explanation. On the other hand, if we use more advanced learning methods, we may get better results at prediction time, but we will need a different mechanism to compute the best-scoring explanation.

We will evaluate the learning approach on the pedagogical domain of sudoku puzzles, as the generated MUSs (explanations) are intuitive to visualise over the Sudoku grid. This makes it a problem domain where many users have the necessary domain knowledge to interpret the explanations, and for which labelling pairs of explanations can be done quickly.

Contributions. To address the challenge of learning from a user's explanation preferences, we contribute the following:

1. We identify domain-agnostic as well as domain-specific features for MUSs by counting over different subsets of constraints how many are present in the MUS.
2. We compare feature choices and use different point-wise and pair-wise learning-to-rank methods, showing that in the Sudoku domain, domain-specific features are essential to accurately discriminate between MUSs, and multiple learning techniques can do this well.
3. We demonstrate that we can not only learn how to *discriminate* between two smallest MUSs, but that a few iterations of iterative learning with linear predictors allows learning a scoring function that can be directly used to *generate* good MUSs.

Chapter structure. In Section 5.2, we formalise our methodology for learning pairwise preferences and generating explanations respectively, and illustrate it using the Sudoku use case in Section 5.3. In Section 5.4, we evaluate how well ML models are able to capture a user's explanation preferences. Finally, Section 5.5 discusses the limitations of the current approach and future work.

5.2 Data-driven Methodology

In Section 3.4.1 of Chapter 3, we introduce explanations of inference steps, and show in Section 3.6.3 that they are closely related to the notion of MUSs. More precisely, computing an explanation $(\mathcal{E} \subseteq I, \mathcal{S} \subseteq C, \mathcal{N})$ of $\mathcal{N} = \{x = v\}$ for given constraints C and facts I corresponds to extracting a MUS of $I \wedge C \wedge \{x \neq v\}$. In Section 4.2, we introduced the `OCUS` algorithm for computing explanations that are optimal with respect to a given cost function characterizing an explanation's difficulty. In the rest of this chapter, we refer to an explanation step $(\mathcal{E}, \mathcal{S}, \mathcal{N})$ as $\mathbf{e}$, and will define a linear scoring function $f(\mathbf{e})$ for use with OCUS.

Our goal is to learn a scoring function that predicts a score for each possible explanation in a given problem domain. We will use ML methods on user labelled data, which requires defining a feature $\phi(\mathbf{e})$, that maps the explanation $\mathbf{e}$ to fixed-size vector of numbers (features). This allows to learn a function $f_\theta(\phi(\mathbf{e}))$ with θ being the parameters of the learning model. A linear function corresponds to $\theta^\top \phi(\mathbf{e})$, which is linear in $\mathbf{e}$ if $\phi(\mathbf{e})$ is.

5.2.1 Feature Encoding

Technically, the OCUS algorithm introduced in Section 4.2.2 allows us to search for an Optimal Unsatisfiable Subset OUS, given a weight for every constraint. However, since the relevant constraints may differ between problem instances in the same domain, we cannot directly learn a weight for each individual constraint directly.

The feature mapping $\phi(\mathbf{e})$ allows us to abstract from that. Each feature should represent a meaningful property of a MUS. A straightforward property is the number of constraints in a MUS, minimizing this leads to SMUS. However, there may be many SMUSs, so additional properties are required to discriminate between them. Additionally, to apply the OCUS algorithm, we need to ensure that the objective is a linear function of the constraints. To do this, we define each feature as a linear sum over a meaningful subset of the constraints. In other words, if we can *group* the constraints into different groups, the *count* of elements within each group serves as the feature value. We can then use machine learning to learn a weighted combination over these counts.

Example 19. *Consider a simple Boolean satisfiability problem:*

$$a, b, \neg a \vee b, \neg a \vee \neg b$$

One SMUS is $a, \neg a \vee b, \neg a \vee \neg b$ and another is $a, b, \neg a \vee \neg b$. Both have size 3 and can not be discriminated based on size, though one can imagine most people finding the second one easier to interpret. If we group constraints into 'unit clauses' and 'other clauses', then the first would have counters (1,2) and the second (2,1) and any function assigning larger weight to the 'other clauses' feature would always generate MUSs of the second kind.

Features for explaining unsatisfiability We consider the case where a constraint is not restricted to a clause (as in SAT solving), but rather can be any expression over finite-domain variables (as is typical in CP). This means that we can group the constraints based on the **syntactic form** of expression used. For instance, we can group constraints into facts ($x = v$), pairwise inequalities ($x_1 \neq x_2$)), linear constraints (($\sum_i(w_i \cdot x_i) \geq t$), particular types of global constraints (e.g. $\texttt{alldifferent}(S)$)), etc. These features are independent of the problem domain and therefore **domain-agnostic**.

We can also identify additional **domain-specific** groupings. For example in the sudoku use case explained below, we will differentiate between alldifferent constraints over rows, over columns and over blocks. Syntactically they are all the same, but visually, users might have preferences of one over the other.

Features for explaining a fact $x = v$ When we wish to explain that $E \wedge S \Rightarrow (x = v)$ we compute a MUS of the unsatisfiable $E \wedge S \wedge (x \neq v)$. In this case, we can exploit the knowledge that we want to explain $x = v$.

We propose to **measure the distance** from any constraint to the fact $x = v$. More specifically, consider the (bipartite) variable-constraint graph in which variables are connected to the constraints they are involved in. We can group constraints by the *distance* (in this graph) to the variable x. For instance, the constraints at distance one are precisely those that mention x. For the special case of constraints that themselves are simple facts, i.e. $y = z$, we can use knowledge of the target fact $x = v$ in two ways: we can group the facts based on the distance of x to y or we can group them based on whether they assign the same value ($v = z$) or a different value ($v \neq z$). As these features are only dependent on the variable-constraint graph, they are **domain-agnostic**. Also here we can identify additional domain-specific groupings; we will clarify this with sudoku examples in Section 5.3 below.

5.2.2 ML Model Training

Our training data consists of a set of triple in the form of $(\mathbf{e}_1, \mathbf{e}_2, l)$, where $\mathbf{e}_1$ and $\mathbf{e}_2$ are two alternative explanations for the same unsatisfiable problem, and l indicates which one the user preferred. Given that users are expected to rank one explanation higher than another, the learning task can be framed as a Learning-to-Rank (LtR) problem [96]. LtR aims to learn a ranking function that explicitly orders any new set of items. We particularly focus pointwise classification and pairwise classification, which learn scoring function. As is common in LtR, we will disregard the triples for which the user has no preference.

Pointwise classification The LtR problem can be framed as a standard classification task. This is a *pointwise* LtR approach because this approach treats each explanation independently. Given a tuple $(\mathbf{e}_1, \mathbf{e}_2, l)$ we generate two data points: $(\phi(\mathbf{e}_l), +1)$ for the explanation that was chosen by the user, and $(\phi(\mathbf{e}_o), -1)$ for the other one. We can then use any off-the-shelf classifier for training, including linear logistic regression [70], decision trees [21], random forests [3], SVM classifiers [124] and more. Out of these, the logistic regression (when dropping the logistic function computation at inference time) and the SVM classifier (when used with a linear kernel) produce linear scoring functions.

Pairwise learning-to-rank In the pairwise LtR formulation, each $(\mathbf{e}_1, \mathbf{e}_2, l)$ can be used directly as training data. Let $\mathbf{e}_l$ be the explanation chosen by the user, and $\mathbf{e}_o$ be the other. Then we know that the former is preferred over the latter: $\mathbf{e}_l \succ \mathbf{e}_o$. In pairwise learning-to-rank, the

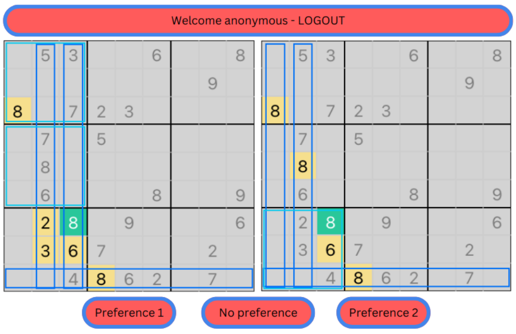


Figure 5.1: Two possible explanation steps from the Sudoku app for $\{\texttt{cells}[7,3] = 8\}$. Facts are highlighted in yellow, while constraints are the blue rectangles.

methods learn a function $f_\theta(\phi(\mathbf{e}))$ such that $f_\theta(\phi(\mathbf{e}_l)) > f_\theta(\phi(\mathbf{e}_o))$ for every labeled query. Examples of pairwise LtR methods are SVMRank [78] and XGBoostRanker [28]. Only SVMRank with a linear kernel will produce a linear scoring function.

5.3 Use-case: Step-wise Explanations for Sudoku

To evaluate the feasibility of using ML to learn a preference scoring function over unsatisfiable subsets, we will use explanation steps in the context of the sudoku puzzle.

An explanation step in Sudoku explains why an empty cells needs to take a particular value. This is then repeated for the remaining empty cells, until all cells in the Sudoku are explained. All cell values must be explainable by the constraints and the initial assignment, because a Sudoku has a unique solution.

For the labelling, we developed a mobile-friendly web application that highlights the facts, rows, columns and squares that are used in an explanation, as shown in Fig. 5.1. Once familiar with the concept of explanation steps, we observed that users tend to have explicit preferences between them, and that they can label pairs relatively quickly.

We store the response as a triple $(\mathbf{e}_1, \mathbf{e}_2, l)$ where $l = 0$ if no preference is given, $l = 1$ if the first is chosen and $l = 2$ if the second was chosen.

Our generation of explanation steps is as follows:

1. We use QQWing [114] to generate *intermediate* Sudoku puzzles. We opt for this level to

avoid limiting participation to only users with expert knowledge.

2. We use `OCUS` to enumerate a sequence of explanation steps by computing the smallest MUS.

3. For every step, we then either generate the second smallest MUS, or use a learned cost function to generate an OUS depending on the experiment.

Feature encoding All constraints for sudoku are either facts ($x = v$) or alldifferent constraints. So the two most basic **syntactic features** are: number of facts and number of constraints. Using the concept of **distance to the fact** explained above, we can further group the constraints into *adjacent* constraints (constraints at distance 1 from the fact to explain $x = v$) and other constraints (distance > 1). The facts can also be split into adjacent facts (the variables of the adjacent constraints) and other facts, as well as whether the fact assigns the same value as $x = v$ or a different one. These features are **domain-agnostic**, in our case we selected a subset of 5 linearly independent features, see Table 5.1 for the detailed list.

Feature	Description
number_adjacent_constraints	Number of used adjacent constraints
number_other_constraints	Number of non-adjacent used constraints
number_adjacent_facts_other_value	Number of used and adjacent facts that have a different value from the explained fact
number_other_facts_same_value	Number of used and non-adjacent facts that have the same value of the explained fact
number_other_facts_other_value	Number of used and non-adjacent facts that have a different value from the explained fact

Table 5.1: Features for Syntactic-only training

For the **domain-specific features**, we can further split the above constraint groups into separate subgroups for row, column and block constraints. We also introduce separate counters for adjacent row/column or block facts; it is not an issue that adjacent block facts overlap with row/column facts. The domain-specific features are considered together with the domain-agnostic ones, resulting in 12 linearly independent features summarised in Table 5.2.

Feature	Description
number_adjacent_facts_other_value	Number of used and adjacent facts that have a different value from the explained fact
number_other_facts_same_value	Number of used and non-adjacent facts that have the same value of the explained fact
number_other_facts_other_value	Number of used and non-adjacent facts that have a different value from the explained fact
number_adjacent_block_cons	Number of of adjacent used block constraints
number_adjacent_row_cons	Number of of adjacent used row constraints
number_adjacent_col_cons	Number of of adjacent used column constraints
number_other_block_cons	Number of non-adjacent used block constraints
number_other_row_cons	Number of non-adjacent used row constraints
number_other_col_cons	Number of non-adjacent used column constraints
number_adjacent_block_facts	Number of used and adjacent facts from blocks
number_adjacent_row_facts	Number of used and adjacent facts from rows
number_adjacent_col_facts	Number of used and adjacent facts from columns

Table 5.2: Features for domain-specific training

5.4 Experiments and Results

We will evaluate how well machine learning methods are able to capture user preferences over sudoku explanation steps. The research questions are:

RQ1 How important is the feature representation to learn good scoring functions?

RQ2 What learning techniques (pointwise and pairwise) perform best?

RQ3 How well can the method be used for generating new explanations, and does this improve over multiple feedback iterations?

Setup We use a single core of a 6-core INTEL(R) 2.6 Ghz processor with 16GB of RAM for all experiments. We implement CSP modeling and OUS-based explanation generation using CPMPY 0.9.16 [64], and ML classifiers from SCI-KIT LEARN 1.3.0 [115], XGBOOST 1.7.6 [28] and PYSVMRANK using 5-fold cross-validation. The source code of the algorithms implemented as well as all user-labeled data will be made available on publication.

5.4.1 Learning a discriminator function

We evaluate which type of features and learning methods are suitable to learn to discriminate between two explanations. For around 250 explanation steps, we generate the top-2 SMUSs and let users label these pairs. Table 5.3 depicts the performance of the models, using 5-fold cross-validation independently for data from 6 different users. We evaluate it according to:

Correct If the scoring function gave the highest score to the user's chosen explanation.

Equal If the scoring function gave the same score to both explanations.

Wrong If the scoring function gave the user-chosen explanation the lowest score.

In Table 5.3 we make a difference between only using the number of constraints and facts as features, using only the domain-agnostic features, and using both domain-agnostic and domain-specific features. As benchmarks, we consider two handcrafted linear functions: the one that is used for computing the SMUS [74] and another from Section 4.5 making a distinction between the number of used alldifferent constraints c and the number of used facts f. In terms of ML techniques we considered linear pointwise classifiers (Logistic Regression, SVM with linear kernel) and a non-linear classifier (Gradient Boosted Trees), as well as a linear ranker (SVM rank with linear kernel) and a non-linear one (Gradient Boosted Tree ranker).

Table 5.3 (top row) shows that only using the size of the MUS leads to a large amount of *Equal* results, which is to be expected since we generate the top-2 SMUSs to choose between. Conversely the handcrafted function from Section 4.5 can distinguish more explanations. Curiously, all ML methods perform slightly worse than this function, perhaps due to the amount of data.

We also see that considering only domain-agnostic features still leads to many *Equals*, meaning that such features are largely insufficient to differentiate between the two SMUSs. At the same time all classifiers perform very similarly.

Finally, when all features are used, we rarely have *Equal* cases. We can also see very good performance for all predictors, linear and non-linear, with an additional benefit for the pairwise ranking methods. This shows that ML methods can learn to discriminate between two SMUSs with very high accuracy, given a labelled dataset. This is true for all users.

5.4.2 Iterative learning a generator function

Given that SVM Ranker (with a linear kernel) returns a linear scoring function, we investigate whether this learned scoring function is suitable for directly *generating* a preferred explanation.

Features	**Classifier**	**Correct**	**Equal**	**Wrong**
# cons, # facts	Handcr. SMUS $(c+f)$ [74]	7.1%	90.1%	2.8%
	Handcr. OUS (20 $\times$ $c+f$)	25.6%	70.6%	3.8%
	Logistic Regression	25.3%	70.6%	4.1%
	SVM Classifier	20.4%	73.0%	6.6%
	SVM Ranker	24.4%	70.7%	4.8%
Domain-agnostic	Logistic Regression	28.8%	64.8%	6.4%
	SVM Classifier	29.9%	64.8%	5.2%
	XGBoost Classifier	28.4%	65.2%	6.4%
	SVM Ranker	29.2%	64.8%	6.0%
	XGBoost Ranker	28.8%	65.7%	5.5%
Domain-specific	Logistic Regression	90.7%	1.2%	8.1%
	SVM Classifier	85.7%	1.4%	12.9%
	XGBoost Classifier	87.3%	1.2%	11.5%
	SVM Ranker	90.8%	1.3%	7.9%
	XGBoost Ranker	91.8%	1.5%	6.7%

Table 5.3: Correctness of Point wise Classifiers and Pairwise ranker methods. We train one classifier per user and compute the average.

To test this, we generate around 50 new pairs of Sudoku explanation steps from a single puzzle. Each pair combines one SMUS and one using the `OCUS` algorithm which optimises the learned scoring function, and we ask the users which one they prefer.

The first row (Iteration 1) in Table 5.4 shows that this initially performs much worse than generating an SMUS. This is because the predictors are only trained on SMUSs and are hence biased towards predicting between MUSs which are already small and subset minimal. Since the learned scoring function is not trained to rank all MUSs, this essentially tests its performance on a large out-of-sample distribution and we can see that it is too specific to the training data.

To address the shortcoming of the data, we perform additional iterations as is common in machine learning with concept drift. In the $i + 1^{\text{th}}$ iteration, the newly labelled explanation pairs are added to the training data. For each iteration we evaluate the prediction accuracy using crossfold validation (3 left columns) and on the newly generated pairs using the learned scoring function (3 right columns).

As the number of iterations increases, the accuracy in predicting the preferred explanation on the training data decreases. This shows that there are fewer 'shortcuts' to take in the data, and the classifier has a harder time balancing the multiple and possibly conflicting preferences in the

	Test data			New generate & label		
Iteration	Correct	Equal	Wrong	ML wins	No preference	SMUS wins
1	94.0%	0.0%	6.0%	0.0%	0.0%	100.0%
2	75.1%	0.0%	24.9%	7.3%	7.3%	65.4%
3	72.2%	0.0%	27.8%	29.8%	8.8%	61.4%
4	70.5%	0.0%	29.5%	44.6%	39.3%	16.1%
5	66.8%	0.0%	33.2%	62.5%	19.6%	17.9%

Table 5.4: Iterative learning approach with SVM Rank (linear kernel) over domain-specific features. In each iteration, new data is labelled by the user and added to the training data of the next iteration.

labelled dataset.

At the same time, we can see that with each iteration, the scoring function becomes better suited at directly generating explanations. In fact, the percentage of cases in which the ML-generated explanation is preferred to an SMUS increases. It is hence learning a preference scoring function that better aligns with the user preferences and can be used as to directly generate explanations..

5.5 Conclusions

In this chapter, we consider the task of learning a scoring function for MUSs, and hence explanation steps, to capture which ones are more preferred by a user. To do so, we propose a principled feature encoding technique for MUSs. This makes it possible to learn a scoring function from a dataset of pairwise comparisons, using learning-to-rank techniques. We demonstrate this approach on the easily interpretable task of explaining sudoku cell assignment. In this domain, using domain-specific features was essential to discriminate between preferred smallest MUSs. Using linear features and linear SVMrank, we further showed that an iterative training approach can learn a scoring function that can be used to directly generate good explanations.

Many avenues for future work exist. First, we have only applied this to the pedagogical Sudoku domain, where it was quick and easy to experiment. This approach can now be validated in more realistic problem domains, such as explaining staff rostering or crew scheduling, where domain experts are harder to find. Our pairwise labelling approach also requires good domain-specific visualisations, so that users can quickly see the differences and express a preference. This potential interaction between visualisation quality and perceived preferences invites a Human-Computer Interaction (HCI) approach to designing and evaluating explanations in domain-specific settings. Our learning techniques can then be applied in a subsequent step.

On the technical side, the sufficiency of a linear scoring function and the effectiveness of the iterative method invite to investigate online, real-time learning approaches. Online methods could require less data, where data labelling is an important bottleneck to the use as well as the evaluation of the techniques in this work. Online learning would also require sufficiently fast `OCUS` algorithms for interactive use, another key bottleneck.

CHAPTER 6

Conclusions, Future Perspectives and Advice for AI Practitioners.

In *'Pursuit of the Holy Grail'* [50], Freuder E. initially laid out a roadmap in 1996 of the challenges and opportunities in constraint satisfaction research. Of the various research directions proposed in that work (i.e. modelling, interaction, explanation, knowledge, agents, evaluation, and synthesis), we have focused on *explanations*, namely (1) *'can we explain a solution?'*; and (2) when a solution is not possible, *'can we understand why?'*.

In this thesis, we have introduced a novel framework for step-wise explaining the maximal consequence of constraint satisfaction problems with a focus on simple, human-interpretable explanations. We address the computational and qualitative challenges to ensure that, at each step, we efficiently generate an explanation that is most helpful to a user.

We propose to first review our contributions and summarise the results in Section 6.1. Next, in Section 6.2, we look at the open challenges and exciting future research directions in explainable constraint solving. Finally, in Section 6.3, we formulate recommendations for AI practitioners in implementing the proposed explanation techniques.

6.1 Summary of the Results

This thesis addressed the problem of step-wise explaining the maximal consequence of constraint satisfaction problems, with a focus on human interpretability. Specifically, we developed an explanation-producing system that is *complete* and *interpretable*. By *complete* we mean that it finds a *sequence* of *simple* reasoning steps that derives all consequences from the given problem specification and a partial solution.

In the following, we identify how each chapter contributes to the development of such an explanation-producing for explaining the satisfiability of CSPs.

6.1.1 Chapter 3: A Framework for Step-wise Explaining CSPs

In Chapter 3, we formally define the problem of step-wise explaining the maximal consequence of a constraint satisfaction problem. We decompose the solution as a sequence of *simple* inference steps, where simplicity is approximated by a cost function. At each step in the sequence, we heuristically guide the search towards low-cost explanations by relying on a one-to-one correspondence between so-called *non-redundant explanations* and Minimal Unsatisfiable Subsets (MUSs) of a derived unsatisfiable formula. For explanations that are still too difficult to understand, we introduced *nested explanations* as a way to refine an explanation step into a sequence of simpler explanation steps using reasoning by contradiction. We empirically show that we are able to generate a sequence of simple inference steps, and that nested explanations help to break down complex inference steps into a sequence of simpler, more understandable explanations.

6.1.2 Chapter 4: Efficiently Explaining CSPs with Unsatisfiable Subset optimisation

In Chapter 4, we tackle the two main weaknesses of Chapter 3 related to MUS-based explanation generation: (1) the *lack of optimality guarantees* with respect to a given cost function, and (2) the *efficiency problems* due to relying on heuristics to guide the search toward low-cost explanations.

To address the lack of optimality guarantees, we have introduced the concept of an optimal constrained unsatisfiable subset (OCUS), which is optimal with respect to a given cost function, and satisfies certain structural constraints. Given a cost function that approximates the human interpretability of explanations, integrating OCUS into explanation generation ensures that we find optimal (easy) explanations and that we explain one fact at each explanation step. Our experiments show that a *MUS*-based solution often misses very 'cheap' explanations (with respect to a given cost function), confirming the need for a cost-based OCUSs approach.

To compute an OCUS, we proposed an algorithm based on the implicit hitting-set duality between minimum correction subsets (MCS) and minimum unsatisfiable subsets (MUS). Similar to the hitting-set–based techniques used in MaxSAT solving [34, 121], we combine the strengths of Mixed Integer Programming (MIP), for computing optimal hitting sets with respect to a given cost function, and Boolean Satisfiability (SAT), for checking satisfiability of a candidate subset.

The main bottleneck of `OCUS` (and other hitting-set–based algorithms) is having to repeatedly compute (optimal) hitting sets [34]. To compensate, we have developed methods for enumerating correction subsets that explore the trade-off between efficiency and quality of the subsets generated. We further optimise the efficiency of explanation generation by introducing explanation-specific versions of the `OCUS` algorithm and *incrementality* to reuse information between consecutive O(C)US calls. Our empirical analysis shows that *incrementality* between explanation calls greatly reduces the total number of sets-to-hit that need to be computed, as well as the time to generate an entire sequence. We are able to reduce this number even further by computing disjoint correction subsets, that remain valid throughout the sequence generation process. Overall, our experimental evaluation on a diverse benchmark of CSP instances shows that we are able to quickly generate a first explanation and significantly reduce the time to generate a full sequence, even for large problems.

6.1.3 Chapter 5: Learning Preferences over Formal Explanations

In Chapter 5, we address the question left open by Chapter 3 and Chapter 4, namely how to characterise which explanation is actually *most helpful* for a user. We proposed to generate a dataset of preferences over pairwise cardinaility-minimal explanations for training an ML model to learn a cost function that reflects user preferences. Next, we investigated training *probabilistic classifiers* and *learning-to-rank* methods over a *feature representation* of explanations, and their integration into the explanation generation methods of Chapter 3 and Chapter 4.

An important concern to keep in mind is whether the learned function can be reformulated as a linear objective function over the elements of the explanation. If so, the learned function can be plugged directly into our O(C)US-based explanation algorithms to generate a most preferred explanation.

Our experimental evaluation shows that a simplistic cost function, based on the number and type of constraints and facts, is not able to fully capture user preferences. In contrast, introducing more complex features improves the effectiveness of explanations by capturing the interaction between different components (constraints and facts) of the explanation. Furthermore, our results show that it is important to include domain-specific features in addition to domain-agnostic features. While the integration into OCUS-based explanation generation methods limits us to linear objective functions, we show that these are sufficiently expressive to capture users' explanation preferences.

Finally, the results of the user study show that the cost function extracted from a trained learning-to-rank model with a linear kernel (SVM Ranker) can be used to generate preferred explanations. However, it is important that the training data set includes explanations that are outside the original sample distribution, i.e. not just cardinaility-minimal explanations, but also explanations with (too) many constraints or facts.

6.2 Future Work

In this section, we identify key research directions for the explanation methods developed in this thesis.

6.2.1 Implicit hitting set algorithms for Efficient Subset optimisation

The implicit hitting set duality between minimum correction subsets and minimum unsatisfiable subsets (see Proposition 1) is used not only for extracting minimal unsatisfiable subsets [117, 8, 94, 74], and OCUS, but also in MaxSAT [33, 35, 121, 122, 45, 12, 111]. Recently, Saikko et al. [122] proposed an abstract framework for describing hitting-set–based algorithms in KR, including optimisation, focussing on MaxSAT terminology. In this context, implicit hitting algorithms in MaxSAT (resp. OCUS) rely on a combination SAT solver to extract unsatisfiable cores (resp. correction subsets from MaxSAT solutions) and an integer programming (IP) solver to optimise hitting sets over unsatisfiable cores.

Unsatisfiable Subset optimisation with Non-linear Objectives

In OCUS and the implicit hitting-set framework for MaxSAT of [122], the goal is to optimise an objective function f and for efficiency reasons, a MI(L)P solver is used for compute optimal hitting sets with respect to f. However, both framework allow for more sophisticated cost functions, such as non-linear objectives, as long as $f(S) < f(S')$ whenever $S \subset S'$. In Chapter 5, we introduced features to characterise an explanation step. A feature is computed over the elements of an explanation step, in other words, subsets of a corresponding unsatisfiable formula. A limitation is that some proposed features are non-linear, meaning that they cannot be encoded as a linear combination of decision variables. Future work there should investigate methods for encoding non-linear objectives for computation hitting sets [27, 121].

Efficient (Incremental) Constraint Solving

Many hitting sets are computed using the `OCUS` algorithm. To improve its efficiency, we initialize the MIP solver once, and reuse throughout the `OCUS` iterations. A similar observation was noted for MaxSAT by Davies and Bacchus [34]). However, Table 4.1 points out that an exponential number of satisfiable subsets (resp. unsatisfiable cores for MaxSAT) need to be computed, and many of the hitting sets are similar. To deal with this, Berg et al. [12] introduce an abstract/compact representation of unsatisfiable cores. Other techniques, such as symmetry breaking in combinatorial optimisation [37, 41, 20], could benefit the `OCUS` algorithm to improve its efficiency. Other ideas from implicit hitting-set–based MaxSAT solving to speed up `OCUS` include local search strategies to add additional constraints on the sets-to-hit to postpone calling the hitting-set solver [35].

6.2.2 A Most Helpful Explanation *Step* for the User

In Chapter 5, we investigated different ML methods for learning which explanation is most helpful to a user, and their integration for directly generating a preferred explanation.

Learning an objective function

The methods for directly optimising an explanation impose limitations on the objective functions that can be expressed, as well as machine learning techniques that can be used to learn such a function. Table 5.4 of our experimental evaluation show that many of the generated explanations were not MUSs. In fact, they contained redundant information, such as too many facts or too many constraints. The learned objective function was not strictly monotonically increasing, and therefore loses the non-redundancy property of explanations. Two sources could cause this. First, the data provided to the machine learning models only compares non-redundant explanations between each other. Furthermore, we already focus on top 2 smallest ones. In future work, the training of the ML model should include pairwise explanations comparing non-redundant explanation and explanation with redundant facts and/or constraints. This would ensure that the scoring function learned by the ML model, for example, scores non-redundant explanations higher than redundant ones. Second, the machine learning methods used do not constrain the learned function, and therefore do not provide any monotonicity guarantee. De and Chakrabarti [36] propose neural models for learning monotone functions that could be used to guarantee the non-redundancy of generated explanations.

Active Learning

In our user study of Chapter 5, users were asked to input their preferences on a large dataset of pairwise explanations. Obtaining a similar dataset for other problems can therefore prove to be a difficult, time-consuming and potentially expensive process [128]. Active learning literature [128] proposes strategies to identify which instance should be labelled, that is most likely to benefit the machine learning model. For explanation generation, this corresponds to identifying *which pair should be shown to a user*.

6.2.3 Improving Explanation *Sequences*

The algorithm for constructing the sequence greedily looks for simple explanations at every step. A similar greedy process is used to construct a nested explanation sequence starting from the negation of a consequence until a contradiction is reached. In contrast to step-wise explaining satisfiability, there is no guidance on which fact should be explained in to reach the contradiction, and nested explanations can contain many redundant steps. Bleukx et al. [17] propose an algorithm for greedily constructing the sequence and filtering algorithms to remove redundant (and unnecessarily complex) steps. A challenge in our work and in that of [17] is how to optimise the sequence of explanations according to a metric. For example, by reordering the steps so that deriving a fact earlier in the sequence might reduce the overall complexity of the sequence.

6.3 Advice for AI Practitioners

As AI systems employ more advanced reasoning mechanisms and computational power, it has become increasingly difficult to understand why certain decisions are made. Explainable AI (XAI) research aims to address the need for trustworthy AI systems to understand why the system made a decision. For constraint satisfaction, this need for explainability manifests itself with the development of tools to explain *why a problem cannot be solved*, and *how to repair its feasibility*.

The type of explanation expected. An array of explanation techniques have been developed in constraint satisfaction (see [40] for a short survey on explanation in constraint satisfaction). However, there is still confusion about the types of explanations, i.e. subsets of the problem constraints, that exist, and which question they answer.

For *unsatisfiable* problems, a minimal unsatisfiable subset (MUS) is extracted as an explanation to understand *why* a subset of the constraints cannot be satisfied. However, there exist many MUSs of different sizes for a given problem. Which one should be shown? Our recommendation would be to extract either a *cardinality-minimal* (minimal size) one [74], a preferred one [80] if preference can be expressed as a relationship between constraints or to rely on an objective function to assign a preference score to subsets of constraints, see Section 4.2.

In contrast to a MUS, a minimal correction subset (MCS) explains which constraints need to be relaxed or removed in order to restore feasibility of the problem.

In this thesis, we tackle a third question: *how to explain a solution*. More precisely, we break down explaining the maximal consequence of a problem as a sequence of simple inference steps. For problems that can be solved without search, e.g. problems such as logic grid puzzles with a unique solution, this means explaining the entire problem-solving process. For problems involving search, this means explaining the inference steps in one search node. Each (preferred) explanation of an inference step explains *why* a variable is assigned a value $x = v$ (or not $x \neq v$), by extracting a (preferred) MUS of a derived unsatisfiable formula (see Section 3.6 for the details of how this is done).

Applications The explanation-generating techniques we developed can be applied in a multitude of use cases. For instance, it can explain the entire sequence of reasoning, such that a user can better understand it. In case of an unexpected outcome, our methods can debug the reasoning system or the set of constraints that specify the problem. As our approach starts from an arbitrary set of facts, it can also be used as a virtual assistant when a user is stuck in solving a problem. The system will explain the simplest possible next step. In an interactive setting, the system can explain how it would complete a partial solution of a user. Such explanations can be useful in the context of *interactive configuration* [46] where a domain expert solves a problem (e.g., a complicated scheduling or rostering problem) but is assisted by an intelligent system. The system in that case typically does not have all the knowledge required to solve the problem, since certain things are hard to formalise, especially when personal matters are involved. In such case, the system cannot find the optimal solution automatically, but it can help the expert, for instance by propagating information that follows from its knowledge base. In case something undesired is propagated, the user might need an explanation about *why* this follows; this is where our methods can be plugged in. Finally, our measures of simplicity of reasoning steps can be used to estimate the difficulty of solving a problem for a human, e.g. for gradual training of experts.

Our explanation-generation techniques are *independent of the consistency level of the propagation* and do not require *augmenting the solver and propagators with explanation capabilities*. As a consequence, they can be directly implemented on top of existing software to provide support in decision-making. In practice, our methods require careful considerations of related to the kind of explanations expected, their efficient generation and the target audience who will receive the explanation.

In the following, we provide an overview of recommendations to help in implementation of our explanation techniques.

Practical implementation There are several challenges when extending an existing decision support system with the explanation techniques we developed. Solvers may not inherently support or simply provide access to the features needed for generating explanations, such as methods for extracting minimal unsatisfiable subsets. Integrating with other solvers might require interfacing with other programming languages (text-based, object-oriented or more), or even with other modelling formalism (CP, SAT, SMT) to fit the input specification. Finally, many of the solvers used in industrial applications are still commercial solvers for larger models due to their performance advantage, although there are many (free) open source solvers available today. Recent initiatives aim to reduce this engineering effort required to interface with different solvers. For example, CPMpy [64] is a solver-independent modelling library for constraint programming that interfaces with MIP, SAT, CP, and SMT solvers, and allows for incremental solving[1]. Another advantage in using CPMpy is that our explanation techniques are implemented, and many examples of use-cases for it are available[2].

In contrast, the development of a decision support system for a new use case provides an opportunity to model the problem around the explanation.

Modelling However, the formalism and abstraction level used in modelling the constraints and the facts will directly impact the kind of explanations generated. Different encodings of constraints exist for a given problem, as explained in Section 1.1.1. A good encoding should reflect the expected level of detail in an explanation, and should be adapted to the level of expertise of the end-user. For instance, in Example 10 of Section 2.1, we model the Sudoku puzzle as a CSP using the *alldifferent* global constraints. A lower level encoding, for example using binary equality and inequality constraints, captures more detailed interactions between the constraints, facts and facts-to-explain. Since a higher-level encoding likely propagates more facts and domain-reductions, it may also lead to shorter sequences. In the following example, we illustrate this effect.

Example 20. *Consider variables x_1, x_2, x_3 with domains $x_1 \in \{1,2,3\}, x_2 \in \{2\}, x_3 \in \{2,3\}$.*

$$(x_1 \neq x_2) \wedge (x_2 \neq x_3) \wedge (x_1 \neq x_3) \quad \textit{vs} \quad \textit{alldifferent}(x_1, x_2, x_3)$$

Explaining why $x_3 = 3$ can be done with:

- $x_2 = 2 \wedge \textit{alldifferent}(x_1, x_2, x_3) \Rightarrow x_3 = 3 \wedge x_1 = 1$. *Notice how the assignment $(x_2 = 2)$ and the alldifferent constraint also propagate $x_1 = 1$.*

- *In contrast, a lower level encoding for the same alldifferent constraint, provides more detail. First, $x_2 = 2 \wedge (x_2 \neq x_3) \Rightarrow (x_3 = 3)$. No more information can be propagated from using this constraint only.*

[1]The documentation for CPMpy is available at `https://cpmpy.readthedocs.io/en/latest/index.html`.

[2]`https://github.com/CPMpy/cpmpy`

Which fact to explain? In most of the examples in this thesis, we focus on explaining variable assignments, i.e. why variable x is assigned value v. Our explanation techniques also handle explaining why a variable x is not assigned value v, to explain why values are not allowed in the variable domain. For Example 20, this corresponds to explaining $x_3 \neq 2$. In practice, we do not fully support their visualisation, and leave this open for future work.

In Section 3.5, we introduce nested explanations (Section 3.5) to zoom in on a difficult explanation step, and compute them using reasoning by contradiction. We assume the negation of a consequence, and generates a sequence of explanation steps until an inconsistency is reached. We observed that the system sometimes finds minimal explanations of $X - 1$ facts $x \neq v$ instead of a corresponding $x = v$ fact when generating these explanations (Section 3.9.4). However, for human reasoners, $x = v$ is often easier to grasp. A preference for these kinds of explanations should be reflected in the cost function, see Section 3.7.

Conveying explanations For the examples of this thesis, we convey our explanations using the visual support that was available (Sudoku or Logic Grid) and the natural language description of the problem (e.g. natural language text, clues, alldifferent, ...). While this can require considerable (engineering) effort, we believe visualisations are important for users to understand the decision, such as a schedule as shown in Example 5 of the (Chapter 1).

An alternative option is to provide natural language descriptions of explanations, e.g. using a template that is completed using textual descriptions of the constraints. For example, Sqalli and Freuder [129] equip inference rules with explanation templates such that each propagation event of an inference rule also has a templated explanation. With the rise of large language models LLMs, there is an opportunity to go beyond templated explanation generation, and instead provide an environment in which users can interact with the explanations. For example, LLMs can be used to paraphrase, simplify a textual explanation, or even tailor the explanations based on the level of expertise of the user. These lower the technical knowledge barrier and make explanations more accessible to non-experts.

For problems that cannot be easily interpreted using text or visually, one can make use of a (constraint) graph to visualise interactions between constraints and facts. For example, Senthooran et al. [126] make use of both a graph representation and a textual description.[1] Other options for visualisation exist; this goes hand in hand with the abstraction level at which explanations are given, and are an essential part of human understandable explanations. In this thesis, we used visualisation that felt natural to us, but a deeper study on the effect of visualisations on explainability and understandability.

[1]An example is provided for the nurse scheduling problem in `https://github.com/CPMpy/XCP-explain/`.

Preferred Explanations Defining a metric (e.g. a cost function) to measure the human interpretability of an explanation is challenging. In Chapter 3 and Chapter 4, we handcrafted cost functions to estimate the difficulty of understanding an explanation based on our experience as constraint programming researchers. Another cost function could be defined as a weighted sum, where each constraint is associated to a weight, such as counting the number of variables that are in a constraint, or estimating the difficulty of using a type of constraint. Keep in mind that the type of cost function imposes limitations on the type of explanation techniques that can be used. Either an optimal explanation can be generated, or candidate explanations should be generated and the best one should be chosen. In practice, while such simplistic functions do give satisfying results on puzzles, they do not scale well, especially for large problems involving thousands of constraints. Chapter 5 demonstrates the importance of learning it directly from (end) users, based on a feature representation of explanations, see Section 5.3 for examples for the Sudoku puzzle. The results of our experiments show that only using problem-agnostic features is not sufficient (29% accuracy), and problem-specific features are needed to improve the accuracy up to 90%. However, defining such problem-specific features is non-trivial. In that case, a textual or visual representation of explanations provides an opportunity to identify patterns in the explanations preferred by users, which can be encoded as features. For example, in Sudoku, we use the specific grid structure to separate constraints into row, column, and block constraints.

Target audience Our user study demonstrates that preferences are not universal, and that ML models perform better when trained on individual preferences rather than combined preferences, see Section 5.4. If explanations are shown to users with different levels of expertise in the application domain, then this should be captured in the training process. For example, a user could be asked about their level of expertise; this information can then be used to associate learned explanation patterns to expertise.

Efficiency recommendations In our work, we have broken down explaining the maximal consequence of a problem as a sequence of simple inference steps. To do so, we greedily build the sequence by selecting the *simplest* explanation out of all candidate explanations generated for each fact to explain.

Computing an explanation for a new fact can be expensive:

1. A minimal unsatisfiable subset (MUS) of a derived unsatisfiable formula is extracted to identify which constraints and derived facts can be used to derive that new fact. However, there exist many MUSs for an unsatisfiable formula, and the ‘best’ one (the least scoring one) is selected according to a cost function. If a cost function can be implemented in MIP,

then we can directly generate an optimal one. If not, either a collection of MUSs need to be enumerated and the best one is selected, or a MUS-based heuristic can be used to speed up the candidate explanation generation, similar to the one introduced in Section 3.6.3.

2. The constraints and facts are propagated to determine if other (new) facts be explained as well.

To generate optimal explanation steps (Algorithm 4.1), we proposed an algorithm to compute an OCUS (Algorithm 4.3) as well as explanation-specific versions of it: `OCUS_Bound` (Algorithm 4.5) and `OCUS_Split` (Algorithm 4.6), an improved version of `OCUS_Bound`. Our experiments show that `OCUS` is better in case only 1 explanation is expected, and `OCUS_Split` is faster at generating the whole sequence. The main bottlenecks in these implicit hitting-set-based algorithms are, in order, repeatedly computing a hitting set over an incremental collection of sets-to hit, computing correction subsets, and checking satisfiability of a hitting set. Solver incrementality is key to speeding up OCUS generation (see Section 4.2.2) for parts that require repeatedly solving a similar (incremental) problem. For the optimal hitting set problem, we instantiate an (incremental) hitting set solver once, and re-use it throughout the `OCUS` iterations. A similar observation holds for checking satisfiability of subsets of constraints for a given (fixed) formula. We reify the constraints by introducing an assumption variable for each constraint, and instantiate the SAT solver once. Satisfiability of a subset of constraints becomes as easy as checking if the corresponding set of assumption variables is satisfiable or not. Finally, finding small correction subsets is crucial for speeding up finding an OCUS, and a good balance should the found to trade-off efficiency of generating a correction subset and its quality (Section 4.2.3). Out of the multiple correction subset enumeration methods introduced in (Section 4.4.1), we recommend using a sat-based heuristic called *Multi* `SubsetMax-SAT`.

Nested explanations In Section 3.5, we introduced nested explanations to zoom in on a difficult explanation step, and compute them using reasoning-by-contradiction. We observed that the generated sequences contained many redundant steps that were not required to explain the contradiction. To cope with this, we filter out the redundant explanations. Another approach, introduced in [17], introduces a heuristic to generate a better sequence, and filtering algorithms to remove all redundant steps and derive only the facts used to explain the contradiction.

With these recommendations, we hope to motivate AI practitioners in using the explanation techniques for real-world problems.

References

[1] Amina Adadi and Mohammed Berrada. Peeking inside the black-box: A survey on explainable artificial intelligence (XAI). *IEEE Access*, 6:52138–52160, 2018. doi: 10.1109/access.2018.2870052.

[2] Nir Ailon. Active learning ranking from pairwise preferences with almost optimal query complexity. *Advances in Neural Information Processing Systems*, 24, 2011.

[3] Jehad Ali, Rehanullah Khan, Nasir Ahmad, and Imran Maqsood. Random forests and decision trees. *International Journal of Computer Science Issues (IJCSI)*, 9(5):272, 2012.

[4] Christian Alrabbaa, Franz Baader, Stefan Borgwardt, Patrick Koopmann, and Alisa Kovtunova. Finding good proofs for description logic entailments using recursive quality measures. *Automated Deduction-CADE*, pages 12–15, 2021. doi: 10.1007/978-3-030-79876-5_17.

[5] Franz Baader, Ian Horrocks, and Ulrike Sattler. Chapter 3 description logics. In *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 135–179. Elsevier, 2008. doi: 10.1016/S1574-6526(07)03003-9.

[6] Fahiem Bacchus and George Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *Computer Aided Verification: 27th International Conference, CAV 2015, USA, Proceedings, Part II*, pages 70–86. Springer, 2015. doi: 10.1007/978-3-319-21668-3_5.

[7] Fahiem Bacchus and George Katsirelos. Finding a collection of muses incrementally. In *Integration of AI and OR Techniques in Constraint Programming: 13th International Conference, CPAIOR*, pages 35–44. Springer, 2016. doi: 10.1007/978-3-319-33954-2_3.

[8] James Bailey and Peter J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *Practical Aspects of Declarative Languages*, pages 174–186. Springer, 2005. ISBN 978-3-540-30557-6. doi: 10.1007/978-3-540-30557-6_14.

[9] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador Garcia, Sergio Gil-Lopez, Daniel Molina, Richard Benjamins, Raja Chatila, and Francisco Herrera. Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Information Fusion*, 58:82–115, 2020. ISSN 1566-2535. doi: 10.1016/j.inffus.2019.12.012.

[10] Jaroslav Bendík and Ivana Černá. Must: minimal unsatisfiable subsets enumeration tool. In *Tools and Algorithms for the Construction and Analysis of Systems: 26th International*

Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Ireland, Proceedings, Part I 26, pages 135–152. Springer, 2020. doi: 10.1007/978-3-030-45190-5_8.

[11] Jaroslav Bendík and Ivana Černá. Replication-guided enumeration of minimal unsatisfiable subsets. In *Principles and Practice of Constraint Programming: 26th International Conference, CP 2020, Belgium, Proceedings 26*, pages 37–54. Springer, 2020. doi: 10.1007/978-3-030-58475-7_3.

[12] Jeremias Berg, Fahiem Bacchus, and Alex Poole. Abstract cores in implicit hitting set maxsat solving. In *Theory and Applications of Satisfiability Testing – SAT 2020*, pages 277–294. Springer, 2020. ISBN 978-3-030-51825-7. doi: 10.1007/978-3-030-51825-7_2 0.

[13] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009. ISBN 9781607503767. doi: 10.5555/1550723.

[14] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, 2009. IOS Press. ISBN 978-1-58603-929-5.

[15] Patrick Blackburn and Johan Bos. Representation and inference for natural language, 2005. ISSN 0717-6163.

[16] Patrick Blackburn and Johan Bos. Working with discourse representation theory. *An Advanced Course in Computational Semantics*, 2006.

[17] Ignace Bleukx, Jo Devriendt, Emilio Gamba, Bart Bogaerts, and Tias Guns. Simplifying Step-Wise Explanation Sequences. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:20, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-300-3. doi: 10.4230/LIPIcs.CP.2023.11.

[18] Bart Bogaerts, Emilio Gamba, Jens Claes, and Tias Guns. Step-wise explanations of constraint satisfaction problems. In *24th European Conference on Artificial Intelligence (ECAI)*, 2020. doi: 10.3233/FAIA200149.

[19] Bart Bogaerts, Emilio Gamba, and Tias Guns. A framework for step-wise explaining how to solve constraint satisfaction problems. *Artificial Intelligence*, 300:103550, 2021. ISSN 0004-3702. doi: 10.1016/j.artint.2021.103550.

[20] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *Journal of Artificial Intelligence Research*, 77:1539–1589, 2023. doi: doi.org/10.1613/jair.1.14296.

[21] Leonard A Breslow, David W Aha, et al. Simplifying decision trees: A survey. *Knowledge engineering review*, 12(1):1–40, 1997.

[22] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011. doi: 10.1145/2043174.2043195.

[23] Allan Caine and Robin Cohen. Mits: A mixed-initiative intelligent tutoring system for sudoku. In *Conference of the Canadian Society for Computational Studies of Intelligence*, pages 550–561. Springer, 2006. doi: 10.1007/11766247_47.

[24] Rocsildes Canoy, Jayanta Mandi, Victor Bucarey Lopez, Emilio Gamba, Maxime Mulamba, and Tias Guns. Tsp with learned zone preferences for last-mile vehicle dispatching. In *Amazon Last-Mile Routing Research Challenge*, pages XXX–1. MIT Libraries, 2021.

[25] Pierre Carbonnelle, Bram Aerts, Marjolein Deryck, Joost Vennekens, and Marc Denecker. An interactive consultant. In *Proceedings of the 31st Benelux Conference on Artificial Intelligence (BNAIC 2019) and the 28th Belgian Dutch Conference on Machine Learning (Benelearn 2019), Brussels, Belgium*, 2019.

[26] Broes De Cat, Bart Bogaerts, Maurice Bruynooghe, Gerda Janssens, and Marc Denecker. Predicate logic as a modeling language: the IDP system. In Michael Kifer and Yanhong Annie Liu, editors, *Declarative Logic Programming: Theory, Systems, and Applications*, pages 279–323. ACM / Morgan & Claypool, 2018. ISBN 978-1-97000-199-0. doi: 10.1145/3191315.3191321.

[27] Karthekeyan Chandrasekaran, Richard Karp, Erick Moreno-Centeno, and Santosh Vempala. Algorithms for implicit hitting set problems. In *Proceedings of the 2011 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 614–629, 2011. doi: 10.1137/1.9781611973082.48.

[28] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *CoRR*, abs/1603.02754, 2016.

[29] Jens Claes. Master thesis: Automatic translation of logic grid puzzles into a typed logic. Master's thesis, KU Leuven, Leuven, Belgium, June 2017.

[30] Jens Claes, Bart Bogaerts, Rocsildes Canoy, Emilio Gamba, and Tias Guns. Zebratutor: Explaining how to solve logic grid puzzles. In *Proceedings of the 31st Benelux Conference on Artificial Intelligence (BNAIC 2019) and the 28th Belgian Dutch Conference on Machine Learning (Benelearn 2019), Brussels, Belgium, November 6-8, 2019*, 2019. URL `http://ceur-ws.org/Vol-2491/demo96.pdf`.

[31] William F Clocksin and Christopher S Mellish. *Programming in PROLOG*. Springer Science & Business Media, 2003. doi: 10.1007/978-1-4471-5487-7.

[32] Vincent Conitzer. Eliciting single-peaked preferences using comparison queries. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–8, 2007. doi: 10.1145/1329125.1329204.

[33] Jessica Davies and Fahiem Bacchus. Solving maxsat by solving a sequence of simpler sat instances. In *Principles and Practice of Constraint Programming – CP 2011*, pages 225–239. Springer, 2011. ISBN 978-3-642-23786-7. doi: 10.1007/978-3-642-23786-7_19.

[34] Jessica Davies and Fahiem Bacchus. Exploiting the power of MIP solvers in MAXsat. In *Proceedings of SAT*, pages 166–181, 2013. doi: 10.1007/978-3-642-39071-5_13.

[35] Jessica Davies and Fahiem Bacchus. Postponing optimization to speed up MAXSAT solving. In *Proceedings of CP*, pages 247–262, 2013. doi: 10.1007/978-3-642-40627-0_21.

[36] Abir De and Soumen Chakrabarti. Neural estimation of submodular functions with applications to differentiable subset selection. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 19537–19552. Curran Associates, Inc., 2022.

[37] Maria Garcia De La Banda, Peter J Stuckey, Pascal Van Hentenryck, and Mark Wallace. The future of optimization technology. *Constraints*, 19:126–138, 2014. doi: 10.1007/s10601-013-9149-z.

[38] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24.

[39] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *Proceedings of SAT*, pages 36–41, 2006. doi: 10.1007/11814948_5.

[40] Sharmi Dev Gupta, Begum Genc, and Barry O'Sullivan. Explanation in constraint satisfaction: A survey. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 4400–4407. International Joint Conferences on Artificial Intelligence Organization, 8 2021. doi: 10.24963/ijcai.2021/601. Survey Track.

[41] Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved static symmetry breaking for sat. In *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 104–122. Springer, 2016. doi: 10.1007/978-3-319-40970-2_8.

[42] Herbert Enderton and Herbert B Enderton. *A mathematical introduction to logic*. Elsevier, 2001.

[43] Guillaume Escamocher and Barry O'Sullivan. Solving logic grid puzzles with an algorithm that imitates human behavior. *arXiv preprint arXiv:1910.06636*, 2019.

[44] Joan Espasa, Ian P. Gent, Ruth Hoffmann, Christopher Jefferson, and Alice M. Lynch. Using small muses to explain how to solve pen and paper puzzles. *ArXiv*, abs/2104.15040, 2021.

[45] Katalin Fazekas, Fahiem Bacchus, and Armin Biere. Implicit hitting set algorithms for maximum satisfiability modulo theories. In *Automated Reasoning*, pages 134–151. Springer, 2018. ISBN 978-3-319-94205-6. doi: 10.1007/978-3-319-94205-6_10.

[46] Alexander Felfernig, Lothar Hotz, Claire ONeill, and Juha Tiihonen. *Knowledge-Based Configuration: From Research to Business Cases*. Morgan Kaufmann, 05 2014. ISBN 978-0124158177.

[47] fet. Fetproact-eic-05-2019, fet proactive: emerging paradigms and communities, call, 05 2019. URL `https://ec.europa.eu/info/funding-tenders/opportunities/portal/screen/opportunities/topic-details/fetproact-eic-05-2019`.

[48] Thibaut Feydy and Peter J Stuckey. Lazy clause generation reengineered. In *International Conference on Principles and Practice of Constraint Programming*, pages 352–366. Springer, 2009. doi: 10.1007/978-3-642-04244-7_29.

[49] Maria Fox, Derek Long, and Daniele Magazzeni. Explainable planning. In *IJCAI'17 workshop on Explainable AI*, 2017. doi: 10.48550/arXiv.1709.10256.

[50] Eugene Freuder. In pursuit of the holy grail. *ACM Computing Surveys (CSUR)*, 28(4es): 63–es, 1996. doi: 10.1145/242224.242304.

[51] Eugene C Freuder, Chavalit Likitvivatanavong, and Richard J Wallace. Explanation and implication for configuration problems. In *IJCAI 2001 workshop on configuration*, pages 31–37, 2001.

[52] Eurgene Freuder. Pthg-19: The third workshop on progress towards the holy grail. `https://freuder.wordpress.com/pthg-19-the-third-workshop-on-progress-towards-the-holy-grail/`, 2019. [Accessed 25-04-2024].

[53] Alan M Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martinez-Hernandez, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008. doi: 10.1007/s10601-008-9047-y.

[54] Emilio Gamba, Bart Bogaerts, and Tias Guns. Efficiently explaining csps with unsatisfiable subset optimization. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence, IJCAI*, pages 1381–1388, 2021. doi: 10.24963/ijcai.2021/191.

[55] Emilio Gamba, Bart Bogaerts, and Jens Claes Tias Guns. A framework for step-wise explaining how to solve constraint satisfaction problems, jun 2021.

[56] Emilio Gamba, Bart Bogaerts, and Tias Guns. Efficiently explaining csps with unsatisfibale subset optimization. *Journal of Artificial Intelligence Research*, 78:709–746, 2023. doi: 10.1613/jair.1.14260.

[57] M. Ganesalingam and W. T. Gowers. A fully automatic theorem prover with human-style output. *Journal of Automated Reasoning*, 58(2):253–291, Feb 2017. ISSN 1573-0670. doi: 10.1007/s10817-016-9377-1.

[58] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. The conflict-driven answer set solver clasp: Progress report. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR*, volume 5753 of *Lecture Notes in Computer Science*, pages 509–514. Springer, 2009. ISBN 978-3-642-04237-9. doi: 10.1007/978-3-642-04238-6_50.

[59] Roman Gershman, Maya Koifman, and Ofer Strichman. An approach for extracting a small unsatisfiable core. *Formal Methods in System Design*, 33(1-3):1–27, 2008. doi: 10.1007/s10703-008-0051-z.

[60] Leilani H. Gilpin, David Bau, Ben Z. Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. Explaining explanations: An overview of interpretability of machine learning. In Francesco Bonchi, Foster J. Provost, Tina Eliassi-Rad, Wei Wang, Ciro Cattuto, and Rayid Ghani, editors, *5th IEEE International Conference on Data Science and Advanced Analytics, DSAA*, pages 80–89. IEEE, 2018. ISBN 978-1-5386-5090-5. doi: 10.1109/DSAA.2018.00018.

[61] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for cnf formulas. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, page 10886, USA, 2003. ISBN 0769518702. doi: 10.1109/DATE.2003.1253718.

[62] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. A survey of methods for explaining black box models. *ACM computing surveys (CSUR)*, 51(5):1–42, 2018. doi: 10.1145/3236009.

[63] David Gunning. Explainable artificial intelligence (xai). *Defense Advanced Research Projects Agency (DARPA), nd Web*, 2(2):1, 2017.

[64] Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation, Held with CP*, volume 19, 2019.

[65] Tias Guns, Emilio Gamba, Maxime Mulamba, Ignace Bleukx, Senne Berden, and Milan Pesa. Sudoku assistant – an ai-powered app to help solve pen-and-paper sudokus. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(13):16440–16442, Sep. 2023. doi: 10.1609/aaai.v37i13.27072.

[66] Shengbo Guo and Scott Sanner. Real-time multiattribute bayesian preference elicitation with pairwise comparison queries. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 289–296. JMLR Workshop and Conference Proceedings, 2010.

[67] Ronan Hamon, Henrik Junklewitz, and Ignacio Sanchez. Robustness and explainability of artificial intelligence. *JRC*, 2020. doi: 10.2760/57493.

[68] Pieter Van Hertum, Ingmar Dasseville, Gerda Janssens, and Marc Denecker. The KB paradigm and its application to interactive configuration. *TPLP*, 17(1):91–117, 2017. doi: 10.1017/S1471068416000156.

[69] Mireille Hildebrandt, Carlos Castillo, Elisa Celis, Salvatore Ruggieri, Linnet Taylor, and Gabriela Zanfir-Fortuna, editors. *Proceedings of FAT**, 2020.

[70] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*. John Wiley & Sons, 2013.

[71] Jinbo Huang. Mup: A minimal unsatisfiability prover. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, volume 1, pages 432– 437 Vol. 1, 02 2005. ISBN 0-7803-8736-8. doi: 10.1109/ASPDAC.2005.1466202.

[72] IBM. What is deep learning? URL `https://www.ibm.com/topics/deep-learning`.

[73] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996. doi: 10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P.

[74] Alexey Ignatiev, Alessandro Previti, Mark Liffiton, and Joao Marques-Silva. Smallest mus extraction with minimal hitting set dualization. In *Principles and Practice of Constraint Programming*, pages 173–182. Springer, 2015. doi: 10.1007/978-3-319-23219-5_13.

[75] Alexey Ignatiev, Mikolás Janota, and João Marques-Silva. Quantified maximum satisfiability. *Constraints*, 21(2):277–302, 2016. doi: 10.1007/s10601-015-9195-9.

[76] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018. doi: 10.1007/978-3-319-94144-8_26.

[77] Elgun Jabrayilzade and Selma Tekir. Lgpsolver-solving logic grid puzzles automatically. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1118–1123, 2020.

[78] Thorsten Joachims. Training linear svms in linear time. In Tina Eliassi-Rad, Lyle H. Ungar, Mark Craven, and Dimitrios Gunopulos, editors, *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*, pages 217–226. ACM, 2006. doi: 10.1145/1150402.1150429.

[79] M. I. Jordan and T. M. Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015. doi: 10.1126/science.aaa8415.

[80] Ulrich Junker. Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, 2001.

[81] Hans Kamp. Discourse representation theory: What it is and where it ought to go. In Albrecht Blaser, editor, *Natural Language at the Computer, Scientific Symposium on Syntax and Semantics for Text Processing and Man-Machine-Communication, Heidelberg, FRG*, volume 320 of *Lecture Notes in Computer Science*, pages 84–111. Springer, 1988. ISBN 3-540-50011-1. doi: 10.1007/3-540-50011-1_34.

[82] Daniel M Kane, Shachar Lovett, Shay Moran, and Jiapeng Zhang. Active classification with comparison queries. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 355–366. IEEE, 2017. doi: 10.1109/FOCS.2017.40.

[83] Hans Kleine Büning and Uwe Bubeck. Theory of quantified boolean formulas. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 735–760. IOS Press, 2009. ISBN 978-1-58603-929-5. doi: 10.3233/978-1-58603-929-5-735.

[84] Antonina Kolokolova, Yongmei Liu, David G. Mitchell, and Eugenia Ternovska. On the complexity of model expansion. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17*, volume 6397 of *Lecture Notes in Computer Science*, pages 447–458. Springer, 2010. ISBN 978-3-642-16241-1. doi: 10.1007/978-3-642-16242-8_32.

[85] Patrick Koopmann. Two ways of explaining negative entailments in description logics using abduction. *Workshop on Explainable Logic-Based Knowledge Representation (XLoKR 2021)*, 2021.

[86] Pat Langley, Ben Meadows, Mohan Sridharan, and Dongkyu Choi. Explainable agency for intelligent autonomous systems. In *Twenty-Ninth IAAI Conference*, volume 31, pages 4762–4763, 2017. doi: 10.1609/aaai.v31i2.19108.

[87] Niklas Lauffer and Ufuk Topcu. Human-understandable explanations of infeasibility for resource-constrained scheduling problems. In *ICAPS 2019 Workshop XAIP*, 2019.

[88] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553): 436–444, 2015. doi: 10.1038/nature14539.

[89] Kevin Leo and Guido Tack. Debugging unsatisfiable constraint models. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 77–93. Springer, 2017. doi: 10.1007/978-3-319-59776-8_7.

[90] Michael Leuschel and Michael Butler. Automatic refinement checking for b. In *International Conference on Formal Engineering Methods*, pages 345–359. Springer, 2005.

[91] Chu Min Li and Felip Manyà. MaxSAT, hard and soft constraints. In *Handbook of satisfiability*, pages 903–927. IOS Press, 2021. doi: 10.3233/978-1-58603-929-5-613.

[92] Beishui Liao and Leendert Van Der Torre. Explanation semantics for abstract argumentation. In *Computational Models of Argument*, pages 271–282. IOS Press, 2020. doi: 10.3233/FAIA200511.

[93] Mark H Liffiton and Ammar Malik. Enumerating infeasibility: Finding multiple muses quickly. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 160–175. Springer, 2013. doi: 10.1007/978-3-642-38171-3_11.

[94] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008. doi: 10.1007/s10817-007-9084-z.

[95] Mark H Liffiton, Alessandro Previti, Ammar Malik, and Joao Marques-Silva. Fast, flexible mus enumeration. *Constraints*, 21(2):223–250, 2016. doi: 10.1007/s10601-015-9183-0.

[96] Tie-Yan Liu et al. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009. doi: 10.1561/1500000016.

[97] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[98] Inês Lynce and João P. Marques Silva. On computing minimum unsatisfiable cores. In *Proceedings of SAT*, 2004.

[99] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[100] Joao Marques-Silva. Minimal unsatisfiability: Models, algorithms and applications. In *2010 40th IEEE International Symposium on Multiple-Valued Logic*, pages 9–14. IEEE, 2010. doi: 10.1109/ISMVL.2010.11.

[101] Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In *Handbook of satisfiability*, pages 131–153. ios Press, 2009. doi: 10.3233/978-1-58603-929-5-131.

[102] Joao Marques-Silva, Federico Heras, Mikolás Janota, Alessandro Previti, and Anton Belov. On computing minimal correction subsets. In *Proceedings of the 23rd international joint conference on Artificial IntelligenceAugust*, page 615–622, 2013. doi: 10.1007/978-3-319-09284-3_6.

[103] George Mavrotas. Effective implementation of the epsilon-constraint method in multi-objective mathematical programming problems. *Appl. Math. Comput.*, 213(2):455–465, 2009. doi: 10.1016/J.AMC.2009.03.037.

[104] Tim Miller. Explanation in artificial intelligence: Insights from the social sciences. *Artificial Intelligence*, 267:1–38, 2019. doi: 10.1016/j.artint.2018.07.007.

[105] David G Mitchell and Eugenia Ternovska. Expressive power and abstraction in essence. *Constraints*, 13(3):343–384, 2008.

[106] David G. Mitchell, Eugenia Ternovska, Faraz Hach, and Raheleh Mohebali. Model expansion as a framework for modelling and solving search problems. Technical Report TR 2006-24, Simon Fraser University, Canada, 2006.

[107] Arindam Mitra and Chitta Baral. Learning to automatically solve logic grid puzzles. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1023–1033, 2015.

[108] Sanjay Modgil, Francesca Toni, Floris Bex, Ivan Bratko, Carlos I Chesnevar, Wolfgang Dvořák, Marcelo A Falappa, Xiuyi Fan, Sarah Alice Gaggl, Alejandro J García, et al. The added value of argumentation. In *Agreement technologies*, pages 357–403. Springer, 2013. doi: 10.1007/978-94-007-5583-3_21.

[109] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007. doi: 10.1007/978-3-540-74970-7_38.

[110] Peter Nightingale, Özgür Akgün, Ian P Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in savile row. *Artificial Intelligence*, 251:35–61, 2017. doi: 10.1016/j.artint.2017.07.001.

[111] Andreas Niskanen, Jeremias Berg, and Matti Järvisalo. Enabling Incrementality in the Implicit Hitting Set Approach to MaxSAT Under Changing Weights. In *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 44:1–44:19. Schloss

Dagstuhl – Leibniz-Zentrum für Informatik, 2021. ISBN 978-3-95977-211-2. doi: 10.4230/LIPIcs.CP.2021.44.

[112] Mariam Obeid, Zeinab Obeid, Asma Moubaiddin, and Nadim Obeid. Using description logic and abox abduction to capture medical diagnosis. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pages 376–388. Springer, 2019. ISBN 978-3-030-22999-3.

[113] Yoonna Oh, Maher N. Mneimneh, Zaher S. Andraus, Karem A. Sakallah, and Igor L. Markov. AMUSE: a minimally-unsatisfiable subformula extractor. In *Proceedings of DAC*, pages 518–523, 2004. doi: 10.1145/996566.996710.

[114] Stephen Ostermiller. Qqwing–sudoku generator and solver, 2011. URL `https://qqwing.com/`.

[115] 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*, 12:2825–2830, 2011.

[116] Protection Regulation. Regulation (eu) 2016/679 of the european parliament and of the council. *REGULATION (EU)*, 679:2016, 2016.

[117] Raymond Reiter. A theory of diagnosis from first principles. *AIJ*, 32(1):57–95, 1987.

[118] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 1135–1144, New York, NY, USA, 2016. Association for Computing Machinery. doi: 10.1145/2939672.2939778.

[119] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006. ISBN 978-0-444-52726-4. doi: 10.5555/1207782.

[120] Stephen Ryder. *Puzzle Baron's logic puzzles*. Alpha Books, Indianapolis, Indiana, 2016. ISBN 9781465454652.

[121] Paul Saikko, Jeremias Berg, and Matti Järvisalo. Lmhs: a sat-ip hybrid maxsat solver. In *International conference on theory and applications of satisfiability testing*, pages 539–546. Springer, 2016.

[122] Paul Saikko, Johannes Peter Wallner, and Matti Järvisalo. Implicit hitting set algorithms for reasoning beyond NP. In *Proceedings of KR*, pages 104–113, 2016.

[123] Zeynep G Saribatur, Peter Schüller, and Thomas Eiter. Abstraction for non-ground answer set programs. In *European Conference on Logics in Artificial Intelligence*, pages 576–592. Springer, 2019.

[124] Bernhard Schölkopf, Alex J Smola, Robert C Williamson, and Peter L Bartlett. New support vector algorithms. *Neural computation*, 12(5):1207–1245, 2000.

[125] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of ICCV*, pages 618–626, 2017. doi: 10.1007/s11263-019-01228-7.

[126] Ilankaikone Senthooran, Matthias Klapperstueck, Gleb Belov, Tobias Czauderna, Kevin Leo, Mark Wallace, Michael Wybrow, and Maria Garcia de la Banda. Human-centred feasibility restoration in practice. *Constraints*, 28(2):203–243, 2023. doi: 10.1007/s10601-023-09344-5.

[127] Dunja Šešelja and Christian Straßer. Abstract argumentation and explanation applied to scientific debates. *Synthese*, 190(12):2195–2217, 2013. doi: 10.1007/s11229-011-9964-y.

[128] Burr Settles. Active learning literature survey. 2009.

[129] Mohammed H. Sqalli and Eugene C. Freuder. Inference-based constraint satisfaction supports explanation. In William J. Clancey and Daniel S. Weld, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 1*, pages 318–325. AAAI Press / The MIT Press, 1996. ISBN 0-262-51091-X. URL `http://www.aaai.org/Library/AAAI/1996/aaai96-048.php`.

[130] Daniele Magazzeni Tim Miller, Rosina Weber, editor. *Proceedings of the IJCAI 2019 Workshop on Explainable Artificial Intelligence*, 2019.

[131] Markus Ulbricht and Johannes P Wallner. Strong explanations in abstract argumentation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35 (7), pages 6496–6504, 2021. doi: 10.1609/aaai.v35i7.16805.

[132] Alexandros Vassiliades, Nick Bassiliades, and Theodore Patkos. Argumentation and explainable artificial intelligence: a survey. *The Knowledge Engineering Review*, 36, 2021. doi: 10.1017/S0269888921000011.

[133] Johan Wittocx, Marc Denecker, and Maurice Bruynooghe. Constraint propagation for first-order logic and inductive definitions. *ACM Trans. Comput. Log.*, 14, 2013.

[134] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML*, volume 97 of *Proceedings of Machine Learning Research*, pages 6984–6994. PMLR, 2019. URL `http://proceedings.mlr.press/v97/`.

[135] Kiana Zeighami, Kevin Leo, Guido Tack, and Maria Garcia de la Banda. Towards semi-automatic learning-based model transformation. In *Principles and Practice of Constraint Programming*, pages 403–419, 2018. doi: 10.1007/978-3-319-98334-9_27.