Solving Combinatorial Optimization Problems With Boolean Satisfiability Solvers

Hendrik 'Henk' Bierlee

Master of Science in *Computer Science*
Uppsala University in Uppsala, Sweden

Master of Engineering (Sino-Swedish dual degree)
National Taiwan Normal University in Taipei, Taiwan

Bachelor of Science in *Computer Science and Engineering*
Delft University of Technology in Delft, The Netherlands

A thesis submitted for the degree of **Doctor of Philosophy** at

Monash University in *2025*

Department of Data Science and AI, Faculty of Information Technology

Abstract

Combinatorial Optimization (CO) is the mathematical study of decision-making problems, which occur throughout nature and society. Examples of CO problems span a wide range of fields, including logistics (e.g. workforce rostering, vehicle routing), information technology (e.g. network routing, software compilation, hardware verification), ethics (e.g. resource allocation, voting audits), gaming (e.g. pathfinding), and so on. Often, engineers face two types of challenges when tackling CO problems. On the one hand, the problem's textbook definition does not cover the *real-world complexities* with its ever-changing requirements and infinite details. On the other hand, the *computational hardness* of even the textbook definition dictates that there currently is no known algorithm to solve it efficiently - and there likely never will be.

Constraint Programming (CP) is a programming paradigm which separates these two concerns. It allows the user to mathematically specify the real-world complexities in a *constraint model*. The model is then solved by powerful *solvers*. A translation layer between model and solver ensures that changes in the details of the problem only require changes in the model, but not in the solver.

This thesis studies the translation (or *encoding*) of models to a particular type of solver based on Boolean Satisfiability (SAT). The performance of these SAT solvers has dramatically improved in the last twenty years, standing out in solver competitions and industrial use cases. However, encoding a constraint model into SAT is uniquely difficult. Usually, multiple encodings of the same problem are possible, and one can be far more easy to solve than another, and there is no straightforward way to predict which is best.

The thesis progresses the field of SAT encodings in the following ways. We develop an encoding framework which automatically resolves the encoding of constraints over integer variables with *different* encodings. These mixed encodings lead to an improvement of various case studies. Second, certain encodings overlap with the field of hardware circuit design, as both often use a binary representation of the integer variables. By adapting these techniques from hardware to software, we show how its benefits carry over in the size of the encoding and its solving performance. Thirdly, existing SAT encoding methods of integer constraints use different abstractions which implicitly represent auxiliary integer variables. We unify these methods as a decomposition into explicit integer variables and constraints. As a result, we improve our theoretical understanding of existing methods, and we gain practical improvements by extending the decompositions in novel ways.

Declaration

This thesis is an original work of my research and contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, or any use of generative artificial intelligence technologies, except where due reference is made in the text of the thesis.

Signature: .....
Print Name: .....
Date: .....

Publications During Enrolment

The following works were published during the thesis project:

CPAIOR'22 H. Bierlee, G. Gange, G. Tack, J. J. Dekker, and P. J. Stuckey, "Coupling Different Integer Encodings for SAT," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 19th International Conference, CPAIOR 2022, Los Angeles, CA, USA, June 20-23, 2022, Proceedings*, P. Schaus, Ed., in Lecture Notes in Computer Science, vol. 13292. Springer, 2022, pp. 44-63. doi: 10.1007/978-3-031-08011-1_5.

CPAIOR'24 H. Bierlee, J. J. Dekker, V. Lagoon, P. J. Stuckey, and G. Tack, "Single Constant Multiplication for SAT," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 21st International Conference, CPAIOR 2024, Uppsala, Sweden, May 28-31, 2024, Proceedings, Part I*, B. Dilkina, Ed., in Lecture Notes in Computer Science, vol. 14742. Springer, 2024, pp. 84-98. doi: 10.1007/978-3-031-60597-0_6.

The following work was accepted to be published during the thesis project:

CPAIOR'25 H. Bierlee, Jip. J. Dekker, and P. J. Stuckey, "Revisiting Pseudo-Boolean Encodings from an Integer Perspective" in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 22nd International Conference, CPAIOR 2025, Melbourne, Victoria, Australia, November 10-13, 2025*

Acknowledgements

First and foremost, I would like to acknowledge and thank my supervisory team at Monash University. It has been an absolute privilege to have a day-to-day working relationship with my main supervisor, **Peter J. Stuckey**. Somehow, Peter always manages to distill something useful out of my ramblings. His tireless energy to engage with topics of such depth and breadth will never cease to amaze me. **Jip J. Dekker** has contributed a far beyond reasonable effort in the design and implementation of this project. In my eyes, Jip embodies the type of programmer which I aspire to be some day: principled, communicative, productive, sharp, and empathetic. **Guido Tack** has been an inspiration as a researcher. Guido's explanations are always insightful and clear, as well as patient and honest. Finally, I have taken much to heart from the bi-weekly meetings with **Graeme Gange** during the first year.

I recently realized that I could have done it all perfectly alone, however, I would not have come out the other side as a better person. This is why I now thank those who were closest to me during this chapter of my life. Here is to Adithi, who says change is the only constant, but for me the real constant was her companionship during lockdown. To Stas, for who the abyss will never dare look back. To Sam, who never stops watering the grass. And to 林兰 (*Summer*), for her never-ending affection and support.

I would like to thank my other close colleagues and friends, especially: Alexander (Alex), Emely, Kelvin, Kevin, Rehan, and Vera. I would like to thank the following past and current affiliates of the optimization group who have made for a truly inspiring work environment: Abdallah, Adam, Alexey, Allen, Andy, Bhagya, Bojie, Canchen, Daniel Bechaz, Daniel Harabor, Draga, Edward (Eddie), Ehsan, Frits, Gleb, Goldi, Gonzalo, Hao, Jaime, Jason, Jayden, Jinqiang, John, Jonathan, Kevin, Long, Maria, Mark Carlson, Mark Wallace, Markus, Piyumi, Ryan, Sameela, Senthoo, Shizhe, Simon, Somayeh, Sullivan, Sushmita, Terrence, Thomas (Tom), Yue, Zahra, and Zhe (Mike).

I would like to thank my enduring milestone panel: Prof. Graham Farr, Dr. Piere Le Bodic, and Dr. Buser Say. I thank the examiners, Prof. Neng-Fa Zhou and Dr. Peter Nightingale, for their thorough examination of this thesis. I thank Julie Holden for her helpful comments on my reports. I thank the many other great OPTIMA members at the University of Melbourne. I thank Dr. Vitaly Lagoon for his help as co-author on our CPAIOR'24 paper. I thank FIT Operations, for closing more than 150 support tickets.

I thank Prof. Pierre Flener, for introducing me to the giants whose shoulders I stand on, and for his warm welcome during my temporary return to Sweden. I thank my friends elsewhere: Erik, Kilian, Rens, Sebas, and Stan. Finally, I thank my councillor at Monash, Haylee Danaher, who has been a lifeline.

A final nod and farewell goes out to Prof. Ariel Liebman. Our OPTIMA debate is fondly remembered, especially that moment when you threw me a softball question which I only later understood was intended to lift me up.

Most dearly would I like to finally express my deep gratitude and admiration for my own family, who are far away in my home country: my father, Henk, and my mother, Jet. I cannot put into words - and neither have I tried to do this often enough - how important your love and unwavering support has been during my years abroad. My deepest thanks also to my uncle, Edwin, for always bringing life to every Christmas Eve. Finally, my heart goes out to my grandmother, Sonja, who I will mention on the next page.

No generative Artificial Intelligence (AI) technology was used in this thesis or its research.

This research was supported by a scholarship from the Faculty of Information Technology, Monash University.

This research was partially funded by the Australian Government through the Australian Research Council Industrial Transformation Training Centre in Optimization Technologies, Integrated Methodologies, and Applications (OPTIMA), Project ID IC200100009.

This work is dedicated to my grandmother, *Oma Sonja*, whom I miss dearly.

Summary for the general public

In combinatorial optimization problems, we make decisions that satisfy constraints and optimize objectives. An example is the hospital staff rostering problem, where we decide which staff members take which shifts. Rosters cannot violate policy constraints, but should minimize costs and maximize quality of care.

We can specify the decisions, constraints, and objective in a so-called constraint model. A model is translated for and then solved by some solving algorithm. Boolean Satisfiability (SAT) solvers are extremely efficient, however, the translation of the model is uniquely challenging. This thesis improves the theoretical understanding and practical effectiveness of the translation process for SAT solvers.

Amendments

The following is a summary of the amendments which were made between the draft and final submission after examination.

From the examiner report of Prof. *Neng-Fa Zhou*, City University of New York:

- Expand discussion of BEE and equi-propagation (pg. 80)
- Discuss limitation due to the lack of heuristic for encoding decisions (pg. 88)
- Add reference for the standard full/half adder encoding (pg. 91)
- Improve discussion of the MBKP benchmark for SCM (pg. 103)
- Discuss combining SCM approaches and improving baseline for large domains (pg. 106)
- Discuss improvements for the experiments (pg. 134)
- Process all typographical comments

From the examiner report of Dr. *Peter Nightingale*, University of York

- Do not characterize CP as being on the highest level (pg. 6)
- Clarify and contextualize the term "SAT-based solvers" (pg. 9)
- Remove a line on shared Boolean variables between integer encodings (pg. 12)
- Fix constraint and propagation defintion w.r.t real-valued solutions (pg. 16)
- Narrow down definition of CP (pg. 19)
- Include Plaisted-Greenbaum transformation (pg. 31)
- Clarify existing work on binary encoding existing (pg. 37)
- Fix conjunction to disjunction in extensional CSP example (pg. 42)
- Fix reference to bounds consistency, missing `CUTOFF` parameter value, and `table-layout` results discussion (Chp. 3)
- Clarify the Multidimensional Bounded Knapsack Problem (MBKP) specification (pg. 103)
- Clarify the encoder configurations in Section 5.3 (pg. 130)
- Add mixed-radix encodings to future work for the integer viewpoint (pg. 134)
- Process all typographical comments

Further changes:

- Add mixed encoding description and experiments following CPAIOR'25 submission (Chp. 5)
- Final typographical changes from proofreading

Contents

’

— Chapter 1 —

Introduction

In society, decisions are made all the time. Whether we are scheduling, planning, managing supply chains, creating work rosters for the hospital staff, designing hardware or compiling software, better decisions are always desirable. When a problem, or part of it, has a mathematical description, we can use computers to search for the best decision. This field of study is known as Combinatorial Optimization (CO): to find an object (e.g. work roster) from a huge set of objects (e.g. incorrect or sub-optimal rosters). The set is finite only in principal, but almost infinite in practice.

In a nurse rostering setting, a better roster in accordance with workplace policy, fairly distributed working hours, and expertise can save time, money, and even lives. One might reasonably think that computing a work roster for a few hundred hospital staff members is easily done by either smart software or powerful hardware. After all, nowadays we never have to wait for some everyday computing tasks such as when requesting the fastest route to our destination in a navigation app. Unfortunately, an important class of real-world problems seems fundamentally cursed by a computational complexity which grows exponentially with regards to its size. That is, not only does the number of potential rosters grow exponentially with every additional variable (e.g. staff member or work shift), but the best known algorithm might have to check all potential rosters in the worst case. Roughly speaking, to roster 1 additional staff member to 100 existing members does not add 1% to the runtime, but might easily multiply it by a large factor. Sooner rather than later, any hard problem inevitably becomes intractable.

The good news is that advances in software (i.e. algorithms) in combination with powerful hardware can postpone the inevitable, allowing us to still tackle realistically sized problems. Arguably the purest example of this can be found in the study of Boolean Satisfiability (SAT), a sub-field of CO. The SAT problem is *the* prototypical hard CO problem. Remarkably, SAT solvers, which are relatively small programs in size but great in sophistication, are defying expectations by solving industrial instances in the order of *millions* of variables. A good idea, then, is to leverage SAT solvers by translating (or *encoding*) real-world problems as SAT problems. In doing so, progress on the efficiency of SAT solvers directly transfers to any problem which can be encoded as a SAT problem.

However, encoding complex problems in the limited language of SAT is challenging. A CO problem of integers variables and constraints (e.g. "nurse number i is rostered for shift number j") needs to be encoded into a SAT problem of only Boolean variables, i.e. true/false decisions. Furthermore, the manner of translation heavily impacts the effectiveness of the solver. Even the best SAT solver, it turns out, cannot solve a badly-translated problem. Thus, it is important to improve the theoretical understanding and practical effectiveness of SAT encoding methods, which is the main aim of this thesis.

The rest of this introductory chapter is organized as follows. In Sec. 1.1, we contextualize our problem domain of CO, the Constraint Programming (CP) paradigm to solving it, and the role of SAT therein. In Sec. 1.2, we motivate the research by answering why this topic is both useful and challenging. In Sec. 1.3, we outline the chapters and their contributions to knowledge.

1.1 Problem domain

A CO problem consists of *decisions* and *constraints*. Let us illustrate these concepts using a classic CO problem: the Graph Colouring Problem (GCP). We will define its map colouring variant (i.e. GCP for planar graphs). On a map with n regions, we are to assign each region to a colour chosen from a set of at most k colours. For example, take the twelve provinces of the Netherlands (see Fig. 1.1) as the regions in question. The decisions are which colour to give to which region. The constraints are that no adjacent (bordering) regions can share a colour.

1.1.1 Uncoloured 1.1.2 Partial colouring 1.1.3 Incorrect colouring

Fig. 1.1: The twelve provinces of the Netherlands as an $n = 12, k = 3$ instance of GCP.[1] None of the islands are independent provinces.

In Sec. 1.1.1, we will think of the complexity of GCP from a combinatorial and computational viewpoint. In Sec. 1.1.2, we will view it from an engineer's perspective tasked with solving the problem in an efficient but flexible manner. In Sec. 1.1.3, we will introduce how SAT solvers can be used to finally solve the problem.

[1] SVG file sourced from https://simplemaps.com (accessed November 2024)

1.1.1 Computational complexity of GCP

How many computational steps are required to find a correct colouring of a particular map? Of course, this depends on the details (or *parameters*) of the map (or *instance*). The parameters are the number of regions n, the number of colours k, and information on which region is adjacent to which. In the instance depicted in Fig. 1.1, $n = 12$ and $k = 3$. Without constraints, there are k^n possible colourings, since there are k colours for the first region, and for each of those choices, there are k colours again for the second region, and so on: k multiplied by itself twelve times. This full set of potential colourings is known as the *search space* of the problem. It is easy to see that adding a region will increase the size of the search space by a factor of k. In Tab. 1.1, the size of the search space is shown for $k = 3$ and $k = 4$ for a growing number of regions n, as well as the runtime of modern hardware to check all colourings for correctness.

n	3^n	runtime	4^n	runtime
1	3	0 s	4	0 s
2	9	0 s	16	0 s
3	27	0 s	64	0 s
4	81	0 s	256	0 s
5	243	0 s	1,024	0 s
6	729	0 s	4,096	0 s
7	2,187	0 s	16,384	0 s
8	6,561	0 s	65,536	0 s
9	19,683	0 s	262,144	0 s
10	59,049	0 s	1,048,576	0 s
11	177,147	0 s	4,194,304	0 s
12	531,441	0 s	16,777,216	0 s
15	14,348,907	0 s	1,073,741,824	0 s
20	3,486,784,401	0 s	1,099,511,627,776	0.5 s
25	847,288,609,443	0.4 s	1,125,899,906,842,624	8 m
30	205,891,132,094,649	1.5 m	1,152,921,504,606,846,976	5.7 d
50	7.2×10^{23}	9655 y	1.3×10^{30}	1.7×10^{10} y

Tab. 1.1: The number of potential 3 and 4-colourings of n regions, and an (under)estimation of the runtime in seconds (s), minutes (m), hours (h), days (d), or years (y) of a brute-force algorithm based on $2{,}356{,}230 \times 10^6$ instructions per second using a state-of-the-art 64 core CPU [1]

We can see from Tab. 1.1 that even for such few colours, the number of possible colourings shoots up dramatically with each region added. Checking 3^{12} colourings of the provinces of the Netherlands can still be done instantly by modern hardware (< 0.1 s). However, if we attempt to colour a map of the 50 United States, we reach the end of what any present or future

hardware would hope to achieve within the limitations of the universe (or our patience). If we had an efficient algorithm, the large search space would not be a problem. However, it has been proven that GCP is NP-hard [2], which means there is no currently known algorithm that is not exponential in n.

1.1.2 Engineering complexity of GCP

A second type of complexity concerns how human engineers can understand and solve GCP. The idea of *abstraction* is important to this. From a mathematical viewpoint, a map is better represented by a *graph*, regions by *vertices*, and adjacencies by *edges* (see Fig. 1.2). The abstraction removes information which is irrelevant to the graph colouring problem, such as the shape of each region. Rather than region and colour names, we can use the natural numbers $1, 2, ...$ This allows us to state the constraints more mathematically: assign vertices $1, ..., n$ to colours $x_1, ..., x_n$ from $1, ..., k$ such that $x_i \neq x_j$ iff there is an edge between vertices i and j.

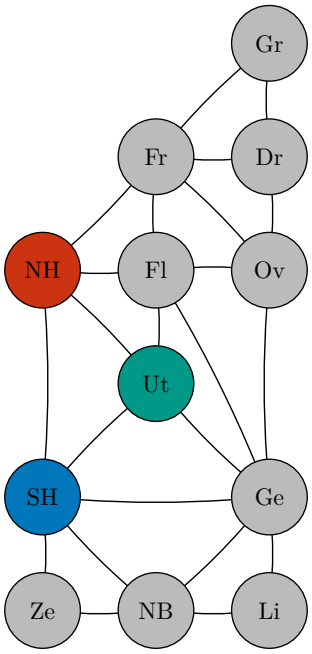


Fig. 1.2: A graph representation of Fig. 1.1.2

It remains to develop an algorithm to solve the instance. Many specialized algorithms for GCP and its variants exist. But implementing and maintaining these algorithms is difficult, and they often do not fit the full complexity of the problem specification. Furthermore, if the problem meaningfully changes, then the algorithm needs to be changed as well. For instance, if we add a new type of constraint which limits the number of times certain colours can be used, then the output of the algorithm will be incorrect. Mending this will require significant changes to the internals of the algorithm.

A different approach has been to separate the problem specification from its solving algorithm. In CP, a *constraint modelling language* such as MiniZinc [3], allows the user (or *modeller*) to

program (or *model*) a CO problem. A MiniZinc model of our instance is shown in Lst. 1.1. First, it declares the (known) number of colours k as a parameter `k=3`. Then, *decision variables* model the (unknown) colour assignments (i.e. $x_1 =$ `Dr`, $x_2 =$ `Fl`, ..., $x_{12} =$ `Ze`). Each variable has an associated *domain* which dictates its allowed assignments (e.g. x_i can only take an integer value from $1, ..., k$). MiniZinc declares a variable using the `var` keyword, followed by a variable name and domain (e.g. `var 1..k: Dr;` has domain `1..k` and name `Dr`). Finally, *constraints* enforce relationships between variables (e.g. the disequality constraint $x_i \neq x_j$ for adjacent vertices i and j). A constraint is added to MiniZinc by using the `constraint` keyword for every edge (e.g. `constraint Dr ≠ Fr;`).

```
int: k = 3;          var 1..k: Ov;        constraint Fl ≠ NH;  constraint Ge ≠ Ut;
var 1..k: Dr;        var 1..k: SH;        constraint Fl ≠ Ov;  constraint Ge ≠ SH;
var 1..k: Fl;        var 1..k: Ut;        constraint Fl ≠ Ut;  constraint Li ≠ NB;
var 1..k: Fr;        var 1..k: Ze;        constraint Fr ≠ Gr;  constraint NB ≠ SH;
var 1..k: Ge;        constraint Dr ≠ Fr;  constraint Fr ≠ NH;  constraint NB ≠ Ze;
var 1..k: Gr;        constraint Dr ≠ Gr;  constraint Fr ≠ Ov;  constraint NH ≠ SH;
var 1..k: Li;        constraint Dr ≠ Ov;  constraint Ge ≠ Li;  constraint NH ≠ Ut;
var 1..k: NB;        constraint Fl ≠ Fr;  constraint Ge ≠ NB;  constraint SH ≠ Ut;
var 1..k: NH;        constraint Fl ≠ Ge;  constraint Ge ≠ Ov;  constraint SH ≠ Ze;
```

Lst. 1.1: A MiniZinc model of the GCP instance from Fig. 1.2

Once specified, the model is translated automatically into input for a variety of powerful solving algorithms, and solver output is translated back into model solutions. The modeller does not need to know how the solver works. Solving Lst. 1.1 for $k = 3$ colours reveals that *none* of the 531,441 colourings satisfy all constraints. Changing to $k = 4$ colours yields one of the 8,832 solutions, shown in Fig. 1.3.

Fig. 1.3: A solution for $k = 4$.

The CP approach has various advantages. The abstraction is at a high level, namely a mathematical description of the problem, without losing information. We can try out various solving technologies and simply choose the best for the problem. A dedicated algorithm for GCP might outperform the solvers, however, if the problem changes, it would add to the engineering complexity to change that algorithm as well.

1.1.3 Solving CP with SAT

How can an algorithm solve a model in its generality, while guaranteeing properties such as correctness (soundness) and completeness? Actually, multiple approaches exist, based on solving technologies such as constraint solvers, Mixed-Integer Linear Programming (MIP), SAT, local search, and others. Each has different characteristics, strengths, and limitations.

Approaches based on SAT have become especially effective. In a SAT problem, we are to decide the assignment of Boolean variables (i.e. decision variables with a true/false domain) such that Boolean formula (i.e. a set of logical statements) holds. Each logical statement is a *clause*, which is an OR-statement over *literals*. A literal is either a Boolean variable b or its negation $\neg b$ (i.e. "not b", which flips the assignment of b from yes to no, and v.v.). An example Boolean formula over Boolean variables a and b is $(a \vee b) \wedge (\neg a \vee \neg b)$ (i.e. "a or b and not a or not b"). This formula encodes a higher-level Boolean constraint, namely the exclusive-or constraint $a \oplus b$. Despite its simplicity, SAT is NP-hard, and adding another variable doubles the search space [4]. Yet, SAT solvers are very successful at solving large and complex problems.

The challenge is that our communication with the SAT solver is limited to clauses. For the GCP this means that we *cannot* ask "what is the colour of region i?" (as we did in our CP model in Lst. 1.1 with the decision variable `var 1..k: Dr;`). Instead, we can only ask: "is the

colour of region i coloured orange, yes or no?" by introducing a Boolean variable for region i for the colour orange. To enforce the high-level integer constraints (e.g. disequality $x_i \neq x_j$), we introduce a clause expressing "region i is not coloured orange, or adjacent region j is not coloured orange". Then, we have to add one clause for every other colour as well.

This concept allows us to build a full SAT encoding of GCP. In the CP version, we represented the colour of a region by integer variables $x_1, ..., x_n$, each taking a value from its domain (i.e. the colours $1, ..., k$). Instead, in a SAT encoding we can represent each integer variable x_i with multiple Boolean variables, one for each colour c: $b_{i,c}, 1 \leq i \leq n, 1 \leq c \leq k$. The idea is to let the SAT solver assign $b_{i,c}$ to true iff vertex i takes colour c in a GCP solution (i.e., iff $x_i = c$). If $b_{i,c}$ is assigned to true, the answer to the question "is the colour of region i coloured c, yes or no?" is "yes", otherwise it is "no".

For every edge between two adjacent vertices i and j, we prevent the vertices from taking the same colour by adding clauses $\neg b_{i,c} \vee \neg b_{j,c}, 1 \leq c \leq k$ (i.e. "region i is not coloured c, or adjacent region j is not coloured c"). There are 23 edges and 3 colours, so this yields 69 clauses. At this point, the SAT solver can solve the problem trivially by assigning all $b_{i,c}$ to `false` (i.e. assigning no colours). To prevent this, we ensure at least one colour is selected for each region i by adding the clause $b_{i,1} \vee \cdots \vee b_{i,k}$. For illustrative purposes, the full encoding is shown in Ex. 1.1.

Example 1.1 A SAT encoding of the GCP problem from the instance in Fig. 1.2:

$$
\begin{aligned}
&(b_{1,1} \vee b_{1,2} \vee b_{1,3}) \wedge (b_{2,1} \vee b_{2,2} \vee b_{2,3}) \wedge (b_{3,1} \vee b_{3,2} \vee b_{3,3}) \wedge (b_{4,1} \vee b_{4,2} \vee b_{4,3}) \\
&\wedge (b_{5,1} \vee b_{5,2} \vee b_{5,3}) \wedge (b_{6,1} \vee b_{6,2} \vee b_{6,3}) \wedge (b_{7,1} \vee b_{7,2} \vee b_{7,3}) \wedge (b_{8,1} \vee b_{8,2} \vee b_{8,3}) \\
&\wedge (b_{9,1} \vee b_{9,2} \vee b_{9,3}) \wedge (b_{10,1} \vee b_{10,2} \vee b_{10,3}) \wedge (b_{11,1} \vee b_{11,2} \vee b_{11,3}) \wedge (b_{12,1} \vee b_{12,2} \vee b_{12,3}) \\
&\wedge (\neg b_{1,1} \vee \neg b_{3,1}) \wedge (\neg b_{1,2} \vee \neg b_{3,2}) \wedge (\neg b_{1,3} \vee \neg b_{3,3}) \wedge (\neg b_{1,1} \vee \neg b_{5,1}) \\
&\wedge (\neg b_{1,2} \vee \neg b_{5,2}) \wedge (\neg b_{1,3} \vee \neg b_{5,3}) \wedge (\neg b_{1,1} \vee \neg b_{9,1}) \wedge (\neg b_{1,2} \vee \neg b_{9,2}) \\
&\wedge (\neg b_{1,3} \vee \neg b_{9,3}) \wedge (\neg b_{2,1} \vee \neg b_{3,1}) \wedge (\neg b_{2,2} \vee \neg b_{3,2}) \wedge (\neg b_{2,3} \vee \neg b_{3,3}) \\
&\wedge (\neg b_{2,1} \vee \neg b_{4,1}) \wedge (\neg b_{2,2} \vee \neg b_{4,2}) \wedge (\neg b_{2,3} \vee \neg b_{4,3}) \wedge (\neg b_{2,1} \vee \neg b_{8,1}) \\
&\wedge (\neg b_{2,2} \vee \neg b_{8,2}) \wedge (\neg b_{2,3} \vee \neg b_{8,3}) \wedge (\neg b_{2,1} \vee \neg b_{9,1}) \wedge (\neg b_{2,2} \vee \neg b_{9,2}) \\
&\wedge (\neg b_{2,3} \vee \neg b_{9,3}) \wedge (\neg b_{2,1} \vee \neg b_{11,1}) \wedge (\neg b_{2,2} \vee \neg b_{11,2}) \wedge (\neg b_{2,3} \vee \neg b_{11,3}) \\
&\wedge (\neg b_{3,1} \vee \neg b_{5,1}) \wedge (\neg b_{3,2} \vee \neg b_{5,2}) \wedge (\neg b_{3,3} \vee \neg b_{5,3}) \wedge (\neg b_{3,1} \vee \neg b_{8,1}) \\
&\wedge (\neg b_{3,2} \vee \neg b_{8,2}) \wedge (\neg b_{3,3} \vee \neg b_{8,3}) \wedge (\neg b_{3,1} \vee \neg b_{9,1}) \wedge (\neg b_{3,2} \vee \neg b_{9,2}) \\
&\wedge (\neg b_{3,3} \vee \neg b_{9,3}) \wedge (\neg b_{4,1} \vee \neg b_{6,1}) \wedge (\neg b_{4,2} \vee \neg b_{6,2}) \wedge (\neg b_{4,3} \vee \neg b_{6,3}) \\
&\wedge (\neg b_{4,1} \vee \neg b_{7,1}) \wedge (\neg b_{4,2} \vee \neg b_{7,2}) \wedge (\neg b_{4,3} \vee \neg b_{7,3}) \wedge (\neg b_{4,1} \vee \neg b_{9,1}) \\
&\wedge (\neg b_{4,2} \vee \neg b_{9,2}) \wedge (\neg b_{4,3} \vee \neg b_{9,3}) \wedge (\neg b_{4,1} \vee \neg b_{10,1}) \wedge (\neg b_{4,2} \vee \neg b_{10,2}) \\
&\wedge (\neg b_{4,3} \vee \neg b_{10,3}) \wedge (\neg b_{4,1} \vee \neg b_{11,1}) \wedge (\neg b_{4,2} \vee \neg b_{11,2}) \wedge (\neg b_{4,3} \vee \neg b_{11,3}) \\
&\wedge (\neg b_{6,1} \vee \neg b_{7,1}) \wedge (\neg b_{6,2} \vee \neg b_{7,2}) \wedge (\neg b_{6,3} \vee \neg b_{7,3}) \wedge (\neg b_{7,1} \vee \neg b_{10,1}) \\
&\wedge (\neg b_{7,2} \vee \neg b_{10,2}) \wedge (\neg b_{7,3} \vee \neg b_{10,3}) \wedge (\neg b_{7,1} \vee \neg b_{12,1}) \wedge (\neg b_{7,2} \vee \neg b_{12,2}) \\
&\wedge (\neg b_{7,3} \vee \neg b_{12,3}) \wedge (\neg b_{8,1} \vee \neg b_{10,1}) \wedge (\neg b_{8,2} \vee \neg b_{10,2}) \wedge (\neg b_{8,3} \vee \neg b_{10,3}) \\
&\wedge (\neg b_{8,1} \vee \neg b_{11,1}) \wedge (\neg b_{8,2} \vee \neg b_{11,2}) \wedge (\neg b_{8,3} \vee \neg b_{11,3}) \wedge (\neg b_{10,1} \vee \neg b_{11,1}) \\
&\wedge (\neg b_{10,2} \vee \neg b_{11,2}) \wedge (\neg b_{10,3} \vee \neg b_{11,3}) \wedge (\neg b_{10,1} \vee \neg b_{12,1}) \wedge (\neg b_{10,2} \vee \neg b_{12,2}) \\
&\wedge (\neg b_{10,3} \vee \neg b_{12,3})
\end{aligned}
$$

The first clause $b_{1,1} \vee b_{1,2} \vee b_{1,3}$ enforces vertex 1 to take either colour 1, 2 or 3. After twelve such clauses, the first edge for the first colour is enforced by the clause $\neg b_{1,1} \vee \neg b_{3,1}$ ensuring vertex 1 and 3 do not both take colour 1. We show the partial assignment of Fig. 1.1.2 by highlighting variables assigned to true, i.e. $b_{8,1}$, $b_{10,2}$ and $b_{11,3}$, and their negation in red. A SAT solver can detect that this assignment does not lead to a solution in linear time.

1.2 Motivation

We now have seen the challenge and applicability of CO problems. We have seen how a CO problem can be modelled in CP and solved using various technologies. And we have seen how SAT solvers operate using a more limited language than CP. A vital part of this pipeline is the encoding process between CP and SAT. As such, the understanding, improvement and evaluation of SAT encodings will be the focus of this thesis. We motivate this goal by answering the questions why solving CP with SAT is useful in Sec. 1.2.1, and why it is challenging in Sec. 1.2.2.

1.2.1 Why is solving CP with SAT useful?

The main motivation to solve CO problems is because of their real-world applicability. Formulating the problem as a CP model is a versatile yet exact way of expressing the CO problem. The main reason to encode the CP model so that SAT solvers can solve the model is because it is often the most efficient solving approach.

The most compelling evidence for this is the MiniZinc Challenge, a solver competition. Some of the latest results are shown in Tab. 1.2. The SAT-based solvers (those which include SAT solvers as a crucial step of their solving algorithm) are consistently outperforming other approaches [5], [6]. In fact, Google's OR-Tools/CP-SAT (`cp-sat`) [7] solver has been the top-ranked solver in the free and open categories for the last eight years. While for `cp-sat` and Chuffed (`chuffed`) the SAT is part of a larger algorithm, Picat-SAT (`picat-sat`) attains second place by using a SAT solver directly (as described in Sec. 1.1.3).

Rank	Solver	Score
#1	**`cp-sat`**	1,557.75
#2	**`picat-sat`**	1,479.17
#3	**`chuffed`**	1,478.11
#4	`gurobi`	1,286.07
#5	`cplex`	1,258.59
#6	`izplus`	1,159.11
#7	`cp-optimizer`	1,137.28
#8	`choco-solver_cp`	1,128.72

Tab. 1.2: Top 8 of the free category (including 21 solvers) of the 2024 MiniZinc Challenge.[2] Solvers based on SAT (as far as we know) are shown in bold.

Second, even though the SAT problem is one of the most studied problems in CO, the rate of improvement of SAT solvers has not slowed down. The historical best SAT solvers from the last 30 years have been recompiled, and tested on the same benchmarks and hardware. In Fig. 1.4, we can see progress which can be only be due to improvements in SAT solving algorithms.

Next, SAT solvers can be used on useful variants of SAT. These include complete and incomplete algorithmic approaches, optimization with Maximum Satisfiability (MaxSAT), exact and approximate model (solution) counting [8], and support for proof-logging (which can rule out bugs in the SAT solving process itself). Arguably, this has been a contributing factor of the adoption of SAT in many industrial applications, such as in software and hardware verification, planning, scheduling and many more [9].

[2] https://www.minizinc.org/challenge/2024/results (accessed November 2024)

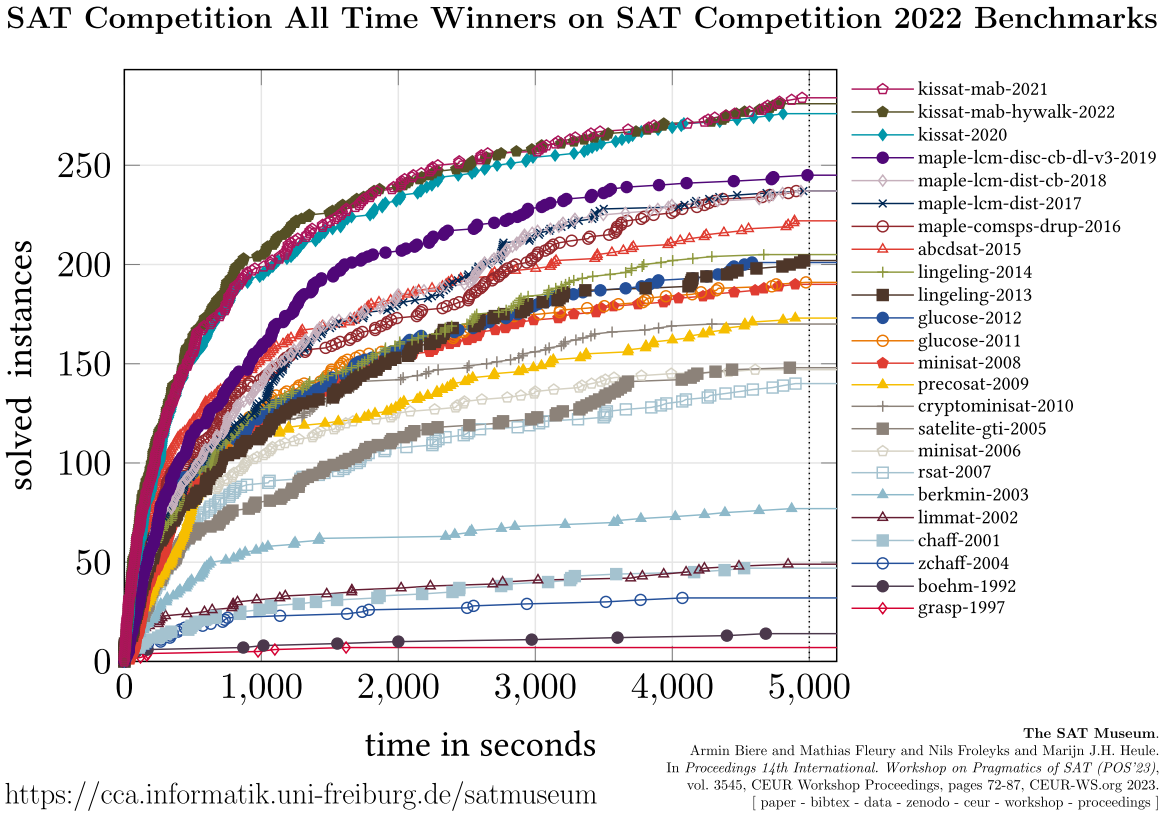


Fig. 1.4: Cactus plot of past SAT competition winners and other historical solvers [10].

We can circle back to the GCP problem to find an example use case of SAT solving which perhaps most clearly demonstrates its power and applicability. How many colours k are needed to colour any two-dimensional map? From Fig. 1.3, we can establish a lower bound of $k \geq 4$ by example. In 1890, an upper bound of $k \leq 5$ was definitively proven [11]. It was only in 1976 when the problem was closed when the four colour theorem (i.e. $k \leq 4$) was proven [12] [13]. Somewhat controversially, the proof was the first to heavily rely on computer assistance. A particular set of 1,834 sub-problems was checked to be four-colourable, which then implied all maps are. The computer-generated portion of the proof requires over 400 pages.

A generalization of GCP is the Packing Colouring Problem (PCP). Consider two vertices with the same colour $1 \leq c \leq k$. PCP requires the shortest path distance d between these vertices to be $d > c$ edges - a natural extension of GCP where $d = 1$ regardless of colour. The original motivation behind the PCP variant was the allocation of radio tower frequencies, as signals which are close in frequency and tower location interfere with each other. Since higher frequencies travel further, they must be more spread out.

In 2002, a particular instance of PCP was proposed. The objective was to find an upper bound on k for an imaginary infinite square grid (with orthogonal adjacencies, but no diagonal). This instance remained unsolved until 2022, when [14] established $k \leq 15$ by solving over 5 million

PCP sub-problems, running 128 SAT solvers in parallel for a total of 4851 CPU hours. The three preceding improvements on this problem were also achieved using SAT solving. The computer-generated portion of the proof requires 34 terabytes, *after* compression.[3] Improvements of the encoding, the topic of this thesis, were instrumental to achieve tractable solve times of the sub-problems for the final proof.

1.2.2 Why is solving CP with SAT challenging?

We now identify several challenges which appear when we go beyond the basic SAT encoding of the GCP problem in Sec. 1.1.3:

The same CO problem can be encoded in different ways At least three encodings of GCP are surveyed in the literature [15]. Yet, a particular constraint can have different encodings as well. A prime example is the so-called At-Most-One (`AMO`) constraint, which has at least seven possible encodings [16]. Some of these encodings can be further fine-tuned to change their behaviour. A difference in the encoding can make a significant difference in the performance of the SAT solver.

Changing the encoding of an integer variable requires changes the encoding of its constraints As an extension of the previous point, the encoding of any single integer variable is of even greater consequence, since it changes other parts of the encoding as well. This is unlike the constraint encoding, which is self-contained and can be swapped out. However, changing the integer encoding changes how each $b_{i,c}$ relates to x_i. For instance, while our so-called *direct* encoding of GCP reasons about the equality $x_i = c$, the *order* encoding instead reasons about the inequality $x_i \geq c$, and the *binary* encoding about a binary representation of x_i.

Consequently, when changing the integer variable encoding of a particular x_i, we also need to change the encoding of all constraints that x_i participates in (e.g. the disequality $x_i \neq x_j$ for an adjacent vertex j). Furthermore, if we allow x_j to have a different encoding from x_i, we have to encode a constraint over *mixed integer variable encodings*. In existing SAT approaches, the choice of integer variable encoding is often fixed to avoid the issue.

SAT encodings are hard to understand SAT reasons at a low level of abstraction compared to CP, since we reason about individual Boolean variables $b_{i,c}$ rather than high-level integer variables x_i. One integer variable becomes many Boolean variables (one for each domain

[3] Often, the computer-generated proof is not intended to be read and verified by humans. Instead, while it is generated using complicated algorithms (e.g. the SAT solver), it is intended to be verified by a simple verification algorithms. If we trust that the verification algorithm is correct, which should be easier to manually check than the complicated solver, then we can trust that the solver's proof output is correct for the given instance. Consequently, the proof of $k \leq 15$ for the infinite square grid instance of GCP is in this fundamental sense more trustworthy than the original proofs of the five and four colour theorems for GCP. And in fact, both the five and four color theorem proofs were corrected in the years following their publication, and have been verified in recent years.

value in the direct encoding of GCP), and one constraint becomes many clauses. The final SAT encoding (such as the one shown in Ex. 1.1) is inscrutable even for experts.

Of course, when implementing solving algorithms for CP, or when constructing models with more complex domains or constraints, experts similarly need to reason about individual domain values of the integer variables. However, there is another indirection in the case of SAT encodings. Depending on the integer variable's encoding, the Boolean variable relates to a specific property (e.g. a bit in a binary representation) of the integer variable's assignment, rather than some domain value directly.

Another side is that many existing encodings implicitly share common techniques, but this pattern is obscured by their individual abstractions and low-level specifications. These qualities make SAT uniquely challenging to reason about in theory and debug in practice.

The performance of a SAT solver is hard to predict Even though SAT experts can experimentally improve their encoding, it is hard to predict the behaviour of the solver. Sometimes, common sense applies, such as that a smaller encoding is easier to solve. But sometimes redundancy can be helpful. Other times, theoretical properties such as their propagation strength (relating to how much information can be inferred from a partial assignment of the encoding) is outweighed by a smaller encoding, while at other times it is not. Furthermore, the type of algorithm the SAT solver uses is also of importance, as is the nature of the instance (e.g. easy or hard to solve, or satisfiable or unsatisfiable)

The encoding process must be reasonably efficient Since the solving process of the problem is delegated to a SAT solver, the encoding process usually is a relatively insignificant part of the pipeline. In the experimental evaluation in this thesis and existing literature, the encode time is usually not counted with the solve time, since we assume that the encoding process can be optimized in the future. However, for larger problems, some care has to be taken that the encoding process fits within the time and memory restrictions of the experimental setup.

Yet, at other times, we know that the optimal encoding of a CO problem can be as hard to compute as the problem is itself. In Chp. 4 we will see that constructing the optimal encoding of the so-called Single Constant Multiplication (SCM) constraint is such a case. We cannot rely on future optimizations of the encoding process, as SCM is NP-hard [17]. This obstacle can be overcome, e.g. by approximating the optimal encoding, or (as we do) by pre-computing the optimal encoding of a wide range of SCM constraints beforehand.

1.3 Outline of chapters and contributions

The thesis is organized in a background chapter, three technical chapters, and a concluding chapter. We now outline the chapters with a short description of their problem domain and main contribution:

Chp. 2: Background This chapter covers background material common to the subsequent technical chapters of this thesis. In Sec. 2.3, we contribute a more formal notion of the correctness of encoding complex CP models to SAT, which is somewhat missing from the literature despite its use in practice.

Chp. 3: Coupling Different Integer Encodings for SAT As mentioned in Sec. 1.2.2, changing an integer variable's encoding requires changing the encoding of its associated constraints. In most existing SAT encoders, the variable's encoding is fixed for this reason, despite its impact on SAT solver performance. In this chapter, we show how to resolve different variable encodings using *coupling* constraints, leading to improved SAT solver performance on five mixed encoding case studies.

This chapter is adapted from published work [18].

Chp. 4: Applying Single Constant Multiplication to SAT Because of its ubiquity, the integer linear constraint is one of the most important to encode to SAT effectively. In the hardware circuit design community, the expensive constant multiplication of each linear term is often decomposed into cheaper operations, namely additions, subtractions and multiplications by a power of two (i.e. shifts). For example, a circuit for the constant multiplication of $5 \cdot x$ is replaced by a cheaper circuit for $4 \cdot x + x$. Since both circuits and SAT encodings can use a binary representation of integer variables, these techniques are applicable to SAT encodings as well. This chapter aims to carry over the advantages from hardware to software by pre-computing the optimal decomposition for SAT according to various objectives, which we experimentally show improve SAT solver performance in different scenarios.

This chapter is adapted from published work [19].

Chp. 5: Revisiting Pseudo-Boolean Encodings from an Integer Perspective This chapter continues the study of integer linear constraints by showing how three existing encoding methods can be alternatively described as distinct *integer decompositions*. The decompositions, when encoded themselves, yields an effectively equivalent set of clauses as the original method. They thus encapsulate a significant amount of shared behaviour. Our improved understanding of the encoding methods allows us to extend the base methods in

six ways. Some of the extensions are theoretical, such as simplified proofs of correctness and propagation strength, and others are practical which improve the SAT solver performance.

Chp. 6: Conclusion This chapter concludes the thesis by summarizing the contributions and opportunities for future work of each chapter.

— Chapter 2 —

Background

This chapter covers background material common to the subsequent technical chapters of this thesis.

This chapter is organized as follows. In Sec. 2.1, we cover Combinatorial Optimization (CO), with a focus on Constraint Programming (CP). In Sec. 2.2, we cover the Boolean Satisfiability (SAT) problem and its solving techniques. In Sec. 2.3, we cover how to solve CP by encoding to SAT. In Sec. 2.4, we cover constraint modelling languages with a focus on MiniZinc.

The background is largely adapted from various primary sources. For Sec. 2.1, the primary source on all things CP is by [20], with additional material by [21] on modelling and online solving of CO. For Sec. 2.2 and Sec. 2.3, the primary source on SAT solving and encoding is by [9], with additional material by [22], on choosing SAT encoding using Machine Learning (ML), and by [23], on incremental SAT solving. For Sec. 2.4, a primary source is by [24].

2.1 Combinatorial Optimization

CO is the field of finding an (optimal) object from a finite set of objects: the mathematical version of picking a needle from a haystack. In Sec. 2.1.1, we cover mathematical conventions used throughout this thesis. In Sec. 2.1.2, we formalize the CO problems and solutions we are interested in as CSPs. In Sec. 2.1.3, we cover CP, a method to search for solutions.

2.1.1 Mathematical conventions

In this thesis, we follow or adapt the following mathematical conventions:

- The equality symbol $=$ is used in constraints (see Def. 2.2), e.g. $a = b$ constrains variables a and b to take the same value.
- The $\neq$ symbol is called a *disequality*, while $\geq, >, <, \leq$ are called *inequalities*.
- The equivalence symbol $\equiv$ is used for equivalent expressions, e.g. $f(a) \equiv b$.
- The logarithm function uses base 2, i.e. $\log(x) \equiv \log_2(x)$.
- A *list* $[a_1, ..., a_n]$ is a sequence of values $a_1, ..., a_n$.
- A *set* written $\{a_1, ..., a_n\}$ where $a_1, ..., a_n$ have a complete order relation implicitly assumes that $a_i < a_{i+1}$ for $1 \leq i < n$.
- An *interval* $[l..u]$ represents the set of integers $\{l, l+1, ..., u\}$.

- A *function* $f : A \to B$ maps values from *function domain* A to *codomain* B.
  - We use the term *function domain* to disambiguate from (variable) domains.
- A *surjective* function (i.e. a function for which every element of its codomain is mapped to by at least one element of its domain) is denoted $f : A \twoheadrightarrow B$.
- A *map* $f = \{(k_1 \mapsto v_1), ..., (k_n \mapsto v_n)\}$ is a set of n key-value pairs so that $\forall_{i \equiv 1}^{n} f(k_i) \equiv v_i$.
- Lists, sets, and maps are *collections* supporting list, set or map *comprehension*, e.g. $\{i \mid i \in \mathbb{N}, i \bmod 2 \equiv 0\}$ is the set of all even natural integers $\{0, 2, 4, ...\}$.
- A *restriction* $f|_{K'}$ of a function $f : K \to V$ to $K' \subseteq K$ is a new function such that $\forall_{k' \in K'}(f|_{K'}(k') \equiv f(k'))$.

2.1.2 Constraint Satisfaction Problem

To formalize CO problems, we adapt the Constraint Satisfaction Problem (CSP) [25]:

Definition 2.1 (CSP) A Constraint Satisfaction Problem (CSP) is a tuple $P \equiv \langle \mathcal{X}, D, \mathcal{C} \rangle$, where

- $\mathcal{X}$ is a set of (decision) variables $\mathcal{X} \equiv \{x_1, ..., x_n\}$;
- D maps every $x \in \mathcal{X}$ to its domain, which is a finite set of discrete values; and
- $\mathcal{C}$ is a set of constraints (see Def. 2.3).

The functions $\text{vars}(P)$ and $\text{cons}(P)$ yield $\mathcal{X}$ and $\mathcal{C}$, respectively.

We can attempt to solve P by assigning variables to values:

Definition 2.2 (Assignment) An assignment $\mathcal{A}$ of P maps every variable $x \in X$, $X \subseteq \text{vars}(P)$ to a value from the reals. A *partial* assignment has $X \subset \text{vars}(P)$. A *total* assignment has $X \equiv \text{vars}(P)$. Extending an assignment adds another variable assignment to X in some way.

Note that the assignment of a variable does not necessarily adhere to that variable's domain. The constraints of the CSP dictate successful solve attempt.

Definition 2.3 (Constraint) Each constraint $c \in \text{cons}(P)$ is a relation over a set of variables $\text{vars}(c) \subseteq \text{vars}(P)$. The relation specifies $\text{sols}(c)$, a set of assignments of $\text{vars}(c)$ that are said to satisfy c. Satisfying assignments are also called the *solutions* of c. An *intensional* constraint specifies a formula which solutions must satisfy. An *extensional* constraint explicitly lists solutions. The number of variables $|\text{vars}(c)|$ is the *arity* of c. A *unary*, *binary*

or *non-binary* constraint has an arity of one, two or greater, respectively. A constraint c over variables $x_1, ..., x_r$ of arity r is also written as $c(x_1, ..., x_r)$.

For example, the extensional constraint $\{\{(x \mapsto 1), (y \mapsto 2)\}, \{(x \mapsto 2), (y \mapsto 3)\}\}$ lists two solutions, while the intensional constraint $x \neq y$ allows an infinite set of assignments for which $\mathcal{A}(x) \neq \mathcal{A}(y)$ holds (e.g. $\{(x \mapsto 1), (y \mapsto 2)\}$ or $\{(x \mapsto 1.0), (y \mapsto 1.1)\}$).

Each variable in a CSP P has an implicit domain constraint, $x \in D(x)$. Together, the domain constraints define the *search space* $\mathcal{A}^*$ of P:

$$\mathcal{A}^*(P) \equiv \text{sols}\left(\forall_{x \in \text{vars}(P)}(x \in D(x))\right) \equiv \prod_{x \in \text{vars}(P)} \{(x \mapsto d) \mid d \in D(x)\}$$

To solve a CSP, we must find assignments of its search space that satisfy the constraints.

Definition 2.4 (Solution) A total assignment $\mathcal{A}$ of P is a solution of P if it satisfies every $c \in \text{cons}(P)$. The *solution space* $\text{sols}(P)$ are all solutions of P:

$$\text{sols}(P) \equiv \left\{\mathcal{A} \mid \mathcal{A} \in \mathcal{A}^*(\text{vars}(P)), \forall_{c \in \text{cons}(P)}\left(\exists_{\mathcal{A}' \in \text{sols}(c)}(\mathcal{A}' \subseteq \mathcal{A})\right)\right\} \tag{2.1}$$

If the solution space is empty, then P is said to be *unsatisfiable*, otherwise it is *satisfiable*.

Usually, a CSP solver yields one or all solutions, or determines unsatisfiability.

Constrained Optimization Problem

Generally, some solutions might be preferable to other solutions. For instance, if different satisfiable rosters of nurses exist, we prefer those of lower cost, or those which align better with nurse shift preferences. In such cases, the problem can be defined as a Constrained Optimization Problem (COP).

Definition 2.5 (COP) A Constrained Optimization Problem (COP) is a quadruple $\langle \mathcal{X}, D, \mathcal{C}, f \rangle$ where $P' \equiv \langle \mathcal{X}, D, \mathcal{C} \rangle$ is a CSP and f is the *objective* function of P. For some subset of variables $\text{vars}(f) \subseteq \text{vars}(P)$, the function $f : \prod_{x \in \text{vars}(f)} D(x) \to \mathbb{Z}$ maps every assignment in the search space to an integer value.[4] The goal of P is to find solutions in $\text{sols}(P)(\equiv \text{sols}(P'))$ which either minimizes or maximizes (which is pre-determined) the

[4] This is sufficient for the purposes of this thesis.

value of f. A solution which is minimal (or maximal) with regard to f is known as an *optimal* solution, f^*.

Usually, a COP solver yields solutions which strictly increase the objective until an optimal solution is found, and then terminate.

Domains

The variables can be of various types, since the decision can be yes-or-no ("is this particular nurse scheduled?"), or regarding some quantity ("how many nurses do we schedule on a particular day?"). A type will dictate the variable's domain. In all cases, we further restrict ourselves to *finite* domains.

Primarily, we are concerned with two types of domains: integer and Boolean. If a variable x is integer, then $D(x) \subset \mathbb{Z}$. If a variable b is Boolean, then $D(b) \subseteq \mathbb{B}, \mathbb{B} \equiv \{\texttt{false}, \texttt{true}\}$ are the Boolean values `false` and `true`. We can indicate whether an arbitrary domain is Boolean or integer with $D^{\mathbb{B}}(x)$ or $D^{\mathbb{Z}}(x)$, respectively. Thus, a CSP containing exclusively Booleans or integers has domain mapping $D^{\mathbb{B}}$ or $D^{\mathbb{Z}}$, respectively.

We can also represent `false` and `true` as the integers 0 and 1. So in either case, we can represent domains as sets of integers. If $|D(x)| \equiv m$, then $\{d_1, ..., d_m\}$ where $d_i < d_{i+1}$ for $1 \leq i < m$. We will call a set or domain consisting of an interval a *contiguous* set or domain, or a set or domain without *gaps*. We denote the lower and upper bounds $\text{lb}(x) \equiv \min(D(x))$ and $\text{ub}(x) \equiv \max(D(x))$ as the least and greatest values in $D(x)$.

Other examples of variable types are continuous (floating point) values, which are outside the scope of this thesis.

Conjoining and splitting CSPs

Two CSPs P and Q can be *conjoined* into a single CSP $P \wedge Q$ which enforces both. The result is a union of the variables, a union of domains (intersecting domains of shared variables), and a conjunction of constraints:

$$\begin{aligned} P \wedge Q \equiv \quad & \langle \text{vars}(P), D_P, \text{cons}(P) \rangle \wedge \langle \text{vars}(Q), D_Q, \text{cons}(Q) \rangle \\ \equiv \quad & \langle \text{vars}(P) \cup \text{vars}(Q), \\ & D_P|_{\text{vars}(P) \setminus \text{vars}(Q)} \cup D_Q|_{\text{vars}(Q) \setminus \text{vars}(P)} \\ & \cup \left\{ \Big(x \mapsto D_P(x) \cap D_Q(x) \Big) \mid x \in \text{vars}(P) \cap \text{vars}(Q) \right\}, \\ & \text{cons}(P) \cup \text{cons}(Q) \rangle \end{aligned} \tag{2.2.1}$$

A CSP P can be split up into sub-CSPs, each consisting of a single constraint, c, which when conjoined together yields the original CSP:

$$P \equiv \bigwedge_{c \in \text{cons}(P)} \langle \text{vars}(c), D|_{\text{vars}(c)}, \{c\} \rangle \tag{2.3}$$

A shorthand notation for CSPs

To facilitate the readability of the transformation of CSPs used throughout this thesis, we introduce the following shorthand notation of a CSP containing a single constraint c:

$$c(x_1 \subseteq D^{\mathbb{Z}}(x_1), ..., x_r \subseteq D^{\mathbb{Z}}(x_r)) \equiv \langle \{x_1, ..., x_r\}, \{(x_1 \mapsto D^{\mathbb{Z}}(x_1)), ..., (x_r \mapsto D^{\mathbb{Z}}(x_r))\}, \{c\} \rangle$$

The left-hand side of this equivalence denotes the CSP over variables in constraint form, but with the domain of each variable's shown as $x \subseteq D(x)$. To avoid redundancy when variables occur multiple times in the CSP, the domain is only shown for the first occurrence. The additional symbol $\subseteq$ is used to disambiguate from the set membership constraint, i.e. the constraint $x \in S$, and mathematical expressions involving set membership in general.

Example 2.1 An example of denoting a CSP containing a linear inequality and a set membership constraint:

$$\begin{aligned} P &\equiv \langle \{x, y, z\}, \{(x \mapsto \{0, 2\}), (y \mapsto \{0, 2\}), (z \mapsto [0..4])\}, \{x + y = z, z \in [1..3]\} \rangle \\ &\equiv ((x \subseteq \{0, 2\} + y \subseteq \{0, 2\} = z \subseteq [0..4]) \wedge (z \in [1..3])) \end{aligned}$$

If a variable's domain is an arbitrary Boolean or integer domain, we write $x \subseteq D^{\mathbb{B}}(x)$ or $x \subseteq D^{\mathbb{Z}}(x)$, respectively. A special CSP is an unconstrained CSP with a single variable, e.g. x with an arbitrary integer domain:

$$x \subseteq D^{\mathbb{Z}}(x) \equiv \langle \{x\}, \{(x \mapsto D^{\mathbb{Z}}(x))\}, \varnothing \rangle \tag{2.4}$$

For such CSPs, the solution space is equivalent to the search space with $\text{sols}(x \subseteq D(x)) \equiv \mathcal{A}^*(x) \equiv \{\{(x \mapsto d)\} \mid d \in D(x)\}$.

2.1.3 Constraint Programming, propagation and backtracking

For the purposes of this thesis, CP is an algorithmic framework for solving CSPs by alternating *search* and *inference*.

The search process, known as backtracking (BT), assigns a heuristically chosen variable to one of its heuristically chosen domain values (reducing its domain to a singleton). Thus, the so-called *search decisions* incrementally build up a partial assignment. Constraint checking on the

partial assignment determines whether the partial assignment violates any of the constraints. If so, failure is detected, upon which BT undoes the most recent search decision. In this manner, the complete search tree can be explored to find one or all solutions to any CSP.

To improve efficiency, the inference step can perform *constraint propagation* after each search decision. Constraint *propagators* evaluate the (current) domains for a given constraint, and if possible, reduce them, or detect failure by returning an empty domain.

Definition 2.6 (Constraint propagator) A constraint propagator $f_c(D)$ for constraint c and domain mapping D yields a new domain mapping such that the following properties hold:

Checking Suppose D is a total assignment (i.e. maps only to singleton domains). Then, if $D \in \text{sols}(c)$, $f_c(D)$ yields D, else it yields $\varnothing$ (failure). In other words, the propagator recognizes solutions from assignments.

Contracting No domain values are added to the new domains, i.e. $\forall_{x \in \text{vars}(c)}(f_c(D)(x) \subseteq D(x))$.

Correct No solutions of the constraint are removed from D, i.e. $\text{sols}(\langle \text{vars}(c), f_c(D), \{c\} \rangle) \equiv \text{sols}(\langle \text{vars}(c), D, \{c\} \rangle)$.

When a propagator cannot make progress (i.e., $f_c(D) = D$), the propagator is at *fixpoint*. The strongest type of propagator for a constraint is *domain consistent*.

Definition 2.7 (Domain consistency) A domain mapping D is *domain consistent* for a constraint $c(x_1, ..., x_r)$ iff for all $x_i, 1 \leq i \leq r$ and all $d_i \in D(x_i)$, there exists $d_j \in D(x_j), 1 \leq j \leq r, j \neq i$ such that $\{(x_1 \mapsto d_1), ..., (x_r \mapsto d_r)\} \in \text{sols}(c)$.

The domain values d_j of the other variables are called the *support values* of the assignment $(x_i \mapsto d_i)$.

Another important form of consistency is *bounds consistency*, of which there are multiple variants. The most important variant of bounds consistency for this thesis is bounds($\mathbb{R}$) consistency, which only requires that the bounds of each variable are supported by a real value within the bounds of another variable.

Definition 2.8 (bounds($\mathbb{R}$) consistency) A domain mapping D is ***bounds*($\mathcal{R}$) *consistent*** for a constraint $c(x_1, ..., x_r)$ iff for all $x_i, 1 \leq i \leq r$ and all $\boldsymbol{d_i} \in \{\mathbf{lb}(\boldsymbol{x_i}), \mathbf{ub}(\boldsymbol{x_i})\}$, there exists **a real $\boldsymbol{d_j}$ with $\mathbf{lb}(\boldsymbol{x_j}) \leq \boldsymbol{d_j} \leq \mathbf{ub}(\boldsymbol{x_j}), 1 \leq j \leq r, j \neq i$** such that $\{(x_1 \mapsto d_1), ..., (x_r \mapsto d_r)\} \in \text{sols}(c)$.

While a bounds consistent propagators removes fewer domain values than the domain consistent propagator, it is usually more efficient to compute. For example, the `all_different`$([x_1, ..., x_r])$ constraint can be made bounds consistent in $O(r)$ whereas it can be made domain consistent in $O(r^2 \max(|D(x_1)|, ..., |D(x_r)|))$. For binary CSPs, domain consistency is referred to as Arc Consistency (AC) [26], and Generalized Arc Consistency (GAC) for non-binary.

To solve a COP, BT can also adopt the classic Brand and Bound (BnB) [27] approach by evaluating the objective. For a maximization problem, BnB holds the $\text{lb}(f)$ (worst-case objective if the current partial assignment is completed, or an underestimation thereof) and $\text{ub}(f)$ (best objective of any candidate solution found so far). If at any search node we have $\text{lb}(f) \geq \text{ub}(f)$, this sub-branch is pruned and we backtrack. If $\text{lb}(f) < \text{ub}(f)$, the next search decision is made to improve the objective further. Once the full search tree is explored, $f = \text{ub}(f)$.

In this thesis, most techniques aim to solve CSPs efficiently. Since a COP can be solved as a CSP using BT, and since we can solve maximization by minimization if we negate the objective, these techniques also apply to COPs.

2.2 SAT problems and SAT solvers

This section covers a) The SAT problem, b) Solving SAT problems, and c) Solving MaxSAT problems.

2.2.1 The SAT problem

A SAT problem S can be seen as a special case of a CSP in which all variables are Boolean. Accordingly, its domain mapping is $D^{\mathbb{B}}$. Often, the constraints are restricted to be clauses (disjunctions) over literals (Boolean variables or their negation). This simple structure allows for powerful solving techniques, such as CDCL (see Sec. 2.2.2), which cannot be applied to general CSPs. Formally:

Definition 2.9 (SAT) A Boolean Satisfiability (SAT) problem S is a CSP for which every variable $b \in \text{vars}(P)$ is Boolean ($D(b) \subseteq \mathbb{B}$) and every constraint $c \in \text{cons}(P)$ is a Boolean formula. A *literal* l is either a Boolean variable b or its negation $\neg b$. A *clause* is a disjunction of literals. If $\mathcal{C}$ contains only clauses, then S is said to be in Conjunctive Normal Form (CNF).

From this perspective, a SAT problem in CNF with n variables and k clauses is a CSP of the form:

$$\bigwedge_{1 \leq i \leq k} \left(\bigvee_{b \in X, X \subseteq \mathcal{X}} (b \in D^{\mathbb{B}}(b)) \vee \bigvee_{b \in \mathcal{X} \setminus X} (\neg b \in D^{\mathbb{B}}(b)) \right)$$

From another perspective, a SAT problem S is a propositional logic formula only containing Boolean connectives `AND`, `OR` and `NOT`. A solution to this problem assigns each variable to `true` or `false` such that the entire formula evaluates to `true`.

In some SAT terminology which we will *not* use, solutions are called *models*. A SAT problem in Disjunctive Normal Form (DNF) is a disjunction of conjunctions of literals. In this thesis, we will assume a SAT problem is in CNF, unless noted otherwise. The size of SAT instance has three dimensions:

- the number of variables $|\text{vars}(S)|$;
- the number of clauses $|\text{cons}(S)|$; and
- the number of literals $\sum_{c \in \text{cons}(S)} \text{vars}(c)$.

We extend negation to operate on literals, i.e. $\neg l \equiv \neg x$ if $l \equiv x$ and $\neg l \equiv x$ if $l \equiv \neg x$. We use the notation $l = v$ where l is a literal and $v \in \{0, 1\}$ to encode the appropriate form of the literal, i.e. if $v = 1$ it is equivalent to l and if $v = 0$ it is equivalent to $\neg l$. The notation $l \neq v$ is defined similarly to encode $\neg(l = v)$. This also extends some integer operations on literals: $\neg l \equiv 1 - l$, $|l| = v$ and $l \cdot l' \equiv l \wedge l'$.

In a domain mapping D, we can check whether a literal l is fixed to true or false by $l \in D \equiv (D(|l|) = \{l\})$. E.g. for $D \equiv \{(x \mapsto \{0\}), (y \mapsto \{0, 1\})\}$, we have $\neg x \in D$ because the variable x has only one domain value left (fixed), which is 0 (equal to the literal $\neg x$). A domain is updated by $D \wedge l \equiv D|_{D \setminus |l|} \cup \{(|l| \mapsto \{l\})\}$. We can also denote a SAT assignment as a conjunction of literals (which would satisfy the formula): $\bigwedge_{(l \mapsto d) \in \mathcal{A}} l = d$.

Some special types of formulas and clauses are:

- The empty formula has zero clauses and is equivalent to `true`.
- The empty clause has zero literals, and is equivalent to `false`.
- A unary, binary, ternary clause has 1, 2, 3 literals, respectively.

MaxSAT problems

Maximum Satisfiability (MaxSAT) is the optimization version of SAT. Since we express SAT as an CSP, we will express MaxSAT as a COP where the objective function f is to maximize the number of satisfied clauses. MaxSAT is more often represented as the equivalent MinUNSAT

problem, where the objective is to minimize the number of unsatisfied clauses. Minimizing the number of satisfied clauses, MinSAT, is quite a different problem. In a *weighted* MaxSAT S, every clause $c_i \in \text{cons}(S)$ is paired with a weight (c_i, w_i), in which case $f \equiv \sum_{(c,w)\in\text{cons}(S)} w_i c_i$. In *unweighted* MaxSAT, every $w_i \equiv 1$. Partial MaxSAT is weighted MaxSAT with objective f, but a set of *hard* clauses must be satisfied.

MaxSAT is functionally equivalent to a form where all soft clauses are unary clauses. Unary clauses with the same variable can be merged. Consequently, weight w_b is associated with each variable $b \in \text{vars}(S)$ and the minimization objective is $f \equiv \sum_{b\in\text{vars}(S)} w_b b$. Internally, many MaxSAT solvers treat the problem in this form.

2.2.2 Solving SAT problems

In this section, we cover the fundamental SAT and MaxSAT solving techniques and algorithms.

The Davis-Putnam algorithm

The Davis-Putnam (DP) [28] algorithm is based on the *resolution* rule, which says that a clause containing l and another containing $\neg l$ can be *resolved* into a new clause containing all literals of both but without l.

$$(p_1 \vee \cdots \vee p_n \vee l) \wedge (\neg l \vee q_1 \vee \cdots \vee q_m) \equiv p_1 \vee \cdots \vee p_n \vee q_1 \vee \cdots \vee q_m$$

Intuitively, the resulting clause is true whether l holds or not. In the logical statement, "If it rains, then we play video games, and if it does not rain, then we go outside", the resolvent "we will play video games *or* go outside" regardless of whether it rains or not. The DP algorithm will resolve the entire formula for one clause at a time. However, the DP algorithm has exponential space requirements. If we want to resolve one clause with l, we need to resolve with every *other* clause containing $\neg l$.

One important special case of *resolution* is *unit resolution*, where at least one of the resolved clauses is unary.

$$(p_1 \vee \cdots \vee p_n \vee \neg l) \wedge l \equiv p_1 \vee \cdots \vee p_n$$

Resolution on clauses containing exactly one literal l, known as *unit clauses*, assigns that literal to `true`.

The heuristic for choosing a resolution literal is also of importance.

The Davis-Putnam-Logemann-Loveland algorithm

The Davis-Putnam-Logemann-Loveland (DPLL) [29] algorithm is fundamentally similar to CP-style BT (see Sec. 2.1.3). Each search decision at the i-th depth of the search tree is called the i-th *decision level*. Only one type of propagator is applied, which is Unit Propagation (UP).

After every search decision, some clauses may become unary (*unit clauses*). We perform unit propagation on a unit clause by assigning its literal l. This satisfies the unit clause and all clauses containing l. Then, we remove (satisfy) all clauses containing l, and remove (falsify) literals $\neg l$ in all other clauses, or detect failure if one of those clauses is unit. Unit propagation leads to more unit propagation, until fixpoint. We can define unit propagation as a constraint propagator:

Definition 2.10 (UP) Unit Propagation (UP) is a propagator $f_{l_1 \vee \cdots \vee l_n}$ for the clause constraint $l_1 \vee \cdots \vee l_n$ and current domain D:

$$f_{l_1 \vee \cdots \vee l_n}(D) \equiv \begin{cases} \varnothing & \text{if } \forall_{1 \leq i \leq n}(\neg l_i \in D) \\ D \cup \{l_j\} & \text{if } \forall_{1 \leq i \neq j \leq n}(\neg l_i \in D) \end{cases}$$

Example 2.2 Throughout this thesis, we visualize the search decisions of the DPLL algorithm the inferences by UP in the following way:

$$\begin{aligned} (p \vee q) \wedge (\neg p \vee r) \wedge (\neg r \vee s \vee q) \quad & \textit{dec. } p \\ r \wedge (\neg r \vee s \vee q) \quad \Longrightarrow \quad & r \\ (s \vee q) \quad & \textit{fixp.} \end{aligned}$$

Given the initial SAT problem of three clauses, search first decides (*dec.* p). This satisfies the first clause by its first literal, indicated by the green colour the literal p and of the parentheses $(...)$ of the clause. In the second clause, $\neg p$ is falsified (indicated by the red colour), which leaves the unit clause r in the next round to be propagated by UP ($\Longrightarrow r$). A final binary clause is left, which means UP is at fixpoint (*fixp.*) with no further search decisions specified.

An alternative visualization shows binary clauses as implications.

$$\begin{aligned} (\neg p \rightarrow q) \wedge (p \rightarrow r) \wedge (\neg r \vee s \vee q) \quad & \textit{dec. } p \\ r \wedge (\neg r \vee s \vee q) \quad \Longrightarrow \quad & r \\ (\neg s \rightarrow q) \quad & \textit{fixp.} \end{aligned}$$

Note that in the case of a false antecedent, e.g. $(p \rightarrow r)$, the implication is satisfied as shown by the green parentheses. On the other hand, a true antecedent, e.g. $(p \rightarrow r)$, has the implication operator coloured green as well to indicate that the consequent r will be unit propagated next.

DPLL improves on DP's exponential space complexity requirement since no new clauses are generated, but this comes at the cost of an exponential run-time complexity.

Applying unit resolution (or *unit propagation*), pure literal elimination and search heuristic can improve the runtime in many cases.

Backjumping and Conflict-driven Clause Learning
In the Davis-Putnam-Logemann-Loveland (DPLL) [29] algorithm, BT always goes back one level (from i to $i-1$). In contrast, *backjumping* (or *non-chronological backtracking*) can backtrack multiple levels, $i' < i$. This technique comes from solving CSPs, found in its earliest form in [30].

When a conflict (empty clause) is found at level i, rather than undoing only the latest assignment of b_i, *conflict analysis* can find *conflict sets* [31] of multiple assignments which together contribute to the conflict. Conflict sets can be found via an *implication graph*, which records what set of assignments unit propagate which new assignment and at what level. Cuts in the implication graph correspond to conflict sets.

We can backjump to the earliest decision level of the literals in the conflict sets. Furthermore, a breakthrough was made with CDCL and *clause learning* (usually attributed to [32], but it is also present in [30]). By adding the negation of the literals in the conflict set as a clause, we prevent similar assignments in future search. Backjumping and clause learning is the backbone of most modern SAT solvers.

2.2.3 Solving MaxSAT problems

To solve a COP using a SAT solver, we will need some form of MaxSAT solving. A straightforward way to solve MaxSAT is with BnB (see Sec. 2.1.3), using unit propagation to do inference.

With some exceptions, BnB does not scale well for MaxSAT problems with more than a thousand variables. A better approach employed by most modern MaxSAT solvers is to use an oracle SAT solver on the MaxSAT decision problem $\mathcal{D}(M, k)$ (i.e., can we solve MaxSAT formula M with cost $f \leq k$?). We can solve $\mathcal{D}(M, k)$ with a SAT solver by encoding it as a SAT problem S (e.g. using Pseudo-Boolean (PB) constraints to encode the objective, see Sec. 2.3.5).

The Linear SAT/UNSAT (LSU) [33] algorithm, which is a *model-improving* approach (model meaning solution), solves the sequence of $\mathcal{D}(M, k_0), \mathcal{D}(M, k_1), ...$ where $k_0 = \inf$ (or $\text{ub}(f)$) and k_i is the objective found by $\mathcal{D}(M, k_{i-1}) - 1$. When unsatisfiable is found, k_{i-1} is the optimal solution. The upside is that this algorithm will find intermediate, ever-improving solutions. The downside is that initial input $\mathcal{D}(M, k_i)$ for the SAT solver are large they are positively proportional to k.

The opposite approach to SAT-UNSAT is SAT where initially $k_0 = 0$ (or $\text{lb}(f)$). In other words, by requiring that k_0 is (probably) less than the optimal solution f^*, solving $\mathcal{D}(M, k_0)$ will likely yield in unsatisfiable. By sequentially solving $k_i > k_{i-1}$, the algorithm will eventually relax the problem enough to find a single, optimal solution (but no intermediate solutions, unlike LSU).

A very important ingredient of the UNSAT-SAT algorithm is an *assumption-based* SAT solver [34]. Once a clause is added to the model, it cannot normally be retracted (relaxed). However, we can condition a clause C upon an *assumption literal* a by posting $a \rightarrow C$. The assumption interface allows us to solve $a \wedge S$. During these *incremental* solve calls (with added clauses or a different set of assumption literals), the internal state (such as learned clauses) of the SAT solver is preserved. With some additional calls, the solver can also generate *UNSAT cores* (unsatisfiable subsets of its input formula). A key improvement by [35] is to recognize that only the soft clauses present in UNSAT cores need to be considered for relaxation.

2.3 Encoding CSPs to SAT

Interesting problems are often not represented in SAT form, let alone CNF. To solve such problems with a SAT solver, we first need to translate (or *encode*) the problem from CSP P into SAT, and specifically CNF. There are multiple ways to encode the same CSP. Furthermore, the complexity of the encoding depends on the complexity of the CSP in terms of what type of domains and constraints it contains. We will incrementally build up an encoding framework which can encode CSPs, from simple to more complex CSPs, using a variety of methods.

This section is organized as follows. In Sec. 2.3.1, we define an encoding in its most general form and give an example of how to encode a Boolean CSP. In Sec. 2.3.2, we show how to encode Boolean CSPs using logical equivalence and the Tseitin encoding. In Sec. 2.3.3, we show how to encode the integer variables of a CSP using three fundamental methods: the direct, order and binary encoding. In Sec. 2.3.4, we show how to encode CSPs with extensional constraints over integer variables. In Sec. 2.3.5, we show how to encode CSPs with intensional constraints over integer variables.

2.3.1 Encoding fundamentals

We can now define an encoding in its most general form.

Definition 2.11 (Encoding) CSP Q encodes CSP P iff for P there exists a surjective (and total) *interpretation function* $\mathcal{J}_P : \text{sols}(Q) \twoheadrightarrow \text{sols}(P)$.

This means that to prove Q is a correct encoding of P, we need to find an interpretation function which has the following two properties:

Codomain restriction Every solution of the encoding Q maps to a solution of P (because $\mathcal{J}$ is a function with codomain $\text{sols}(P)$)

Surjectivity For every solution of P there is at least one solution of Q that maps to it (because $\mathcal{J}$ is surjective).

If there is no such function, then Q does not encode P. This is harder to show, but generally not what we are interested in.

Alternatively, we can prove Q encodes P by showing that for some interpretation function, the set of solutions of P is equivalent to the set of the interpreted solutions of Q:

$$\text{sols}(P) \equiv \{\mathcal{J}_P(\mathcal{A}) \mid \mathcal{A} \in \text{sols}(Q)\}$$

We will illustrate the use of interpretation functions by considering the following special Boolean CSP encoding case. To prove that a Boolean CSP Q encodes another Boolean CSP P, we can often use the following interpretation function provided that $\text{vars}(P) \subseteq \text{vars}(Q)$:

$$\mathcal{J}_P^{\mathbb{B}}(\mathcal{A}) \equiv \mathcal{A}|_{\text{vars}(P)} \tag{2.5}$$

Example 2.3 Consider two example Boolean CSPs, $P \equiv (p \vee q)$ and $Q \equiv (p \vee r) \wedge (\neg r \vee q)$. We can say that Q encodes P, because there exists a surjective interpretation function, namely Eq. 2.5, which yields all three solutions of $\text{sols}(P)$ given all four solutions of $\text{sols}(Q)$:

$$\begin{aligned}
&\left\{\mathcal{J}_P^{\mathbb{B}}(\mathcal{A}) \mid \mathcal{A} \in \text{sols}(Q)\right\} \\
&\equiv \left\{(\neg p \wedge q \wedge r)|_{\{p,q\}}, (p \wedge \neg q \wedge \neg r)|_{\{p,q\}}, (p \wedge q \wedge \neg r)|_{\{p,q\}}, (p \wedge q \wedge r)|_{\{p,q\}}\right\} \\
&\equiv \{\neg p \wedge q, p \wedge \neg q, p \wedge q, p \wedge q\} \equiv \{\neg p \wedge q, p \wedge \neg q, p \wedge q\} \equiv \text{sols}(P)
\end{aligned}$$

But we cannot say that P encodes Q, because there is no function (surjective or otherwise) which can map a single solution $(p \wedge q) \in \text{sols}(P)$ to its two output solutions, $p \wedge q \wedge \neg r$ and $p \wedge q \wedge r$ of $\text{sols}(Q)$.

The *encoding function* E given CSP P yields an encoding $E(P)$ of P.

A SAT encoding has the additional requirement of being a SAT problem in CNF. However, other types of encodings (e.g. MIP encodings, see Sec. 2.4.3) are also covered by Def. 2.11. In this thesis, if not specified otherwise, an encoding is always a SAT encoding. If an encoding is non-SAT, it is a CSP encoding, which we will call (CSP) *decompositions*.

We call variables $\text{vars}(P) \cap \text{vars}(E(P))$ occurring in both P and $E(P)$ the *principal* variables. The encoding function might introduce new variables $\text{vars}(E(P)) \setminus \text{vars}(P)$, which we will call *auxiliary* variables.

In the literature, *equisatisfiability* is often used to qualify encoding-like transformations (i.e. CSPs P and Q are equisatisfiable iff both are satisfiable or if both are unsatisfiable). P and $E(P)$ are indeed always equisatisfiable, since $\text{sols}(P)$ is empty iff $\text{sols}(E(P))$ is empty. However, equisatisfiability by itself is too weak to describe a generally correct encoding. For instance, a Boolean CSP $p \vee q$ is equisatisfiable with the CSP `true`, yet the latter is not an encoding of the former.

A better definition for the correctness of Boolean encodings comes from [36], which says a constraint c is encoded by $E(c)$ iff: for each solution $\mathcal{A} \in \mathcal{A}^*(c)$, $\mathcal{A} \in \text{sols}(c)$ iff $\mathcal{A} \subseteq \mathcal{A}', \mathcal{A}' \in \text{sols}(E(c))$. This aligns with our definition of encodings, in particular with the interpretation function for Boolean encodings (see Eq. 2.5). But, we can extend our definition to support the encoding of integer variables (see Sec. 2.3.3).

Notice that both definitions of encoding correctness are asymmetrical between the original problem (or constraint) and the encoded problem, i.e. one encodes the other but not necessarily the other way around. In [37], the same author defines S-equivalence, which is more or less a symmetrical version of the encoding relationship. Let S be a set of variables. Then, two CSPs P and Q are S-equivalent iff: for every assignment $\mathcal{A}$ of S, P is satisfiable by an assignment extending $\mathcal{A}$ iff Q is satisfiable by an assignment extending $\mathcal{A}$ (i.e. $\forall_{\mathcal{A} \in \mathcal{A}^*(S)}(\mathcal{A} \subseteq \text{sols}(P)|_S$ iff $\mathcal{A} \subseteq \text{sols}(Q)|_S)$).

S-equivalence also bridges (logical) equivalence (where CSPs are satisfied by the same set of models) and equisatisfiability. Namely, two S-equivalent CSPs P and Q are equivalent iff $S \supseteq \text{vars}(P) \cup \text{vars}(Q)$, and are equisatisfiable iff $S \equiv \varnothing$. Surprisingly, we cannot find an earlier source for this definition, although this concept should be at the heart of encoding methods.

Constraint encoding functions

The *global* encoding function $E(P)$ is defined to encode a general CSP with different constraints and domains (within the scope of this thesis, e.g. having finite integer and Boolean domains). A *constraint* encoding function E^c is defined to encode a specific constraint, c. Since the input to an encoding function is a CSP, the global encoding function splits up its input, the *global* CSP, into single constraint sub-CSPs with Eq. 2.3 and encodes each with its appropriate constraint encoding function. The encoded output CSPs are conjoined by Eq. 2.2.1.

$$E(P) \equiv \bigwedge_{c \in \text{cons}(P)} E^c(c)$$

Encoding the constraints as CSPs rather than as constraints allows us access to the variable domains from the CSP. Recall that these sub-CSPs are still often denoted as constraints with domains (using the notational shorthand from Sec. 2.1.2). Furthermore, we often denote $E^c(c)$ as simply $E(c)$, disambiguating the encoding function based on its input. If multiple encodings are known for c, we will disambiguate further.

2.3.2 Encoding Boolean CSPs to SAT

In this section, we will show various methods to encode Boolean CSPs (in which all variables have Boolean domains) to SAT.

Encoding by logical equivalence

When two Boolean CSPs are logically equivalent, e.g. $p \in D^{\mathbb{B}}(p) \oplus q \in D^{\mathbb{B}}(q) \equiv (p \vee q) \wedge (\neg p \vee \neg q)$, they have the same set of solutions. Equivalent CSPs necessarily have the same set of variables. Thus, equivalent CSPs encode each other, since the identity function is a correct interpretation.

We cover some important laws of equivalence which are used throughout this thesis to encode CSPs. Sub-expressions under disjunction $\bigvee$ and conjunction $\bigwedge$ connectives use the associative laws:

$$E(p \vee (l_1 \vee \cdots \vee l_r)) \equiv p \vee l_1 \vee \cdots \vee l_r \tag{2.6.1}$$

$$E(p \wedge (l_1 \wedge \cdots \wedge l_r)) \equiv p \wedge l_1 \wedge \cdots \wedge l_r \tag{2.6.2}$$

And distributive laws:

$$E(p \vee (l_1 \wedge \cdots \wedge l_r)) \equiv (p \vee l_1) \wedge \cdots \wedge (p \vee l_r) \tag{2.7.1}$$

$$E(p \wedge (l_1 \vee \cdots \vee l_r)) \equiv (p \wedge l_1) \vee \cdots \vee (p \wedge l_r) \tag{2.7.2}$$

By applying the (generalized) De Morgan's law, we can also encode $\neg \bigvee$ (negated disjunctions) and $\neg \bigwedge$ (negated conjunctions):

$$\neg(l_1 \wedge \cdots \wedge l_r) \equiv \neg l_1 \vee \cdots \vee \neg l_r \tag{2.8.1}$$

$$\neg(l_1 \vee \cdots \vee l_r) \equiv \neg l_1 \wedge \cdots \wedge \neg l_r \tag{2.8.2}$$

Other connectives $\rightarrow$ (implication), $\leftrightarrow$ (bi-implication), $\oplus$ (exclusive-or), and $\neq$ (non-equivalence) are:

$$p \rightarrow q \equiv (\neg p \vee q) \tag{2.9.1}$$

$$p \leftrightarrow q \equiv p \rightarrow q \wedge q \rightarrow p \quad (\equiv (\neg p \vee q) \wedge (p \vee \neg q)) \tag{2.9.2}$$

$$p \neq q \equiv p \oplus q \equiv p \leftrightarrow \neg q \qquad (\equiv (p \vee q) \wedge (\neg p \vee \neg q)) \tag{2.9.3}$$

Note we recursively apply logical equivalences to resolve Eq. 2.9.2 to CNF using Eq. 2.9.1, and Eq. 2.9.3 to Eq. 2.9.2. This also allows us to resolve Boolean sub-expressions.

Example 2.4 An encoding of a Boolean CSP with sub-expressions by recursively applying logical equivalence:

$$
\begin{aligned}
& E((p \wedge \neg q) \leftrightarrow r) \\
\equiv\ & ((p \wedge \neg q) \rightarrow r) \wedge (r \rightarrow (p \wedge \neg q)) && \text{by Eq. 2.9.2} \\
\equiv\ & (\neg(p \wedge \neg q) \vee r) \wedge (\neg r \vee (p \wedge \neg q)) && \text{by Eq. 2.9.1} \\
\equiv\ & ((\neg p \vee q) \vee r) \wedge ((\neg r \vee p) \wedge (\neg r \vee \neg q)) && \text{by Eq. 2.8.1, Eq. 2.7.1} \\
\equiv\ & (\neg p \vee q \vee r) \wedge (\neg r \vee p) \wedge (\neg r \vee \neg q) && \text{by Eq. 2.6}
\end{aligned}
$$

Encoding `EO`, `AMO`, `ALO` and `IC` with equivalence

We now discuss some important Boolean constraints which semantically describe higher-level relations, but can still be encoded by equivalence.

The Exactly-One (`EO`) constraint enforces for a list of Boolean literals $[l_1, ..., l_r]$ that exactly one is true. Specifically:

$$\text{sols}(\texttt{EO}([l_1, ..., l_r])) \equiv \left\{ \left\{ \left(l_j \mapsto j = i\right) \mid 1 \leq j \leq r \right\} \mid 1 \leq i \leq r \right\} \tag{2.10}$$

Similarly, the At-Least-One (`ALO`) constraint enforces that at least one Boolean variable is true. This can be encoded by a single clause:

$$\texttt{ALO}([l_1, ..., l_r]) \equiv \bigvee_{1 \leq i \leq r} l_i \tag{2.11}$$

Similarly, the At-Most-One (`AMO`) constraint enforces that at most one Boolean variable is true. Different encodings of this constraint are discussed in Sec. 2.3.4, but one way to encode `AMO` using equivalence is by the *pairwise* encoding (folklore):

$$\texttt{AMO}([l_1, ..., l_r]) \equiv \bigwedge_{1 \leq i < j \leq r} \left(\left(\neg l_i \vee \neg l_j\right)\right. \tag{2.12}$$

Now, the `EO` constraint can be encoded as the conjunction of the encodings of `ALO` and `AMO` constraints:

$$\texttt{EO}([l_1, ..., l_r]) \equiv \texttt{ALO}([l_1, ..., l_r]) \wedge \texttt{AMO}([l_1, ..., l_r]) \tag{2.13}$$

Another important constraint is the Implication Chain (`IC`) constraint, which encodes a chain of implications over its literals:

$$\texttt{IC}(l_1, ..., l_r) \equiv (l_1 \rightarrow l_2) \land (l_2 \rightarrow l_3) \land \cdots \land (l_{r-1} \rightarrow l_r) \tag{2.14}$$

Specifically, the solution space of the `IC` constraint is:

$$\text{sols}(\texttt{IC}(l_1, ..., l_r)) \equiv \left\{ \left\{ \left(l_j \mapsto j \leq i \right) \mid 1 \leq j \leq r \right\} \mid 1 \leq i \leq r+1 \right\} \tag{2.15}$$

Encoding Boolean CSPs with the Tseitin encoding

By recursively applying the laws of logical equivalence from Sec. 2.3.2 we can encode Boolean CSPs. However, for more complex Boolean constraints, which are a function f of Boolean sub-expressions (e.g. $(p \land q) \lor (c \land (p \oplus q))$ is a disjunction of sub-expressions $p \land q$ and $(c \land (p \oplus q)))$ the encoding grows exponentially in size with regards to the number of literals in the input.

The Tseitin encoding [38] E^T encodes using a linear number of auxiliary variables. It does so by replacing each complex sub-expressions in f by an auxiliary Boolean variable constrained to be true iff the sub-expression is true (e.g. introducing $b_{p \land q} \subseteq D^{\mathbb{B}}\left(b_{p \land q}\right) \leftrightarrow p \land q$). A variant of the Tseitin encoding is the Plaisted-Greenbaum transformation, which instead *half-reifies* the constraint by encoding only one direction (e.g. $b_{p \land q} \subseteq D^{\mathbb{B}}\left(b_{p \land q}\right) \rightarrow p \land q$) if it suffices for equisatisfiability [39].

Definition 2.12 (Tseitin encoding) The Tseitin encoding E^T can encode a *reified* Boolean constraint of the form $b_c \leftrightarrow f(Q_1, ..., Q_r)$, where f is a function of Boolean sub-expressions $Q_1, ..., Q_r$ and $b_{Q_1} \subseteq D^{\mathbb{B}}\left(b_{Q_1}\right), ..., b_{Q_r} \subseteq D^{\mathbb{B}}\left(b_{Q_r}\right)$ are auxiliary Boolean variables:

$$E^T\left(b_c \subseteq D^{\mathbb{B}}(b_c) \leftrightarrow f(Q_1, ..., Q_r)\right) \equiv E\left(b_c \leftrightarrow f\left(b_{Q_1} \subseteq D^{\mathbb{B}}\left(b_{Q_1}\right), ..., b_{Q_r} \subseteq D^{\mathbb{B}}\left(b_{Q_r}\right)\right)\right) \land E^T\left(b_{Q_1} \leftrightarrow Q_1\right) \land \cdots \land E^T\left(b_{Q_r} \leftrightarrow Q_r\right)$$

Then, the Tseitin encoding E^T can also encode a non-reified constraint with $E^T(\texttt{true} \leftrightarrow f(Q_1, ..., Q_r))$. If f returns a literal l , we can use it instead of reifying it with an additional auxiliary variable using $E(b_l \leftrightarrow l)$.

Example 2.5 We now show an example of the Tseitin encoding. Consider the following Boolean constraint modelling an important Boolean formula (the carry of a full adder circuit, used in binary addition) over variables p, q, c and c':

$$c' \leftrightarrow (p \wedge q) \vee (c \wedge (p \oplus q))$$

The first steps of its Tseitin encoding are given below:

$$\begin{aligned} & E^T(c' \leftrightarrow ((p \wedge q) \vee (c \wedge (p \oplus q)))) \\ \equiv\ & E\left(c' \leftrightarrow \left(b_{p\wedge q} \vee b_{c\wedge(p\oplus q)}\right)\right) \wedge E^T\left(b_{p\wedge q} \leftrightarrow p \wedge q\right) \wedge E^T\left(b_{c\wedge(p\oplus q)} \leftrightarrow c \wedge (p \oplus q)\right) \end{aligned}$$

The first encoding has no further sub-expressions, so is encoded using equivalence:

$$\left(\neg c' \vee b_{p\wedge q} \vee b_{c\wedge(p\oplus q)}\right) \wedge \left(\neg b_{p\wedge q} \vee c'\right) \wedge \left(\neg b_{c\wedge(p\oplus q)} \vee c'\right) \tag{2.16}$$

As is the second:

$$\left(\neg b_{p\wedge q} \vee p\right) \wedge \left(\neg b_{p\wedge q} \vee q\right) \wedge \left(\neg p \vee \neg q \vee b_{p\wedge q}\right) \tag{2.17}$$

However, the third encoding call is replaced by two additional encodings:

$$E\left(b_{c\wedge(p\oplus q)} \leftrightarrow c \wedge b_{p\oplus q}\right) \wedge E^T\left(b_{p\oplus q} \leftrightarrow (p \oplus q)\right)$$

The first of which is encoded directly:

$$\left(\neg b_{c\wedge(p\oplus q)} \vee c\right) \wedge \left(\neg b_{c\wedge(p\oplus q)} \vee b_{p\oplus q}\right) \wedge \left(\neg c \vee \neg b_{p\oplus q} \vee b_{c\wedge(p\oplus q)}\right)$$

The second of which has $p \oplus q$, equivalent to $(p \vee q) \wedge (\neg p \vee \neg q)$, which has two sub-expressions that need to be reified:

$$\begin{aligned} & E^T\left(b_{p\oplus q} \leftrightarrow (p \oplus q)\right) \equiv E^T\left(b_{p\oplus q} \leftrightarrow ((p \vee q) \wedge (\neg p \vee \neg q))\right) \\ \equiv\ & E\left(b_{p\oplus q} \leftrightarrow b_{p\vee q} \wedge b_{\neg p\vee\neg q}\right) \wedge E^T\left(b_{p\oplus q} \leftrightarrow (p \vee q)\right) \wedge E^T\left(b_{p\oplus q} \leftrightarrow (\neg p \vee \neg q)\right) \end{aligned}$$

These encodings have no complex sub-expression, and are all encoded directly to clauses similar to Eq. 2.16 and Eq. 2.17.

2.3.3 Encoding integer variables

Many interesting CSPs include integer variables. The domains of integer variables can take many values, but Boolean variables can only take two. This means we require multiple Boolean variables to encode a single integer variable.

There are many ways to encode an integer into Booleans. The three fundamental encoding functions $E^{\mathbb{D}}$, $E^{\mathbb{O}}$ and $E^{\mathbb{B}}$ use the *direct*, *order* and *binary* integer variable encoding, respectively.

Since different integer variable encodings exist, we define an abstract *integer variable encoding function* for integer variable encoding $\mathbb{E}$. The solutions of a direct, order and binary encoding of an example CSP and their interpretation is shown in Tab. 2.1.

The abstract integer variable encoding function

As the input to a constraint encoding function is a CSP of a single constraint c, the input of an integer variable encoding function is an unconstrained CSP with a single integer variable x (see Eq. 2.4). An *integer variable encoding* is just an encoding of (the CSP of) $x \in D^{\mathbb{Z}}(x)$.

Definition 2.13 (Integer variable encoding) An *integer variable encoding function* $E^{\mathbb{E}}$ encodes (the CSP of) $x \in D^{\mathbb{Z}}(x)$ according to particular integer variable encoding method $\mathbb{E}$ with m variables:

$$E^{\mathbb{E}}(x \in D^{\mathbb{Z}}(x)) \equiv \langle \{b_1, ..., b_m\}, D^{\mathbb{B}}, \mathcal{C} \rangle$$

As with any other encoding, to prove that $E^{\mathbb{E}}$ is correct, we need to show the existence of a surjective interpretation function $\mathcal{I}^{\mathbb{E}}_{x \in D^{\mathbb{Z}}(x)} : \text{sols}(E^{\mathbb{E}}(x \in D^{\mathbb{Z}}(x))) \twoheadrightarrow \text{sols}(x \in D^{\mathbb{Z}}(x))$.

An integer encoding function introduces a number of Boolean *encoding* variables $b_1, ..., b_m$ to encode the *encoded* integer variable, x. We use semantic brackets to name the encoding variables, where Boolean $[\![f]\!]$ is true iff the formula f holds.

$$[\![f]\!] \equiv b_i, \text{where } b_i \in \text{vars}(E^{\mathbb{E}}(x))$$

Two Booleans with different names $[\![f]\!]$ and $[\![g]\!]$ refer to the same underlying literal iff $f \equiv g$.

By encoding so-called *consistency constraints* $\mathcal{C}$ we can ensure the domain is faithfully encoded. If the consistency constraints are enforced, we call the encoding *consistent*. Without a consistency constraint, the interpretation might map outside $\text{sols}(x)$, or even outside $\mathcal{A}^*(x)$. However, we will see cases where the consistency constraints can be omitted nonetheless.

The direct encoding

The *direct encoding* (folklore, but usually attributed to [40]), $E^{\mathbb{D}}(x \in D^{\mathbb{Z}}(x))$, encodes an integer variable using primitive equality constraint $x = d$. The idea is to let the SAT solver assign an encoding variable $[\![x = d]\!]$ to `true` in a solution of $E^{\mathbb{D}}(x \in D^{\mathbb{Z}}(x))$ iff x is assigned to d in a solution of $x \in D^{\mathbb{Z}}(x)$. Encoding an `EO` (see Eq. 2.13) constraint enforces that exactly one $[\![x = d]\!]$ is true.

Definition 2.14 (Direct encoding) The direct (integer variable) encoding function $E^{\mathbb{D}}$ encodes (the CSP of) an integer variable x:

$$E^{\mathbb{D}}(x \subseteq \{d_1, ..., d_m\}) \equiv \texttt{EO}([[\![x = d_1]\!], ..., [\![x = d_m]\!]]) \tag{2.18}$$

We can now prove that the direct encoding is a correct encoding of $x \subseteq D^{\mathbb{Z}}(x)$ by introducing an interpretation function, $\mathcal{J}^{\mathbb{D}}_{x \subseteq D^{\mathbb{Z}}(x)}$.

Theorem 2.1 The encoding function $E^{\mathbb{D}}$ (from Eq. 2.18) is correct, since there exists the following surjective interpretation function $\mathcal{J}^{\mathbb{D}}_{x \subseteq D^{\mathbb{Z}}(x)} : \text{sols}(E^{\mathbb{D}}(x \subseteq D^{\mathbb{Z}}(x))) \twoheadrightarrow \text{sols}(x \subseteq D^{\mathbb{Z}}(x))$:

$$\mathcal{J}^{\mathbb{D}}_{x \subseteq D^{\mathbb{Z}}(x)}(\mathcal{A}) \equiv \sum_{d \in D(x)} d \cdot \mathcal{A}([\![x = d]\!]) \tag{2.19}$$

Before the proof, let us look at a concrete example of the direct encoding and its interpretation.

Example 2.6 The direct encoding of the CSP of $x \subseteq [1..3]$ is:

$$\begin{aligned}
& E^{\mathbb{D}}(x \subseteq [1..3]) \\
\equiv\ & \texttt{EO}([[\![x = 1]\!], [\![x = 2]\!], [\![x = 3]\!]]) \\
\equiv\ & ([\![x = 1]\!] \vee [\![x = 2]\!] \vee [\![x = 3]\!]) \wedge ([\![x \neq 1]\!] \vee [\![x \neq 2]\!]) \\
& \qquad \wedge ([\![x \neq 1]\!] \vee [\![x \neq 3]\!]) \wedge ([\![x \neq 2]\!] \vee [\![x \neq 3]\!])
\end{aligned}$$

Each of the three solutions in $\text{sols}(E^{\mathbb{D}}(x \subseteq [1..3]))$ are mapped by $\mathcal{J}^{\mathbb{D}}_{x \subseteq D^{\mathbb{Z}}(x)}$ to each of the three solutions in $\text{sols}(x)$:

$$\begin{aligned}
& \left\{ \mathcal{J}^{\mathbb{D}}_{x \subseteq D^{\mathbb{Z}}(x)}(\mathcal{A}) \mid \mathcal{A} \in \text{sols}(E^{\mathbb{D}}(x \subseteq [1..3])) \right\} \\
\equiv\ & \Big\{ \mathcal{J}^{\mathbb{D}}_{x \subseteq D^{\mathbb{Z}}(x)}(\ \ [\![x = 1]\!] \wedge \neg[\![x = 2]\!] \wedge \neg[\![x = 3]\!]), \\
& \quad \mathcal{J}^{\mathbb{D}}_{x \subseteq D^{\mathbb{Z}}(x)}(\neg[\![x = 1]\!] \wedge \ \ [\![x = 2]\!] \wedge \neg[\![x = 3]\!]), \\
& \quad \mathcal{J}^{\mathbb{D}}_{x \subseteq D^{\mathbb{Z}}(x)}(\neg[\![x = 1]\!] \wedge \neg[\![x = 2]\!] \wedge \ \ [\![x = 3]\!]) \Big\} \equiv [1..3] \equiv \text{sols}(x)
\end{aligned}$$

Proof. Since $\text{sols}(x) \equiv D(x)$, we know that $D(x) \equiv \{d_1, ..., d_m\}$ is the codomain of $\mathcal{J}^{\mathbb{D}}_{x \in D^{\mathbb{Z}}(x)}$. The function domain of $\mathcal{J}^{\mathbb{D}}_{x \in D^{\mathbb{Z}}(x)}$ is equivalent to the solution space of the encoded `EO` constraint (see Eq. 2.10):

$$\begin{aligned}\text{sols}\big(E^{\mathbb{D}}(x \in D^{\mathbb{Z}}(x))\big) &\equiv \text{sols}\big(\texttt{EO}([\![x = d_i]\!] \in D^{\mathbb{B}}([\![x = d_i]\!]) \mid 1 \leq i \leq m])\big) \\ &\equiv \{\{([\![x = d_i]\!] \mapsto (j = i)) \mid 1 \leq i \leq m\} \mid 1 \leq j \leq m\}\end{aligned}$$

We now show $\mathcal{J}^{\mathbb{D}}_{x \in D^{\mathbb{Z}}(x)}$ maps $\text{sols}\big(E^{\mathbb{D}}(x \in D^{\mathbb{Z}}(x))\big)$ to $\text{sols}(x)$. Let $\mathcal{A}_j \in \text{sols}\big(E^{\mathbb{D}}(x \in D^{\mathbb{Z}}(x))\big)$ be the j-th solution.

$$\begin{aligned}&\Big\{\mathcal{J}^{\mathbb{D}}_{x \in D^{\mathbb{Z}}(x)}(\mathcal{A}_j) \mid 1 \leq j \leq m\Big\} \\ &\equiv \big\{d_1 \cdot \mathcal{A}_j([\![x = d_1]\!]) + \cdots + d_j \cdot \mathcal{A}_j\big([\![x = d_j]\!]\big) + \cdots + d_m \cdot \mathcal{A}_j([\![x = d_m]\!]) \mid 1 \leq j \leq m\big\} \\ &\equiv \big\{d_1 \cdot 0 + \cdots + d_{j-1} \cdot 0 + d_j \cdot 1 + d_{j+1} \cdot 0 + \cdots + d_m \cdot 0 \mid 1 \leq j \leq m\big\} \\ &\equiv \big\{d_j \mid 1 \leq j \leq m\big\} \equiv D(x) \equiv \text{sols}(x)\end{aligned}$$

□

The order encoding

The *order encoding* [41] [42] [43], $E^{\mathbb{O}}(x \in D^{\mathbb{Z}}(x))$, encodes an integer variable using primitive inequality constraint $x \geq d$. The idea of the order encoding is to let the SAT solver assign $[\![x \geq d]\!]$ to `true` in a solution of $E^{\mathbb{O}}(x \in D^{\mathbb{Z}}(x))$ iff x is assigned to $\mathcal{A}(x) \geq d$ in a solution of $x \in D^{\mathbb{Z}}(x)$. To ensure the encoding is consistent, we enforce that the encoding variables are sorted in descending order by encoding an Implication Chain (`IC`) constraint (see Eq. 2.14)

Definition 2.15 (Order encoding) The order (integer variable) encoding function $E^{\mathbb{O}}$ encodes (the CSP of) the integer variable $x \in D^{\mathbb{Z}}(x)$:

$$E^{\mathbb{O}}(x \in \{d_1, ..., d_m\}) \equiv \texttt{IC}([[\![x \geq d_m]\!], ..., [\![x \geq d_2]\!]]) \tag{2.20}$$

Theorem 2.2 The encoding function $E^{\mathbb{O}}$ (from Eq. 2.20) is correct, since there exists the following surjective interpretation function $\mathcal{J}^{\mathbb{O}}_{x \in D^{\mathbb{Z}}(x)} : \text{sols}\big(E^{\mathbb{O}}(x \in D^{\mathbb{Z}}(x))\big) \twoheadrightarrow \text{sols}(x \in D^{\mathbb{Z}}(x))$:

$$\mathcal{J}^{\mathbb{O}}_{x \in D^{\mathbb{Z}}(x)}(\mathcal{A}) \equiv d_1 + \sum_{i \equiv 2}^{m} (d_i - d_{i-1}) \cdot \mathcal{A}([\![x \geq d_i]\!]) \tag{2.21}$$

Before the proof, let us look at a concrete example of the order encoding and its interpretation.

Example 2.7 The order encoding of the CSP of $x \in [0..3]$ is:

$$\begin{aligned} E^{\mathbb{O}}(x \in [0..3]) &\equiv \texttt{IC}([[\![x \geq 3]\!], [\![x \geq 2]\!], [\![x \geq 2]\!]) \\ &\equiv ([\![x \geq 3]\!] \to [\![x \geq 2]\!]) \wedge ([\![x \geq 2]\!] \to [\![x \geq 1]\!]) \end{aligned}$$

Each of the four solutions in $\text{sols}\big(E^{\mathbb{O}}(x \in [0..3])\big)$ are mapped by $\mathcal{J}^{\mathbb{O}}_{x \in [0..3]}$ to each of the four solutions in $\text{sols}(x \in [0..3])$:

$$\begin{aligned} &\Big\{\mathcal{J}^{\mathbb{O}}_{x \in [0..3]}(\mathcal{A}) \mid \mathcal{A} \in \text{sols}\big(E^{\mathbb{O}}(x \in [0..3])\big)\Big\} \\ \equiv &\Big\{\mathcal{J}^{\mathbb{O}}_{x \in [0..3]}(\neg[\![x \geq 1]\!] \wedge \neg[\![x \geq 2]\!] \wedge \neg[\![x \geq 3]\!]), \\ &\quad \mathcal{J}^{\mathbb{O}}_{x \in [0..3]}(\ \ [\![x \geq 1]\!] \wedge \neg[\![x \geq 2]\!] \wedge \neg[\![x \geq 3]\!]), \\ &\quad \mathcal{J}^{\mathbb{O}}_{x \in [0..3]}(\ \ [\![x \geq 1]\!] \wedge \ \ [\![x \geq 2]\!] \wedge \neg[\![x \geq 3]\!]), \\ &\quad \mathcal{J}^{\mathbb{O}}_{x \in [0..3]}(\ \ [\![x \geq 1]\!] \wedge \ \ [\![x \geq 2]\!] \wedge \ \ [\![x \geq 3]\!])\Big\} \equiv [0..3] \equiv \text{sols}(x) \end{aligned}$$

Proof. Since $\text{sols}(x) \equiv D(x)$, we know that $D(x) \equiv \{d_1, ..., d_m\}$ is the codomain of $\mathcal{J}^{\mathbb{O}}_{x \in D^{\mathbb{Z}}(x)}$. The function domain of $\mathcal{J}^{\mathbb{O}}_{x \in D^{\mathbb{Z}}(x)}$ is equivalent to the solution space of the encoded `IC` constraint (see Eq. 2.15):

$$\begin{aligned} \text{sols}\big(E^{\mathbb{O}}(x \in D^{\mathbb{Z}}(x))\big) &\equiv \text{sols}([\![x \geq d_1]\!] \in D^{\mathbb{B}}([\![x \geq d_1]\!]), [\![x \geq d_2]\!] \in D^{\mathbb{B}}([\![x \geq d_2]\!]), ...) \\ &\equiv \{\{([\![x \geq d_k]\!] \mapsto (k \leq j)) \mid 2 \leq k \leq m\} \mid 1 \leq j \leq m\} \end{aligned}$$

We now show $\mathcal{J}^{\mathbb{O}}_{x \in D^{\mathbb{Z}}(x)}$ maps $\text{sols}\big(E^{\mathbb{O}}(x \in D^{\mathbb{Z}}(x))\big)$ to $\text{sols}(x)$. Let $\mathcal{A}_j \in \text{sols}\big(E^{\mathbb{O}}(x \in D^{\mathbb{Z}}(x))\big)$ be the j-th solution.

$$\begin{aligned} &\Big\{\mathcal{J}^{\mathbb{O}}_{x \in D^{\mathbb{Z}}(x)}(\mathcal{A}_j) \mid 1 \leq j \leq m\Big\} \\ \equiv &\left\{d_1 + \sum_{i \equiv 2}^{m} (d_i - d_{i-1}) \mathcal{A}_j([\![x \geq d_i]\!]) \mid 1 \leq j \leq m\right\} \\ \equiv &\big\{d_1 - d_1 \mathcal{A}_j([\![x \geq d_2]\!]) + d_2 \mathcal{A}_j([\![x \geq d_2]\!]) - d_2 \mathcal{A}_j([\![x \geq d_3]\!]) + ... \mid 1 \leq j \leq m\big\} \\ \equiv &\big\{(d_1 - d_1 \cdot 1) + \cdots + \big(d_{j-1} \cdot 1 - d_{j-1} \cdot 1\big) + d_j \cdot 1 - d_j \cdot 0 + d_{j+1} \cdot 0 - \cdots - d_m \cdot 0 \mid 1 \leq j \leq m\big\} \\ \equiv &\big\{d_j \mid 1 \leq j \leq m\big\} \equiv D(x) \equiv \text{sols}(x) \end{aligned}$$

□

The binary encoding

The binary encoding (also known as the *log* encoding) represents the domain using some pre-determined binary representation scheme. While the direct and order encodings are

unary encodings because they require $O(|D(x)|)$ variables, the binary encoding requires only $O(\log|D(x)|)$ variables. The binary encoding was first used to encode specific graph-related problems (e.g. the Hamiltonian circuit, graph colouring) [44], then planning [45]. Later, further improvements were made on the encoding of graph colouring problems [15].

Various binary representation schemes exist, which determine various aspects of the resulting binary encoding. We will present an abstract binary encoding using a given scheme R. First, the scheme defines a function $\mathrm{bit}^R(c,k)$ which yields `true` if the k-th least significant bit is `1` in the binary representation of constant c, and `false` if it is `0`. The least significant bit is $\mathrm{bit}^R(c,0)$. The idea is to let the SAT solver assign an encoding variable $[\![\mathrm{bit}^R(x,k)]\!]$ to `true` in a solution of $E^{\mathbb{B}}(x \subseteq D^{\mathbb{Z}}(x))$ iff x is assigned to d for which $\mathrm{bit}^R(d,k)$ is `true` in a solution of $x \subseteq D^{\mathbb{Z}}(x)$.

The number of encoding variable introduced is determined by the function $\mathcal{B}^R(x)$, which depends on the domain of x and the chosen scheme R. We define the *encoding range* $\mathcal{R}^R(x)$ as the set of values which the encoding can represent. We choose $\mathcal{B}^R(x)$ so that $D(x) \subseteq \mathcal{R}^R(x)$.

Often, we will have that $D(x) \subset \mathcal{R}^R(x)$. Consequently, the encoding can be assigned values not in the domain of the encoded variables. At the highest level, the consistency constraint simply enforces x to be assigned one of its domain values using a set membership constraint, $x \in D(x)$ (enforcing $\mathrm{sols}(x \in D(x)) \equiv \{\{(x \mapsto d)\} \mid d \in D(x)\}$). The simplest encoding of this constraint prohibits each assignment outside the domain:

$$E(x \in D(x)) \equiv E(x \notin \mathcal{R}^R(x) \setminus D(x)) \equiv \bigwedge_{d \in \mathcal{R}^R(x) \setminus D(x)} (E(x \neq d)) \tag{2.22}$$

The encoding of the disequality constraints for each prohibited domain value d is as follows:

$$E(x \neq d) \equiv \bigvee_{k \in \mathcal{B}(x)} ([\![\mathrm{bit}(x,k)]\!] \neq \mathrm{bit}(d,k)) \tag{2.23}$$

Finally, the abstract binary encoding is defined as follows:

Definition 2.16 (Binary encoding) The binary (integer variable) encoding function $E^{\mathbb{B}}$ for given binary representation scheme R encodes (the CSP of) the integer variable x:

$$E^{\mathbb{B}}(x \subseteq D^{\mathbb{Z}}(x)) \equiv \langle \{[\![\mathrm{bit}^R(x,k)]\!] \mid 0 \leq k < \mathcal{B}^R(x)\}, D^{\mathbb{B}}, \{x \in D(x)\} \rangle \tag{2.24}$$

The unsigned binary encoding

The simplest type of binary encoding uses the *unsigned* binary representation scheme $R \equiv \texttt{0}$. It has $\mathcal{B}^{\texttt{0}}(x) \equiv \lfloor \log(\text{ub}(x)) \rfloor + 1$ and $\mathcal{R}^{\texttt{0}}(x) \equiv \left[0..2^{\mathcal{B}^{\texttt{0}}(x)} - 1\right]$. We can define $\text{bit}^{\texttt{0}}(c, k)$ mathematically, or in C notation where `c` and `k` are of a sufficiently large unsigned integer type, as:

$$\text{bit}^{\texttt{0}}(c, k) \equiv \frac{c}{2^k} \bmod 2 \equiv \texttt{c>>k \& 1}$$

We can now prove that the unsigned binary encoding is a correct encoding of $x \subseteq D^{\mathbb{Z}}(x)$ by introducing its interpretation function, $\mathcal{I}^{\mathbb{B}^{\texttt{0}}}$.

Theorem 2.3 The encoding function $E^{\mathbb{B}}$ (from Eq. 2.24) is correct, since there exists the following surjective interpretation function $\mathcal{I}^{\mathbb{B}}_{x \subseteq D^{\mathbb{Z}}(x)} : \text{sols}(E^{\mathbb{B}}(x \subseteq D^{\mathbb{Z}}(x))) \twoheadrightarrow \text{sols}(x \subseteq D^{\mathbb{Z}}(x))$:

$$\mathcal{I}^{\mathbb{B}}_{x \subseteq D^{\mathbb{Z}}(x)}(\mathcal{A}) \equiv \sum_{k \equiv 0}^{\mathcal{B}^{\texttt{0}}(x)-1} 2^k \cdot \mathcal{A}([\![\text{bit}^{\texttt{0}}(x, k)]\!]) \tag{2.25}$$

We will not give a further proof of the correctness of the binary encoding, as it relies on well-established principals of the binary encoding, its interpretation (as a sum of powers-of-two coefficients) and the correctness of the encoding of $x \in D(x)$ from Eq. 2.22. We will show a concrete example of the unsigned binary encoding and its interpretation.

Example 2.8 The unsigned binary encoding of the CSP of $x \subseteq [0..2]$ is:

$$E^{\mathbb{B}}(x \subseteq [0..2]) \equiv x \in [0..2] \equiv (\neg[\![\text{bit}(x, 0)]\!] \vee \neg[\![\text{bit}(x, 1)]\!])$$

Each of the three solutions in $\text{sols}(E^{\mathbb{B}}(x \subseteq [0..2]))$ are mapped by $\mathcal{I}^{\mathbb{B}^{\texttt{0}}}_{x \subseteq [0..2]}$ to each of the three solutions in $\text{sols}(x \subseteq [0..2])$:

$$\begin{aligned}
&\left\{\mathcal{I}^{\mathbb{B}^{\texttt{0}}}_{x \subseteq [0..2]}(\mathcal{A}) \mid \mathcal{A} \in \text{sols}(E^{\mathbb{B}}(x \subseteq [0..2]))\right\} \\
\equiv &\Big\{\mathcal{I}^{\mathbb{B}^{\texttt{0}}}_{x \subseteq [0..2]}(\neg[\![\text{bit}(x, 0)]\!], \neg[\![\text{bit}(x, 1)]\!]), \\
&\quad \mathcal{I}^{\mathbb{B}^{\texttt{0}}}_{x \subseteq [0..2]}(\ \ [\![\text{bit}(x, 0)]\!], \neg[\![\text{bit}(x, 1)]\!]), \\
&\quad \mathcal{I}^{\mathbb{B}^{\texttt{0}}}_{x \subseteq [0..2]}(\neg[\![\text{bit}(x, 0)]\!], \ \ [\![\text{bit}(x, 1)]\!])\Big\} \equiv [0..2] \equiv \text{sols}(x)
\end{aligned}$$

Signed binary encodings

The unsigned binary encoding cannot represent negative domains since $\mathcal{R}^{\mathtt{0}}(x) \subset \mathbb{N}$. For this, we need some kind of *signed* binary representation scheme for which $\mathcal{R}^{\mathtt{0}}(x) \subset \mathbb{Z}$.

The *offset* binary scheme, $R \equiv K$, is a signed scheme which is a natural extension of the unsigned scheme, $R \equiv \mathtt{0}$. The idea is to represent the $x + K$ for constant offset value K using the unsigned binary representation: $\mathrm{bit}^K(c, k) \equiv \mathrm{bit}^{\mathtt{0}}(c - K, k)$. When all bits are assigned `false`, x is assigned K. For an offset binary scheme, the required bits, encoding range and interpretation function change accordingly:

$$\begin{aligned}
\mathcal{B}^K(x) &\equiv \lfloor \log(\mathrm{ub}(x) - K) \rfloor + 1 \equiv \mathcal{B}^{\mathtt{0}}(x - K) \\
\mathcal{R}^K(x) &\equiv \left[K..2^{\mathcal{B}^K(x)} - 1 + K\right] \equiv \mathcal{R}^{\mathtt{0}}(x) + [K..K] \\
\mathcal{I}^{\mathbb{B}^K}(\mathcal{A}) &\equiv K + \sum_{k \equiv 0}^{\mathcal{B}^K(x)-1} 2^k \cdot \mathcal{A}(\llbracket \mathrm{bit}^K(x, k) \rrbracket) \equiv K + \mathcal{I}^{\mathbb{B}^{\mathtt{0}}}(\mathcal{A})
\end{aligned} \tag{2.26.1}$$

Note that the unsigned binary scheme is an offset binary scheme where $K \equiv 0$. We can represent any finite integer domain by choosing an offset $K \leq \mathrm{lb}(x)$. Usually, we set $K \equiv \mathrm{lb}(x)$, in which case x is assigned its lower bound iff all $\llbracket \mathrm{bit}(x, d) \rrbracket$ are false.

Another well-known signed binary representation scheme is the *two's complement* binary scheme $R \equiv \mathtt{T}_B$. Unlike the other schemes so far, we need to know the number of bits B to define its function $\mathrm{bit}^{\mathtt{T}_B}$. Then, two's complement is equivalent to offset binary where $K \equiv -2^B$ binary with a negated most significant bit (also known as the *sign* bit in signed schemes).

$$\mathrm{bit}^{\mathtt{T}_B}(c, k) \equiv \begin{cases} \mathrm{bit}^{-2^B}(c, k) & \text{if } k < B - 1 \\ \neg \mathrm{bit}^{-2^B}(c, k) & \text{else} \end{cases}$$

When encoding x, we can set a suitable value for B by determining the number of required bits $B \geq \mathcal{B}^{\mathtt{T}}(x)$:

$$\mathcal{B}^{\mathtt{T}}(x) \equiv \max(\lceil \log(\max(1, -\mathrm{lb}(x))) \rceil, \lceil \log(\max(1, \mathrm{ub}(x) + 1)) \rceil) + (\mathrm{lb}(x) < 0)$$

We can see how the lower and upper bound of x impact the number of required bits. Furthermore, for $\mathrm{lb}(x) \geq 0$, we can omit the sign bit. Arguably, this function does not yield the *minimal* number of required bits for two's complement, as for $\mathrm{lb}(x) < \mathrm{ub}(x) < 0$, some number of bits between the least significant bits and the sign bit are always false. We leave out this optimization for now, as we do for optimizations that might result from domain gaps. The encoding range and interpretation follow from the offset scheme's definitions: $\mathcal{R}^{\mathtt{T}_B}(x) \equiv \mathcal{R}^{-2^B}(x)$ and $\mathcal{I}^{\mathbb{B}^{\mathtt{T}_B}}(x) \equiv \mathcal{I}^{\mathbb{B}^{-2^B}}(x)$.

Finally, the sign-and-magnitude scheme $R \equiv \pm$ uses a different system where the most significant bit $\text{bit}^{\pm}(c, \mathcal{B}^{\pm}(x) - 1)$ signifies whether the representation is to be interpreted positively or negatively. For $\mathcal{A} \in \mathcal{A}^*\left(E^{\mathbb{B}^{\pm}}(x)\right)$, let $s \equiv [\![\text{bit}^{\pm}(x, \mathcal{B}^{\pm}(x) - 1)]\!]$ be the sign bit and $u \equiv \mathcal{I}^{\mathbb{B}^0}\left(\mathcal{A}|_{\text{vars}\left(E^{\mathbb{B}^{\pm}}(x) \setminus \{s\}\right)}\right)$ the unsigned interpretation of the assignment of all other bits. Then, $\mathcal{I}^{\mathbb{B}^{\pm}}\left(\mathcal{A}(E^{\mathbb{B}}(x))\right)$ is u if $\mathcal{A}(s)$ or $-u$ if $\neg\mathcal{A}(s)$. Unlike the other schemes, the sign-and-magnitude scheme is not a special case of the offset binary scheme.

Extensions

We extend our semantic brackets notation to express other relations on $x \subseteq \{d_1, ..., d_m\}$ in terms of the encoding variables:

$$[\![x = d]\!] \equiv \texttt{false}, d \notin D(x) \tag{2.27.1}$$

$$[\![x \geq d]\!] \equiv \texttt{true}, d \leq d_1 \tag{2.27.2}$$

$$[\![x \geq d]\!] \equiv [\![x \geq d_{i+1}]\!], d_i < d \leq d_{i+1} \tag{2.27.3}$$

$$[\![x \geq d]\!] \equiv \texttt{false}, d > d_m \tag{2.27.4}$$

$$[\![x > d]\!] \equiv [\![x \geq d + 1]\!] \tag{2.27.5}$$

$$[\![x < d]\!] \equiv \neg[\![x \geq d]\!], [\![x \leq d]\!] \equiv \neg[\![x \geq d + 1]\!] \tag{2.27.6}$$

$$[\![\text{bit}^K(x, k)]\!] \equiv [\![\text{bit}^{\pm}(x, k)]\!] \equiv \texttt{false}, k \geq \mathcal{B}^K(x) \tag{2.27.7}$$

$$[\![\text{bit}^{\mathtt{T}_B}(x, k)]\!] \equiv [\![\text{bit}^{\mathtt{T}_B}(x, k - 1)]\!], k \geq B \tag{2.27.8}$$

$$[\![\text{bit}^R(x, k)]\!] \equiv \texttt{undefined}, k < 0 \tag{2.27.9}$$

The `undefined` output of Eq. 2.27.9 means we will never use this function for this input.

Encoding interpretation example

The following is a table illustrating the interpretation of the solution of each fundamental encoding.

	$E^{\mathbb{D}}(x)$				$E^{\mathbb{O}}(x)$			$E^{\mathbb{B}}(x)$	
$\mathcal{A}(x)$	$[\![x = 0]\!]$	$[\![x = 1]\!]$	$[\![x = 2]\!]$	$[\![x = 3]\!]$	$[\![x \geq 1]\!]$	$[\![x \geq 2]\!]$	$[\![x \geq 3]\!]$	$[\![\text{bit}^0(x, 1)]\!]$	$[\![\text{bit}^0(x, 0)]\!]$
0	1	0	0	0	0	0	0	0	0
1	0	1	0	0	1	0	0	0	1
2	0	0	1	0	1	1	0	1	0
3	0	0	0	1	1	1	1	1	1

Tab. 2.1: Solutions for $x \subseteq [0..3]$ and corresponding solutions of the encoding variables of $E^{\mathbb{D}}(x)$, $E^{\mathbb{O}}(x)$ and $E^{\mathbb{B}}(x)$

2.3.4 Encoding extensional constraints and general CSPSs

We have seen how to encode a CSP of a single integer using integer variable encoding $\mathbb{E}$. Now, we can use integer encoding $\mathbb{E}$ to encode a single constraint CSP of $c(x_1, ..., x_r)$ as follows:

$$E^{\mathbb{E}}(c) \equiv E^{\mathbb{E}}(c(x_1, ..., x_r)) \equiv E^{\mathbb{E}}(x_1) \wedge \cdots \wedge E^{\mathbb{E}}(x_r) \wedge Q_c \tag{2.28}$$

The integer encodings are conjoined with the constraint's *inner encoding*, Q_c. The inner encoding is constructed so that the following interpretation function is a surjective function:

$$\mathcal{I}_c^{\mathbb{E}} : \text{sols}(E^{\mathbb{E}}(c)) \twoheadrightarrow \text{sols}(c) \equiv \left\{ \left(x \mapsto \mathcal{I}_x^{\mathbb{E}}\left(\mathcal{A}|_{\text{vars}(E^{\mathbb{E}}(x))}\right)\right) \mid x \in \text{vars}(c) \right\} \tag{2.29}$$

Example 2.9 Encoding $c \equiv x \subseteq [1..3] \neq 2$ (for which $\text{sols}(c) \equiv \{(x \mapsto 1), (x \mapsto 3)\}$) using the direct encoding yields an encoding:

$$E^{\mathbb{D}}(c) \equiv E^{\mathbb{D}}(x \subseteq [1..3] \neq 2) \equiv E^{\mathbb{D}}(x) \wedge Q_{x\neq 2}$$

By choosing $Q_{x\neq 2} \equiv [\![x \neq 2]\!]$, we get an encoding which we then simplify (using unit propagation):

$$(\llbracket x = 1 \rrbracket \rightarrow \llbracket x \neq 2 \rrbracket) \wedge (\llbracket x = 1 \rrbracket \rightarrow \llbracket x \neq 3 \rrbracket) \wedge (\llbracket x = 2 \rrbracket \rightarrow \llbracket x \neq 3 \rrbracket) \wedge \llbracket x \neq 2 \rrbracket \implies \llbracket x \neq 2 \rrbracket$$
$$(\llbracket x = 1 \rrbracket \rightarrow \llbracket x \neq 3 \rrbracket) \quad \textit{fixp.}$$

Note that $Q_{x\neq 2}$ by itself is not an encoding of c. However, when $Q_{x\neq 2}$ is conjoined with $E^{\mathbb{D}}(x)$, it makes $E^{\mathbb{D}}(c)$ an encoding of c since the interpretation function Eq. 2.29 is surjective:

$$\begin{aligned}
&\{\mathcal{I}_c^{\mathbb{D}}(\mathcal{A}) \mid \mathcal{A} \in \text{sols}(E^{\mathbb{D}}(c))\} \\
&\equiv \{\mathcal{I}_c^{\mathbb{E}}(\llbracket x = 1 \rrbracket \wedge \llbracket x \neq 2 \rrbracket \wedge \llbracket x \neq 3 \rrbracket), \mathcal{I}_c^{\mathbb{E}}(\llbracket x \neq 1 \rrbracket \wedge \llbracket x \neq 2 \rrbracket \wedge \llbracket x = 3 \rrbracket)\} \\
&\equiv \{\{(x \mapsto \mathcal{I}_x^{\mathbb{E}}(\llbracket x = 1 \rrbracket \wedge \llbracket x \neq 2 \rrbracket \wedge \llbracket x \neq 3 \rrbracket))\}, \{(x \mapsto \mathcal{I}_x^{\mathbb{E}}(\llbracket x \neq 1 \rrbracket \wedge \llbracket x \neq 2 \rrbracket \wedge \llbracket x = 3 \rrbracket))\}\}) \\
&\equiv \{\{(x \mapsto 1)\}, \{(x \mapsto 3)\}\}) \equiv \text{sols}(c)
\end{aligned}$$

There are two details to finalize before we can encode and interpret a general CSP P: Boolean variables and multiple constraints.

- To handle a mix of integer and Boolean variables, we extend the encoding and interpretation on the CSP $b \subseteq D^{\mathbb{B}}(b)$ of a single, unconstrained Boolean variable to simply pass it through, i.e. $E^{\mathbb{E}}(x \subseteq D^{\mathbb{B}}(x)) \equiv x \subseteq D^{\mathbb{B}}(x)$ and $\mathcal{I}^{\mathbb{E}}_{x \subseteq D^{\mathbb{B}}(x)}$ are identity functions.

- If a CSP P has multiple constraints, they are split up (see Eq. 2.3), encoded, and conjoined by $E^{\mathbb{E}}(P) \equiv \bigwedge_{c \in \text{cons}(P)} E^{\mathbb{E}}(c)$. Constraints which share one or more integer variables duplicate consistency constraints, but these are removed when conjoining the encodings.

With that, an encoding of a general CSP P is correct if every constraint encoding makes the following interpretation function surjective:

$$\mathcal{I}_P^{\mathbb{E}} : \text{sols}\big(E^{\mathbb{E}}(P)\big) \twoheadrightarrow \text{sols}(P) \equiv \left\{ \left(x \mapsto \mathcal{I}_x^{\mathbb{E}}\left(\mathcal{A}|_{\text{vars}(E^{\mathbb{E}}(x))}\right)\right) \mid x \in \text{vars}(P) \right\} \tag{2.30}$$

Encoding extensional CSPs

In this section, we will show how extensional CSPs are historically encoded, some of which can be described using the definitions of the different integer variable encoding we have seen in Sec. 2.3.3. Consider an extensional, binary CSP where constraints list disallowed assignments. A potential constraint is $x \neq 1 \wedge y \neq 2$, prohibiting the assignment of x and y to 1 and 2, respectively.

One of the first encoding method (folklore) of an extensional CSP uses the direct encoding in conjunction with *conflict clauses*. A *conflict* clause enforces the constraint by encoding $\neg[\![x = 1]\!] \vee \neg[\![y = 2]\!]$ (similar to Ex. 2.9). Consequently, the number of (binary) clauses is equal to the number of conflicts, which is $\sum_{c \in \text{cons}(P)} \prod_{x \in \text{vars}(c)} (|D(x)|)$. More generally:

$$\begin{aligned} \mathcal{A}(x) \notin \text{sols}(c) \equiv \neg(\mathcal{A}(x) \in \text{sols}(c)) &\equiv \neg((x = d \wedge y = e) \in \text{sols}(c)) \\ &\equiv (x \neq d \vee y \neq e) \\ &\equiv ([\![x \neq d]\!] \vee [\![y \neq e]\!]) \end{aligned} \tag{2.31.1}$$

Alternatively, a binary encoding of the conflict in Eq. 2.31.1 requires more literals per clause [15], [44], [45], which we have seen as encoding $E(x \neq d)$ in Eq. 2.23.

This approach can be used to encode the consistency constraints in Eq. 2.22. As we can see, the encoding of a unary constraint of one variable is used in more general constraints (in this case, the encoding of binary constraints by a decomposition and encoding of unary constraints). We will complete this picture in Sec. 3.3.3.

Alternatively, for binary CSPs, *support* clauses [46], based on earlier work [47], can encode support information. That is, for every binary constraint (x, y), we can extract the set of support values $S \subseteq D(y)$ (see Def. 2.7) for $x = d$. Once $x = d$ loses its support, the consequence is that $x \neq d$. This support information can be encoded with the direct encoding as a clause:

$$\left(\neg[\![x = d]\!] \vee \bigvee_{e \in S} ([\![y = e]\!]) \right)$$

The strength of the support encoding is that by applying unit propagation on the encoding variables until fixpoint, domain consistency is established over the encoded variable. In this case, the clauses remove (each in linear time) any value $[\![x = d]\!]$ which cannot be domain consistent if $x = d$. Once a potential for x, say d, loses all its support values, its support clause is unit. In other words, a domain consistent propagator would remove the same values. This is an advantage over the conflict encoding, which maintains a weaker type of consistency called Forward Checking [40].

Other encodings

Other integer variable encodings of extensional CSPs exist, including:

- *Hierarchical encodings* [48], *Representative-sparse* [49], *representative-order* [50] and *compact order* encoding [43]: these approaches all subdivide the domains in a (potentially multi-level) hierarchical structure into groups, where one variable dictates whether any of the (direct/ order) variables in the group should be activated.
- *Log-support encoding* [51]: uses the binary encoding as a base, but some set of suitable conflict clauses can be replaced by support clauses. The authors also evaluate Gray code as an alternative binary representation scheme, showing it outperforms binary in its considered benchmarks.
- Some encoding techniques adapt the consistency constraint:
  - Through *binary transform* [52], we can sometimes omit consistency constraints of direct or binary encodings.
  - In the *multivalued* encoding, the `AMO` constraints from Eq. 2.12 are omitted to let the direct encoding represent multiple solutions, which can be easily extracted one-by-one. The *maximal* encoding [53] also omits the `AMO` clause but does add *maximality* clauses that speed up all-solution search by breaking dominance.

2.3.5 Encoding intensional constraints

This thesis will deal primarily with intensional constraints which do not list conflicts explicitly. Instead, the extensional conflicts are implicitly generated using some formula. For instance, the $\texttt{AMO}([b_1, ..., b_r])$ constraint ensures at-most-one of its Boolean variables $b_1, ..., b_r$ is true. We can express this constraint succinctly as $b_1 + \cdots + b_r \leq 1$. In the direct encoding's consistency constraint, Eq. 2.12, we have seen the pairwise encoding of the `AMO` constraint. Other `AMO` encodings include the *ladder* encoding [54], *binary* [52] (later *bitwise* [55]) encoding, the *bimander* [16] encoding, and more.

To linger on the ladder encoding, it defines auxiliary variables $b'_1, ..., b'_r$ and encodes the `IC` constraint (seen in the order encoding's consistency constraint, Eq. 2.20):

$$\texttt{IC}([b'_r, ..., b'_1])$$

Then, so-called *channelling clauses* between the principal b_i and auxiliary b'_i variables are added:

$$\bigwedge_{2 \leq i \leq r} (b_i \leftrightarrow (b'_{i-1} \wedge b'_i))$$

This is an example of *implicitly* introducing both direct and order encodings ($b_i \equiv [\![x = i]\!]$ and $b'_i \equiv [\![x \geq i]\!]$), channelling, and making both consistent via the consistency constraint of one (in this case, a consistent order encoding). This concept will be explored thoroughly in Chp. 3, Sec. 3.3.3.

The `AMO` constraint can be generalized to other common constraints:

- The *cardinality* constraint generalizes the `AMO` constraint with $b_1 + \cdots + b_r \leq k$ for integer constant k. These encoding have been surveyed along with `AMO` constraints in [56], [57].
- The Pseudo-Boolean (PB) constraint generalizes the cardinality constraint with $c_1 b_1 + \cdots + c_r b_r \leq k$ for integer constants $c_1, ..., c_r$.
- The Integer Linear (IL) constraint generalizes the PB constraint with $c_1 x_1 + \cdots + c_r x_r \leq k$ for integer variables $x_1, ..., x_r$.

By changing $\leq$ to a general comparator $\# \in \{<, \leq, =, \neq, \geq, >\}$ we encounter other variants. For `AMO`, we'd encounter the `EO` and `ALO` constraints. All IL constraints (and its special cases presented here), can be normalized [58] to the comparator $\leq$, and $k \geq 1$, and $1 \leq q_i \leq k, \forall 1 \leq i \leq n$. Replacing $=$ by a $\leq$- and a $\geq$-PB constraint does lose propagation strength. Otherwise, the normalization results in trivial SAT or UNSAT constraints, e.g. if $c_i \geq 1, k < 0$.

The PB constraint has been studied extensively in the context of SAT encodings. Approaches include: Generalized Totalizer (GT) [59], Generalized n-Level Modulo Totalizer (GMTO) [60], Sequential Weight Counter (SWC) [61], Binary Decision Diagram (BDD) [58], Ripple Carry Adder (RCA) [62], Sorting Network (SN) [58], and Local and Global Polynomial Watchdog (LPW/GPW) [63]. In [64], most methods are surveyed, and some are improved, or generalized for IL constraints. Improvements on SN have been explored [65]. In Chp. 5, we give the full definitions of some of these PB encodings.

No PB encoding has surfaced as the best in all situations. ML approaches can successfully find good choices, but there remains a sizable gap with the virtual best choice for every instance [66].

Global constraints

Another extensive class of constraints are the *global* constraints. Often these constraints capture some combinatorial sub-structure which occur frequently in CSPs. The canonical example of a global constraint is `all_different`$([x_1 \in D^{\mathbb{Z}}(x_1), ..., x_r \in D^{\mathbb{Z}}(x_r)])$, which requires its input variables to take different values, i.e. $x_1 \neq \cdots \neq x_r$. Because of their importance,

individual papers are published on the SAT encoding of specific global constraints, including `all_different` [54] [67].

Let the function $D^{\cup}$ yield the union of the domains of a set of integer variables X with $D^{\cup}(X) = \bigcup_{x \in X} D(x)$. The `all_different` constraint holds iff for each value d in $D^{\cup}(x_1, ..., x_r)$, at most one variable x_i is equal to it, i.e. $x_i = d$ is true at most once. The constraint $x_i = d$ is naturally encoded with the direct encoding, so we conjoin $E^{\mathbb{D}}(x_i)$ for every input variable of `all_different`.

$$E(\texttt{all\_different}([x_1, ..., x_r])) \equiv E^{\mathbb{D}}(x_1) \wedge \cdots \wedge E^{\mathbb{D}}(x_r) \wedge \tag{2.32.1}$$

$$\bigwedge_{d \in D^{\cup}(x_1,...,x_r)} \texttt{AMO}([x_1 = d, ..., x_r = d]) \tag{2.32.2}$$

Other examples of global constraints include `circuit`, `disjunctive` and `cumulative` [68].

2.4 Constraint modelling languages

In this section, we discuss implementations of CSPs, which are known as *constraint modelling languages*. In section Sec. 2.4.1, we cover constraint modelling languages for CSP, in particular MiniZinc, and how SAT problems are specified. In section Sec. 2.4.2, we show how a CSP model of Sudoku can be implemented as a constraint model in MiniZinc, and encoded to SAT by a SAT library.

This section covers a) Modelling CSP and specifying SAT problems, b) Solving a Sudoku problem using MiniZinc and SAT, and c) Other CO solving technologies.

2.4.1 Modelling CSP and specifying SAT problems

Commonly known programming languages, such as C and Python, follow the *imperative* programming paradigm. An imperative program expresses a set of instructions that are executed step by step to solve a problem. The CSP, however, only declares a problem, but not the steps to solving it. A constraint modelling language implements a CSP as a *constraint model*. Such languages thus follow the *declarative* programming paradigm. The constraint modelling language usually does not solve the constraint model. It transforms the constraint model to be solved by some target *solver*.

Examples of constraint modelling languages are `AMPL` [69], `xcsp3` [70], `Essence` [71], `OPL` [72], Python (using the `CPMpy` library [73]), and `MiniZinc` [3]. Additionally, languages based on logic programming such as `picat` [74] and `SICStus Prolog` [72] can be used to model and solve CSPs.

MiniZinc and FlatZinc

MiniZinc is used extensively in this thesis. It offers a high-level CP view of the CSP, allowing for custom predicates and offers a large library of global constraints that capture common patterns.

It is solver-independent, and in its 2024 annual solver competition (the MiniZinc Challenge [5], [6]), 16 solvers participated. MiniZinc models can be solved by a wide range of supported solver technologies, including CP, MIP, SAT, Lazy Clause Generation (LCG), and local search solvers.

Internally, MiniZinc compiles to a simpler constraint modelling language, FlatZinc, a subset language of MiniZinc. It consists of only FlatZinc built-ins (such as `int_ne(x,y)` for $x \neq y$) which every solver must support (to solve models in which they occur), and the constraints which a solver declares to support. Unsupported constraints which are not FlatZinc built-ins, i.e. global constraints, are decomposed into FlatZinc built-ins. For example, consider a sample CSP, `all_different`$([x_1 \in [1..3], x_2 \in [1..3], x_3 \in [1..3]])$, implemented in MiniZinc in Lst. 2.1.

```
1  include "globals.mzn";
2  array[1..3] of var 1..3: xs;
3  constraint all_different(xs);
```

Lst. 2.1: A MiniZinc model for `all_different`$([x_1 \in [1..3], x_2 \in [1..3], x_3 \in [1..3]])$ (`./decomp.mzn`)

MiniZinc will compile this model differently depending on the target solver. For the `Gecode` solver, which does support the `all_different` constraint, the resulting FlatZinc includes a `fzn_all_different_int` constraint.

```
1  predicate fzn_all_different_int(array [int] of var int: x);
2  var 1..3: X_INTRODUCED_16_;
3  var 1..3: X_INTRODUCED_17_;
4  var 1..3: X_INTRODUCED_18_;
5  array [1..3] of var int: xs ::output_array([1..3]) =
[X_INTRODUCED_16_,X_INTRODUCED_17_,X_INTRODUCED_18_];
6  constraint fzn_all_different_int(xs);
7  solve  satisfy;
```

Lst. 2.2: `./decomp.gecode.fzn`

However, if the solver does not support the `all_different` constraint, its decomposition from the MiniZinc's standard library (`std`) is used instead to produce a model of exclusively FlatZinc built-ins:

```
%-----------------------------------------------------------%
% Constrains the array of objects 'x' to be all different.
%-----------------------------------------------------------%

predicate fzn_all_different_int(array[int] of var int: x) =
    forall(i,j in index_set(x) where i < j) ( x[i] != x[j] );
```

Lst. 2.3: `./std/fzn_all_different_int.mzn`

The result for solvers which do not support the `all_different` is as follows:

```
array [1..2] of int: X_INTRODUCED_19_ = [1,-1];
var 1..3: X_INTRODUCED_16_;
var 1..3: X_INTRODUCED_17_;
var 1..3: X_INTRODUCED_18_;
array false1..3] of var int: xs ::output_array([1..3]) =
[X_INTRODUCED_16_,X_INTRODUCED_17_,X_INTRODUCED_18_];
constraint int_lin_ne(X_INTRODUCED_19_,[X_INTRODUCED_16_,X_INTRODUCED_17_],0);
constraint int_lin_ne(X_INTRODUCED_19_,[X_INTRODUCED_16_,X_INTRODUCED_18_],0);
constraint int_lin_ne(X_INTRODUCED_19_,[X_INTRODUCED_17_,X_INTRODUCED_18_],0);
solve  satisfy;
```

Lst. 2.4: `./decomp.std.fzn`

Rather than a propagator, solvers can also provide their own decomposition in a library, such as MIP solvers which use MiniZinc's linearization library (`linear`):

```
include "domain_encodings.mzn";

predicate fzn_all_different_int(array[int] of var int: x) =
  if length(x)<=1 then
    true
  else
    let {
      array[int,int] of var 0..1: x_eq_d = eq_encode(x)
    } in forall(d in index_set_2of2(x_eq_d))(
      sum(i in index_set_1of2(x_eq_d))( x_eq_d[i,d] ) <= 1
    )
  endif;
```

Lst. 2.5: `./linear/fzn_all_different_int.mzn`

This idea of this decomposition is the same as for the SAT decomposition from Eq. 2.32. The result is a fully linearized model, suitable for MIP solvers, exclusively using integer linear `int_lin_*` constraints:

```
array [1..3] of int: X_INTRODUCED_24_ = [1,1,1];
array [1..4] of int: X_INTRODUCED_30_ = [1,2,3,-1];
var 1..3: X_INTRODUCED_16_:: is_defined_var;
var 1..3: X_INTRODUCED_17_:: is_defined_var;
var 1..3: X_INTRODUCED_18_:: is_defined_var;
var 0..1: X_INTRODUCED_19_ ::var_is_introduced ;
var 0..1: X_INTRODUCED_20_ ::var_is_introduced ;
% [...]
constraint int_lin_le(X_INTRODUCED_24_,
[X_INTRODUCED_20_,X_INTRODUCED_32_,X_INTRODUCED_40_],1);
constraint int_lin_le(X_INTRODUCED_24_,
[X_INTRODUCED_21_,X_INTRODUCED_33_,X_INTRODUCED_41_],1);
solve  satisfy;
```

Lst. 2.6: Partial decomposition (note the ellipsis: `% [...]`) (`./decomp.linear.fzn`)

DIMACS

Since SAT is a very simple type of CSP, its constraint modelling language, DIMACS [75], is a simple specification format. In DIMACS, a preamble line prefixed by `p cnf` shows the (positive) number of variables and clauses. Lines prefixed `c` are comment lines which are ignored. Each clause is a sequence of non-zero integers i ($-i$). Each integer represents one positive (negative) literal with variable number $|i|$. Each clause is separated by a zero.

For example, a DIMACS specification of the encoding $E(p \oplus q) \equiv (p \vee q) \wedge (\neg p \vee \neg q)$ is shown in Lst. 2.7.

```
1  p cnf 2 2
2  c Comment: a clause, `p or q`
3  1 2 0
4  c Another clause `not p or not q`
5  -1 -2 0
```

Lst. 2.7: `./xor.dimacs`

A SAT solver supporting DIMACS yields a status `s`-line of either `SATISFIABLE`, `UNSATISFIABLE` or `UNKNOWN` (for timeouts). If satisfiable, most SAT solvers output the solution with a `v`-line. The solution is essentially shown as a conjunction of literals. The output of `CaDiCal` is shown in Lst. 2.8. Its solution line shows `v 1 -2 0`, which translates to the solution $\{(p \mapsto \texttt{true}), (q \mapsto \texttt{false})\}$.

```
c --- [ banner ] ---------------------------------------------------------
c
c CaDiCaL SAT Solver
```

```
 c Copyright (c) 2016-2023 A. Biere, M. Fleury, N. Froleyks, K. Fazekas, F.
Pollitt
 c
 % [...]
 c
 s SATISFIABLE
 v 1 -2 0
 c
```

Lst. 2.8: Output of `CaDiCal` on input from Lst. 2.7

WDIMACS

The constraint modelling language for MaxSAT is known as WDIMACS. While similar to DIAMCS, a fairly recent update of the format omits the `p`-line, prefixes hard clauses by `h`, and soft clauses by their weight.

Whereas `CaDiCal` assigned q to `false`, we can prefer a solution where q is assigned `true` instead by adding a soft clause in Lst. 2.9, line 5, penalizing the falsification of q.

```
1 h 1 2 0
2 h -1 -2 0
3 c A penalty of 1 if q is assigned false
4 1 2 0
```

Lst. 2.9: `./xor.wdimacs`

The output of a MaxSAT solver `Open-WBO` is shown in Lst. 2.10. Given the added soft clause, it outputs a different (first) solution of $\{(p \mapsto \texttt{false}), (q \mapsto \texttt{true})\}$. The `s`-line now indicates that the solver found the optimal solution. The output format is the same as DIMACS, but with `o`-lines indicating objective values of intermediate solutions.

```
 c
 c TT-Open-WBO-Inc:   an Anytime MaxSAT Solver -- based on IntelSATSolver
 c Version:   Development after MaxSAT Evaluation 2024
 c Author:  Alexander Nadel
 % [...]
 s OPTIMUM FOUND
 o 0
 v 01
```

Lst. 2.10: Output of `Open-WBO` on input from Lst. 2.9

2.4.2 Solving a Sudoku problem using MiniZinc and SAT

In this section, we show how to encode, solve and interpret a simple MiniZinc model for a 9×9 Sudoku instance. In a Sudoku, the goal is to fill a grid with numbers such that in all 9 rows, columns, and 3×3 sub-grids, no number appears twice. Some numbers are already given as clues.

Encoding a Sudoku CSP to SAT

We will first formalize the Sudoku problem as a CSP. Let X be a set of integer variables where $x_{i,j} \in X$ represents the number at row i, column j in the Sudoku grid. Let C be a (smaller) set of integers where $c_{i,j} \in C$ is the given clue for $x_{i,j}$. Let $\texttt{row}(X, i)$, $\texttt{col}(X, i)$, and $\texttt{sub}(X, i)$ be functions which yield a list of variables in the i-th row, column and sub-grid, respectively. The `all_different` global constraint (see Sec. 2.3.5) enforces the constraint for each row, column and sub-grid.

$$
\begin{aligned}
P \equiv \langle & && X, \\
& \left\{ \left(x_{i,j} \mapsto \begin{cases} \{c_{i,j}\} & \text{if } c_{i,j} \in C \\ [1..9] & \text{else} \end{cases} \right) \mid x_{i,j} \in X \right\}, \\
& \{ && \texttt{all\_different}(\texttt{row}(X, i)) \\
& && \wedge \texttt{all\_different}(\texttt{col}(X, i)) \\
& && \wedge \texttt{all\_different}(\texttt{sub}(X, i)) \\
& && \mid 1 \leq i \leq 9 \} \rangle
\end{aligned}
$$

To encode P is to encode a series of `all_different` constraints.

$$
\begin{aligned}
& E(P) \\
\equiv & \bigwedge_{c \in \text{cons}(P)} E(c) \\
\equiv & \; E(\texttt{all\_different}(\texttt{row}(X, 1))) \wedge \cdots \wedge E(\texttt{all\_different}(\texttt{row}(X, 9))) \\
\equiv & \; \forall_{x \in \texttt{row}(X,1)} (E^{\mathbb{D}}(x)) \wedge \forall_{d \in D^{\cup}(\texttt{row}(X,1))} (\texttt{AMO}([\![x = d]\!] \mid x \in \texttt{row}(X, 1))) \wedge ... \\
\equiv & \; \forall_{x \in \texttt{row}(X,1)} (\texttt{EO}([\![x = d]\!] \mid d \in D(x))) \wedge \forall_{d \in D^{\cup}(\texttt{row}(X,1))} (\texttt{AMO}([\![x = d]\!] \mid x \in \texttt{row}(X, 1))) \wedge ... \\
\equiv & \bigwedge_{x \in \texttt{row}(X,1)} \left(\bigvee_{d \in D(x)} ([\![x = d]\!]) \wedge \bigwedge_{d_i, d_j \in D(x), i<j} (([\![x \neq d_i]\!] \vee [\![x \neq d_j]\!])) \right) \wedge \\
& \bigwedge_{d \in D^{\cup}(\texttt{row}(X,1))} \left(\bigwedge_{x_{1,i}, x_{1,j} \in \texttt{row}(X,1), i<j} (([\![x_{1,i} \neq d]\!] \vee [\![x_{1,j} \neq d]\!])) \right) \wedge ...
\end{aligned}
$$

We note some details of this encoding. First, since $|\texttt{row}(X, 1)|$ is equal to the cardinality of the union of domains, we could have enforced `EO` instead of `AMO`. This redundant constraint might

aid propagation but does not change the encoding's solution space. Second, for variables $x_{i,j} = c_{i,j}$ fixed by clue $c_{i,j} \in C$, the unary domain of $x_{i,j}$ results in a single literal encoding $[\![x_{i,j} = c_{i,j}]\!]$. This literal is immediately fixed to `true` by the unary clause $[\![x_{i,j} = c_{i,j}]\!]$ encoded by its consistency constraint. This ensures that another variable $x_{i,j'}$ in the same row cannot take $c_{i,j}$ by the clause $([\![x_{i,j} \neq c_{i,j}]\!] \vee [\![x_{i,j'} \neq c_{i,j}]\!]) \equiv [\![x_{i,j'} \neq c_{i,j}]\!]$. Similarly, $[\![x_{i,j} = d]\!] = \texttt{false}$ whenever $d \neq c_{i,j}$ by Eq. 2.27.1 will exactly satisfy all clauses $[\![x_{i,j} \neq c_{i,j}]\!] \vee [\![x_{i,j'} \neq c_{i,j}]\!]$ that should *not* propagate other variables.

Encoding Sudoku with MiniZinc

In this section, we show how to use MiniZinc to model the Sudoku CSP and solve it using a CP solver. Then, we show how we can implement a minimal quasi-library to encode, solve and interpret the Sudoku problem as SAT according to the encoding in the preceding section.

We first declare the variables as a two-dimensional array `X` of integer variables (indicated by the `var` keyword) with domain `1..9`. The Sudoku instance to solve (the given clues) is represented by another two-dimensional array. By assigning `X` to the instance, some of the integer variables of `X` are fixed (e.g. `x[1,1]=1`), while others are still unknown (e.g. `x[1,2]=_`). The index sets of `X` are inferred by the dimensions of the fixed array on the right-hand side, i.e. `X` has two index sets, `1..9` and `1..9`.

```
% `X` denotes a 2D array of square variables w/ domain 1..9
array[int, int] of var 1..9: X = [|
  1,_,_, _,4,8, _,_,_ | % `_` are the unknowns
  _,5,_, _,_,_, 9,_,_ |
  _,_,6, _,_,_, 3,_,_ |

  _,_,_, 5,7,_, 2,_,_ |
  8,_,3, _,_,_, _,_,_ |
  _,_,_, 9,_,_, _,_,_ |

  _,_,_, _,_,_, _,4,1 |
  6,7,_, _,_,_, _,_,_ |
  _,_,_, 2,_,_, _,_,_ |];
```

Lst. 2.11: The variables for `sudoku.mzn`

We can use a `forall` to generate the i-th `all_different` constraint for each of the 27 rows, columns and sub-grids. Using `forall`, the integer variables of one row, column or sub-grid are collected as the input for the `all_different`. For instance, `row`$(X, 1)$ uses a slice `X[1,..]` to construct a one-dimensional array with the variables of the first row of `X`. The `sub`(X, i) is implemented by calculating the indices of the sub-grids, which are translated to the coordinates

to slice the whole grid. A slice with two index sets yield a two-dimensional which is flattened before passed on to `all_different`.

```
% For each row, column and sub-grid, the variables are all different
constraint forall(i in 1..9)(all_different(X[i,..])); % row(i) slices `X`
constraint forall(j in 1..9)(all_different(X[..,j]));
constraint forall(i in 1..9)(
  let {                     % i=1,2,3,4,5,6,7,8,9
    int: a = ((i-1) div 3); % a=0,0,0,1,1,1,2,2,2
    int: b = ((i-1) mod 3); % b=0,1,2,0,1,2,0,1,2
  } in all_different(
    array1d( % coerce 2D to 1D array
      % i=1 -> [1..3,1..3], i=2 -> [1..3,4..6], ...
      X[(3*a)+1..3*(a+1), (3*b)+1..3*(b+1)]
    )));
```

Lst. 2.12: A MiniZinc model `sudoku.mzn` for a Sudoku instance

In MiniZinc, global constraints need to be `included`.

```
include "all_different.mzn";
```

Lst. 2.13: Loading the definition of the `all_different` constraint

In this case, the `gecode` solver implements an `all_different` propagator. If not, the standard library has a decomposition for every global constraint which can be used instead. Alternatively, we can define the `all_different` constraint ourselves by adding a `predicate` for it. A predicate in MiniZinc is a function which returns a `var bool`, and is used to define (relational) constraints. In this case, we will create a minimal SAT library for MiniZinc which constrains `all_different` using the direct encoding.

We first add a function to encode integer variables. Then, we add an `encode` function which creates and returns the direct encoding `bs` for its input argument `x`, an integer variable. The data-structure for the direct encoding variables is an array of Booleans, where the index `d` of the array maps to the array element `bs[d]`, which is the Boolean variable $[\![x = d]\!]$. In MiniZinc, the index set can be any contiguous range, allowing us to have elements indexed by the interval $[\mathrm{lb}(x)..\mathrm{ub}(x)]$. We do need to ensure that gaps in the domain are set to `false` elements in the array. We also add the `ALO` and `AMO` consistency constraints.

The next line attaches a "reverse mapper" for `x`, instructing MiniZinc to assign `x` by calling the interpretation function `interpret(bs)`. The existence of the reverse mapper is indicated with an *annotation*, `::is_reverse_map`. The `interpret` function only needs to be implemented for a

fixed input, since it is only called during the output phase where all variables are fixed. We implement the interpretation according to Eq. 2.19.

```
% Encode integer variable `x` using the direct encoding
function array[int] of var bool: encode(var int: x) =
  let {
    % Create a mapping from domain value to Boolean variable
    array[int] of var bool: bs = [
      d: % domain value d maps to:
        if d in dom(x) then % a new Boolean variable, `[x=d]`
          let { var bool: b::promise_total } in b
        else  % but `[x=d]=false` iff `d` is in a domain gap
          false
        endif
      | d in lb(x)..ub(x)
    ];
    constraint EO(bs); % encode the exactly-one consistency constraint
    constraint (x = interpret(bs))::is_reverse_map; % attach interpretation
function
  } in bs;

% Interpret assignment to encoding as assignment to integer
function var int: interpret(array[int] of var bool: bs)::output_only;
function int: interpret(array[int] of bool: bs) =
  sum(d in index_set(bs))(d*bs[d]);
```

Lst. 2.14: A direct integer variable encoding function in MiniZinc

We can now implement the encodings for `all_different`, `EO`, `AMO` and `ALO` as conjunctions of `clause` constraints. MiniZinc implements Common Subexpression Elimination (CSE), thus calling `encode` again on the same input variables does *not* create a new encoding variables or constraints, but allows us to access the same encoding variables from anywhere.

```
% Encode the `all_different` constraint to SAT
predicate all_different(array[int] of var int: xs) =
  % Encode all integer variables
  forall(x in xs)(let { array[int] of var bool: bs = encode(x) } in true) /\
  % Encode that every unique domain value is true at most once
  forall(d in dom_array(xs))(
    AMO([ encode(x)[d]::maybe_partial | x in xs ])
  );

% Encode that exactly-one Boolean variable is true
predicate EO(array[int] of var bool: bs) = ALO(bs) /\ AMO(bs);
```

```
12
13  % Encode that at-least-one Boolean variable is true using the clause `bs[1] v
bs[2] v ..`
14  predicate ALO(array[int] of var bool: bs) = exists(bs); % `exists` posts
disjunction
15
16  % Encode that at-most-one Boolean variable is true using pairwise encoding
17  predicate AMO(array[int] of var bool: bs) =
18    forall(i, j in index_set(bs) where i < j)(
19      not bs[i] v not bs[j]
20    );
```

Lst. 2.15: Various constraint encoding functions in MiniZinc

2.4.3 Other CO solving technologies

This section briefly covers other solving technologies that can be used to solve CSPs.

Linear and Mixed-Integer Programming

Linear programming solver solve Linear Programming (LP) problems, which are COPs with only linear equality and inequality constraints, and a linear objective function. In an LP P, the variables $x \in \text{vars}(P)$ are real-valued, i.e. $D(x) \subseteq \mathbb{R}$. A similar encoding framework (such as MiniZinc's linearization library [76]) will need to be used to encode interesting problem as linear programs. A linear program can be solved in polynomial time. However, if we add the constraint that some or all variables have integer domains, in which case the problem is called an Integer Linear Programming (ILP) or MIP problem respectively, it becomes NP-hard. MIP solvers, both proprietary such as Gurobi [77] and open-source such as SCIP [78], are used extensively in industrial applications.

Lazy Clause Generation

The CP solvers often find themselves resolving the same unsatisfiable sub-problem which caused failure before. Lazy Clause Generation (LCG) [79] is an approach to learn from failure during the search. It does so by incrementally building up a SAT encoding of the problem. After every search decision and propagation, literals are added to a clause *explaining* the steps.

For example, consider the constraint $x \in [0..7] + y \in [0..7] + \leq z \in [0..14]$. If a search decision is made to update $D(y)$ to $[5..7]$, then a (bounds consistent) constraint propagator should infer $D(z) = [5..14]$. The clause explaining the inference is $[\![y \geq 5]\!] \rightarrow [\![z \geq 5]\!]$. At some point, the incremental SAT solver can be executed to solve the partially represented problem. If it encounters failure, backjumping can be used to return to earlier parts of the search. This incremental encoding is in contrast to the one-shot encoding, which is the main topic of this thesis, Successful LCG solvers include Chuffed [80], CPX[81], Geas [82] and Google's OR-Tools/ CP-SAT (`cp-sat`) [7]

Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) [83] is a generalization of SAT, extending the Boolean formula with numerical values or even higher-level structures such as bit vectors. While SMT solving uses Boolean logic using a SAT solver back-end, non-Boolean expressions can be understood and solved by providing programs known as *theories*. Thus, SMT veers back into the territory of CSPs and CP, with theories taking the role of constraint propagators. However, due to its more formal background logic, SMT sees more adoption in automated theorem proving and program verification (e.g. using the well-known Z3 theorem prover [84]).

— Chapter 3 —

Coupling Different Integer Encodings for SAT

This chapter presents MiniZinc-SAT (`mzn-sat`), a Boolean Satisfiability (SAT) library for MiniZinc which allows the modeller to choose the encoding of individual integer variables. This choice is critical to the SAT solver performance, however, in most other SAT encoding frameworks, it is fixed for all integers. The underlying reason is that changing the encoding of the integer variable requires changing the encoding of its constraints. To resolve this, this chapter proposes the encoding of *coupling constraints* which support mixed choices of integer variable encoding. We propose various coupling constraints, which we show benefit SAT solver performance in five case studies in the experimental evaluation.

This chapter is adapted from published and presented[5] work at CPAIOR'22 [18]:

> H. Bierlee, G. Gange, G. Tack, J. J. Dekker, and P. J. Stuckey, "Coupling Different Integer Encodings for SAT," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 19th International Conference, CPAIOR 2022, Los Angeles, CA, USA, June 20-23, 2022, Proceedings*, P. Schaus, Ed., in Lecture Notes in Computer Science, vol. 13292. Springer, 2022, pp. 44-63. doi: 10.1007/978-3-031-08011-1_5.

This chapter extends work from my MSc. thesis [85].

This chapter is organized as follows. In Sec. 3.1, we introduce the problem and our approach. In Sec. 3.2, we give additional preliminaries specific to this chapter. In Sec. 3.3, we cover the main theoretical contribution: the encoding of coupling constraints. In Sec. 3.4, we experimentally evaluate the new encoding framework. In Sec. 3.5, we conclude with our findings and discuss future work.

[5] https://www.youtube.com/watch?v=mTl4xnfKJMI (accessed November 2024)

3.1 Introduction

Within the last twenty years, SAT solving has increased in terms of scalability and performance. More recently, optimization approaches based on SAT, so called Maximum Satisfiability (MaxSAT) technology, have been rapidly developing. SAT and MaxSAT solvers now provide a viable alternative solving technology for many discrete optimization problems.

In Chp. 2, we have seen different encoding approaches of CSPs, constraints, and integer variables. For example, let's recall the three fundamental integer variable encodings: the *direct*, *order* and *binary* encoding. Some integer constraints are more effectively encoded using one integer encoding than another. In realistic problems with many different constraints, the right approach then is to encode each constraint using its preferred integer encoding. This means we inevitably have to *couple* different integer encodings. This can be managed by encoding an integer variable in two ways and *channelling* the two encodings. Or, we can avoid a redundant encoding by directly encoding constraints that couple the different encodings of the variables involved.

There is little existing work on coupling encodings except for channelling constraints, that is, coupling encodings of equality and coupling encodings of linear inequality constraints [86]. Currently, expert SAT modellers build models directly from clauses, or use libraries to encode common constraints such as At-Most-One (`AMO`), cardinality or Pseudo-Boolean (PB) constraints [87], [88]. Alternatively, SAT compilers such as `FznTini` [89] and Picat-SAT (`picat-sat`) [90], or some modelling languages such as Essence, determine a translation of a high-level model. In the existing approaches, the variable encoding choices made by the SAT compiler or modeller are hard or impossible to change, since the encoding of every constraint depends on it. While some encodings already implicitly or explicitly couple integer encodings, no previous work has comprehensively explored the ways in which constraints can couple integer variables.

Encoding constraints into clauses is a rich area of research. Even for simple constraints defined on Boolean variables such as at-most-one, cardinality, or PB constraints there are dozens of papers [16], [65], [91], [92], [93]. No single approach has surfaced as best for all problems. Determining which of these approaches to use for which problem is a non-trivial question. In this chapter, we show how a mix of encodings within the same problem is likely to outperform a single or fixed encoding approach. While the particular mix requires expert insight and experimentation, the MiniZinc constraint modelling language [3] makes this accessible through variable annotations specifying the encodings.

Contributions

The contributions in this chapter are:

- A framework to encode individual integer variables with the direct, order or binary integer variable encoding using coupling constraints
- Coupling constraints and propagation proofs for equality, inequality and element constraints. Furthermore, we discuss zero-overhead views for the uniform encoding of affine, minimum/ maximum and element constraints.
- An evaluation of the framework on five case studies showing improvement in solve times on mixed encodings of the same problem compared to uniform encodings and control SAT-based encoders.

3.2 Preliminaries and related works

In this section, we cover the preliminary concept specific to this chapter, namely of propagation completeness in Sec. 3.2.1, and we discuss related work in Sec. 3.2.2.

3.2.1 Propagation Completeness

In this chapter, we extend our toolset on Constraint Programming (CP) and SAT in order to formally prove the propagation strength of encodings. For this, we primarily adapt the concept of propagation completeness.

CP solvers work by propagation over atomic constraints. An *atomic constraint* is (for our purposes) a unary constraint from a given language (which includes the always false constraint `false`). The usual atomic constraints for CP are $x = d$, $x \neq d$, $x \geq d$ and $x \leq d$. A propagator f_c for constraint c takes a current domain $\mathcal{D}$ given as a set of atomic constraints, and determines a set of new atomic constraints. These are a consequence of the current domain and the constraint, i.e. $a \in f_c(\mathcal{D})$ implies that $\mathcal{D} \wedge c \rightarrow a$.

A propagator f_c is Propagation Complete (PC) [94] for constraint c and a language of atomic constraints $\mathcal{L}$ iff for any $\mathcal{D} \subseteq \mathcal{L}$ and new atom $a \in \mathcal{L}$, $a \notin \mathcal{D}$: if $\mathcal{D} \wedge c \rightarrow a$ then $a \in f_c(\mathcal{D})$. That is, the propagator finds all atomic constraints in $\mathcal{L}$ that are consequences of the constraint and the current domain. In this chapter, the propagator is always Unit Propagation (UP) (see Def. 2.10).

Let $\text{DIRECT}(x)$ be the language of atomic constraints $\texttt{false} \cup \{x = d, x \neq d \mid x \in \mathcal{X}, d \in D(x)\}$. A propagator is *domain consistent* if it is PC for $\mathcal{L} = \text{DIRECT}(x)$. Let $\text{ORDER}(x)$ be the language of atomic constraints $\texttt{false} \cup \{x \leq d, x \geq d \mid x \in \mathcal{X}, d \in D(x)\}$. A propagator is bounds consistent (see Def. 2.8) if it is PC for $\mathcal{L} = \text{ORDER}(x)$ [95]. Finally, let $\text{BINARY}(x)$ be the language of atomic constraint $\{\texttt{false}\} \cup \{\text{bit}(x, k), \neg\text{bit}(x, k) \mid x \in \mathcal{X}, k \in \mathcal{B}(x)\}$.

If we prove that an encoding is PC, then this also implies that the codomain of the encoding is correct. The reason for this is that UP on PC encoding should immediately infer `false` for any assignment which is not a solution.

Proposition 3.1 Consider a function $E : P \to Q$ from and to CSPs P and Q. If UP is PC on $E(P)$ for P and language $\mathcal{L}$, then there exists a function with codomain $\text{sols}(P)$, i.e. $\mathcal{J}_P : \text{sols}(Q) \to \text{sols}(P)$.

Proof. Assume there is an $\mathcal{A} \in \text{sols}(E(P))$ for which $\mathcal{J}_P(\mathcal{A}) \notin \text{sols}(P)$. Then, since $E(P)$ is PC for P, UP should set `false` as a consequence of $\mathcal{A} \wedge E(P)$. This is a contradiction, because $\mathcal{A} \notin \text{sols}(E(P))$. □

3.2.2 Existing SAT encoders

In this section, we survey some existing, general SAT encoders. We discuss in what sense different integer variable encodings are used within the same problem.

Order encoding is adopted by some encoders such as `PBSugar` [96] and `BEE` [97], despite the risk of exponential blow-up. This is sometimes paired with a channelled direct encoding for flexibility `Sp-Or` [98] and `proteus` [99]. Some works mix features of different encodings into other encodings, which are covered in Sec. 2.3.4.

Some works are concerned with picking the right encoding for the right variable. The encoder `proteus` attempts to predict, for a given instance, which encoding of variables (and constraints) will be most effective, then commits to that encoding for the whole instance. The encoder `satune` [100] selects encodings on a per-variable basis, and optimizes the domain representation, based on a training phase for a class of instances. However, all connected variables still share a common representation, so the encoding is not as flexible as the coupling constraints proposed in this chapter. A more radical approach, adopted by current Lazy Clause Generation (LCG) solvers is to implicitly maintain a partial representation consisting of the direct and order encoding, introducing new literals and channelling as necessary. An example of this is is Chuffed (`chuffed`) [80].

Diet-Sugar/Fun-sCOP (`diet-sugar`) [101] is the only other SAT encoder that we are aware of that considers encoding the same constraints with different variable encodings for each variable. It is limited to linear constraints, for which it combines the order and binary encoding. The SAT translation chooses a single encoding for each variable using a heuristic that leads to an overall small and effective encoding of the constraints. The resulting mixed encodings are shown to yield significant improvement. In many cases, however, the best encoding of a problem might turn out to be completely uniform.

A uniform binary encoding is a popular choice for SAT encoders. General SAT encoders that support MiniZinc have been developed using binary encoding are `FznTini` [89] and Picat-SAT

(`picat-sat`) [90]. Both use the sign-and-magnitude representation scheme. The binary encoding has a lower propagation strength (as shown by Prop. 3.4), has continued to prove itself in the `picat-sat` encoder in the yearly MiniZinc Challenge In the 2024 edition, among 23 solvers, `picat-sat` attained 2nd place. However, it is clear that encoders such as `picat-sat` essentially introduce other forms of integer encoding in some global constraint decompositions [102].

Savile Row (`savile-row`) [103] similarly converts a high-level model (specified in Essence Prime) into SAT. `savile-row` commits to a channelled direct-order encoding for variables, allowing each constraint to choose its preferred uniform encoding, though they apply some transformations [104] to improve the generated SAT model.

SAT pre-processing tools

Some techniques and tools exist between encoder and solver.

`Espresso` [105] is a logic optimizer which produces minimal representations from truth-tables. `picat-sat` uses this for the table constraint, or for the binary encoding's consistency constraints of small-domain variables. `Satelite` [106] is known as a SAT pre-processor, which attempts to simplify encodings to make them easier to solve. It works by removing redundancies (or by adding *useful* redundancies).

Another interesting approach is the detection of higher-level structures within SAT encodings. The `lingeling` SAT solver implements this for cardinality constraints, essentially recovering the starting point of the encoder. Finally, `BEE` searches its order encoding for all Boolean variable equivalences (using a SAT solver), simplifying the resulting SAT model.

3.3 Coupling integer encodings

This section covers the central theory of this chapter. In Sec. 3.3.1, we show proofs of PC for each integer encoding. In Sec. 3.3.2, we show a general framework which can combine different integer encodings for the same problem. The following sections all propose concrete coupling encodings for different constraints: equality (Sec. 3.3.3), inequality (Sec. 3.3.4) and element (Sec. 3.3.5). In Sec. 3.3.6, we expand on *views*, encodings which for some integer encodings of some constraints can reuse some or all encoding variables.

3.3.1 Encoding integer variables

Recall our three fundamental integer encodings from Sec. 2.3.3: the direct, order and binary encoding. In this section, we show which are PC and which are not. For simplicity, we reason with the unsigned binary scheme, i.e. $R = \mathtt{0}$ unless specified otherwise. However, for negative domains values, the same encodings hold for the offset scheme, and by extension the two's complement scheme.

Proposition 3.2 UP on $E^{\mathbb{D}}$ (see Eq. 2.18) is PC for the constraint $x \in D(x)$ and the language of atomic constraints DIRECT(x).

This is because we can use a domain consistent encoding for the `ALO` and `AMO` encodings (in Eq. 2.11 and Eq. 2.12, respectively), which is enforced on $\text{vars}(E^{\mathbb{D}}(x))$. Thus, if all literals except $[\![x = d]\!], d \in D(x)$ are set to `false`, i.e. $[\![x \neq d']\!]$ for $d' \in D(x), d' \neq d$, encoding `ALO` propagates the remaining literal $[\![x = d]\!]$. For example, UP on the direct encoding of $x \in [1..3] \neq 1 \wedge x \neq 2$ sets $[\![x = 3]\!]$:

$$\begin{aligned}
([\![x = 1]\!] \vee [\![x = 2]\!] \vee [\![x = 3]\!]) \wedge [\![x \neq 1]\!] \wedge [\![x \neq 2]\!] &\implies [\![x \neq 1]\!] \\
([\![x = 2]\!] \vee [\![x = 3]\!]) \wedge [\![x \neq 2]\!] &\implies [\![x \neq 2]\!] \\
[\![x = 3]\!] &\implies [\![x = 3]\!] \\
\top \quad \textit{fixp.}
\end{aligned}$$

If instead a single literal $[\![x = d]\!]$ is set to `true`, encoding `AMO` propagates literals $[\![x \neq d']\!]$ for all other $d' \in D(x), d' \neq d$. For example, UP on the direct encoding of $x \in [1..3] = 1$ sets $\neg[\![x = 2]\!] \wedge \neg[\![x = 3]\!]$:

$$\begin{aligned}
([\![x \neq 1]\!] \vee [\![x \neq 2]\!]) \wedge ([\![x \neq 1]\!] \vee [\![x \neq 3]\!]) \wedge ([\![x \neq 2]\!] \vee [\![x \neq 3]\!]) \wedge [\![x = 1]\!] &\implies [\![x = 1]\!] \\
[\![x \neq 2]\!] \wedge [\![x \neq 3]\!] \wedge ([\![x \neq 2]\!] \vee [\![x \neq 3]\!]) &\implies [\![x \neq 2]\!] \\
[\![x \neq 3]\!] &\implies [\![x \neq 3]\!] \\
\top \quad \textit{fixp.}
\end{aligned}$$

Proposition 3.3 UP on $E^{\mathbb{O}}$ (see Eq. 2.20) is PC for constraint $x \in D(x)$ and the language of atomic constraints ORDER(x).

Proof. If $[\![x \geq d_i]\!]$, then $[\![x \geq d_{i-1}]\!]$, then $[\![x \geq d_{i-2}]\!]$, and so on. Similarly, if $[\![x \leq d_i]\!]$, then $[\![x \leq d_{i+1}]\!]$, $[\![x \leq d_{i+1}]\!]$, and so on. □

For example, UP on the order encoding of $x \in [0..4] \geq 2$ sets $[\![x \geq 1]\!]$:

$$\begin{aligned}
([\![x \geq 2]\!] \rightarrow [\![x \geq 1]\!]) \wedge ([\![x \geq 3]\!] \rightarrow [\![x \geq 2]\!]) \wedge ([\![x \geq 4]\!] \rightarrow [\![x \geq 3]\!]) \wedge [\![x \geq 2]\!] &\implies [\![x \geq 2]\!] \\
[\![x \geq 1]\!] \wedge ([\![x \geq 4]\!] \rightarrow [\![x \geq 3]\!]) &\implies [\![x \geq 1]\!] \\
([\![x \geq 4]\!] \rightarrow [\![x \geq 3]\!]) \quad \textit{fixp.}
\end{aligned}$$

Proposition 3.4 UP on $E^{\mathbb{B}}$ (see Eq. 2.24) is *not* PC for constraint $x \in D(x)$ and the language of atomic constraints.

Proof. By counter-example. Let $x_k \equiv [\![\text{bit}(x,k)]\!]$ for readability. Consider the following binary encoding, containing only ternary clauses:

$$E^{\mathbb{B}}(x \subseteq \{0,3,4,7\}) \equiv x \neq 1 \wedge x \neq 2 \wedge x \neq 5 \wedge x \neq 6$$
$$\equiv \neg(x_0 \wedge \neg x_1 \wedge \neg x_2) \wedge \neg(\neg x_0 \wedge x_1 \wedge \neg x_2) \wedge \neg(x_0 \wedge \neg x_1 \wedge x_2) \wedge \neg(\neg x_0 \wedge x_1 \wedge x_2)$$

Note that no ternary clause can propagate from a single literal.

$$(\neg x_0 \vee x_1 \vee x_2) \wedge (x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_1 \vee \neg x_2) \wedge (x_0 \vee \neg x_1 \vee \neg x_2) \wedge x_0 \implies x_0$$
$$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \quad \textit{fixp.}$$

After setting $x_0 = \texttt{true}$, only odd domain values remain with $\{3,7\}$. Both 3 and 7 have $x_1 = \texttt{true}$, so this literal should be propagated but is not. □

3.3.2 Coupling integer encodings in constraints and CSPs

We now show how the different integer encodings can be combined for a single problem. We first show this for a single constraint, and then for a CSP.

Coupling integer encodings in constraints

In Eq. 2.28, we have seen how a CSP P of a single constraint $c(x_1,...,x_r)$ is encoded using a variable encoding $\mathbb{E}$ for all input variables. This is called a *uniform* encoding of constraint c. We now introduce a notation to specify *mixed* encodings of constraints in which different variables have different encodings. To this end, we extend the encoding and interpretation functions. We assume that all $\mathbb{E} \in \{\mathbb{D}, \mathbb{O}, \mathbb{B}\}$, but the approach can include any correct integer variable encoding.

Definition 3.1 The encoding of (the CSP of) constraint $c(x_1,...,x_r)$ using variable encodings $\mathbb{E}_1,...,\mathbb{E}_r$ for integer variables $x_1,...,x_r$, respectively, yields:

$$E(c(x_1{:}\mathbb{E}_1,...,x_r{:}\mathbb{E}_r)) \equiv E^{\mathbb{E}_1}(x_1),...,E^{\mathbb{E}_r}(x_r) \wedge Q_c \tag{3.1}$$

We also define a shorthand for variable encodings, $E^{\mathbb{E}}(x) \equiv E(x{:}\mathbb{E})$. Again, inner encoding Q_c is chosen so that the following interpretation function is surjective:

$$\mathcal{J}_{c(x_1:\mathbb{E}_1,\dots,x_r:\mathbb{E}_r)} : \mathrm{sols}(E(c(x_1{:}\mathbb{E}_1,\dots,x_r{:}\mathbb{E}_r))) \twoheadrightarrow \mathrm{sols}(c(x_1{:}\mathbb{E}_1,\dots,x_r{:}\mathbb{E}_r)) \tag{3.2.1}$$

$$\equiv \left\{\left(x_i \mapsto \mathcal{J}^{\mathbb{E}_i}_{x_i}\left(\mathcal{A}|_{\mathrm{vars}(E(x_i:\mathbb{E}_i))}\right)\right) \mid x_i \in \mathrm{vars}(c(x_1,\dots,x_r))\right\} \tag{3.2.2}$$

Note that correct encodings of (non-tautological) constraints must use the given variable encoding literals in the construction of the inner encoding Q_c. They cannot introduce another encoding of their variables and enforce the constraint based on it. Otherwise, the interpretation function would operate on assignments of unconstrained encoding variables, leading to an incorrect encoding.

The particular choice of variable encodings $\mathbb{E}_1,\dots,\mathbb{E}_r$ changes the encoding of the constraint. For instance, consider a unary integer constraint $c(x \subseteq D^{\mathbb{Z}}(x))$ and two of its encodings $E(c(x){:}\mathbb{D}) \equiv E(x{:}\mathbb{D}) \wedge Q_c$ and $E(c(x){:}\mathbb{O}) \equiv E(c(x){:}\mathbb{O}) \wedge Q'_c$. If $E(x{:}\mathbb{D}) \not\equiv E(x{:}\mathbb{O})$ (when $|D(x)| > 1$), then $Q_c \not\equiv Q'_c$ in order to still encode $c(x)$ correctly in both cases. Different constraint encodings are thus also disambiguated by the specified variable encodings. When multiple possible constraint encodings for the same variable encodings are known, we will have to disambiguate further.

If all encodings are the same (i.e. $\mathbb{E}_1 = \cdots = \mathbb{E}_r$), the constraint has a uniform encoding. We introduce a shorthand notation for uniform encodings: $c(x_1,\dots,x_r){:}\mathbb{E} \equiv c(x_1{:}\mathbb{E},\dots,x_r{:}\mathbb{E})$. If different encodings are used (i.e. $\exists_{1\leq i<r}(\mathbb{E}_i \neq \mathbb{E}_{i+1})$), the constraint is a *coupling* constraint which has a *mixed* encoding.

To encode $E(c(x_1{:}\mathbb{E}_1,\dots,x_r{:}\mathbb{E}_r))$, the challenge is to implement a correct inner encoding Q_c for each choice of $\mathbb{E}_1,\dots,\mathbb{E}_r$. Since there are three encodings we support, there are $O(3^r)$ choices. A *comprehensive coupling*, which covers all combinations of encodings, is possible for some constraints of small r. However, since the number of choices grows exponentially, and since constraints might be have an arbitrary number of arguments, or collections of integer variables (e.g. an array) of arbitrary size, we often need a generalized method. For some constraint, creating a comprehensive coupling encoding is still straightforward.

Definition 3.2 A comprehensive coupling encoding of the linear constraint, $E\left(\sum_{i=1}^{r} q_i x_i : \mathbb{E}_i \in \mathbb{Z} \# k\right)$, rewrites each linear term $q_i x_i{:}\mathbb{E}_i$ as multiple linear PB terms:

$$q_i \cdot x_i{:}\mathbb{E}_i \equiv \begin{cases} \sum_{d_j \in D(x_i)} q_i \cdot d_j \cdot [\![x_i = d_j]\!] & \text{if } x_i{:}\mathbb{D} \\ \mathrm{lb}(x_i) + \sum_{d_j \in D(x_i)\setminus\{\mathrm{lb}(x_i)\}} q_i \cdot (d_j - d_{j-1}) \cdot [\![x_i \geq d_j]\!] & \text{if } x_i{:}\mathbb{O} \\ \sum_{0\leq k<\mathcal{B}^0(x_i)} 2^k \cdot [\![\mathrm{bit}^0(x_i,k)]\!] & \text{if } x_i{:}\mathbb{B} \end{cases}$$

This rewrite corresponds to how an integer variable encoding is interpreted by $\mathcal{I}^{\mathbb{E}}$. The result is a PB encoding of the linear constraint, which can be encoded into SAT by various PB-SAT encoding methods. Note that we can similarly encode arbitrary integer linear objectives into a PB objective required for MaxSAT.

Coupling integer encoding in CSPs

Let us now generalize the encoding function $E^{\mathbb{E}}$ for general CSPs with uniform encoding $\mathbb{E}$ to an encoding function E for general CSP P which potentially has more than one encoding.

For such mixed encoding CSPs, coupling constraints are necessary. Consider the *constraint graph* of CSP P, which we define as a hypergraph in which variables are represented by vertices and constraints by hyperedges. A set of vertices form a hyperedge iff their corresponding variables occur in the same constraint. If two different constraints $c_i{:}\mathbb{E}$ and $c_j{:}\mathbb{E}'$ where $i \neq j$ appear within the same component of the constraint graph, and they are are encoded with *different* uniform encodings $\mathbb{E} \neq \mathbb{E}'$, then there must be one or more coupling constraint along any path from a variable $x{:}\mathbb{E}$ in $\mathrm{vars}(c)$ to a variable $y{:}\mathbb{E}'$ in $\mathrm{vars}(c_j)$.

The encoding function E decides the encodings of each variable of each constraint, however, it must follow one invariant: every variable x must always be encoded using the same variable encoding $\mathbb{E}_x$. Otherwise, different constraints are enforced on different, unconnected representations of x, leading to incorrect encodings.

We can update the interpretation function $\mathcal{I}_P^{\mathbb{E}}$ of Eq. 2.30 for uniform CSPs to one which works for CSPs with mixed encodings. This allows us to create the following interpretation function for mixed encodings of general CSPs:

$$\mathcal{I}_P : \mathrm{sols}(E(P)) \twoheadrightarrow \mathrm{sols}(P) \equiv \left\{ \left(x \mapsto \mathcal{I}_x^{\mathbb{E}_x}\left(\mathcal{A}|_{\mathrm{vars}(E(x:\mathbb{E}_x))} \right) \right) \mid x \in \mathrm{vars}(P) \right\} \tag{3.3}$$

3.3.3 Coupling equality constraints (channelling)

For linear constraints, encoding for any choice of variable encodings was straightforward (see Def. 3.2). However, for other constraints, a correct and comprehensive coupling encoding is not so obvious. An alternative to directly coupling the remaining mixed choices (including order encoded variables) is to encode a *channelling constraint*:

$$E(c(x_1{:}\mathbb{E}_1, ..., x_r{:}\mathbb{E}_r)) \equiv E(x_1{:}\mathbb{E}_1 = y_1{:}\mathbb{E}'_1) \wedge \cdots \wedge E(x_r{:}\mathbb{E}_r = y_r{:}\mathbb{E}'_r) \wedge E(c(y_1{:}\mathbb{E}'_1, ..., y_r{:}\mathbb{E}'_r))$$

If we lack support for the encoding of the constraint for $\mathbb{E}_1, ..., \mathbb{E}_r$, but we do support the encoding of that constraint for $\mathbb{E}'_1, ..., \mathbb{E}'_r$, and we support coupled encodings of the equality constraints for all $\mathbb{E}_i$, $\mathbb{E}'_i$, then we can couple c comprehensively for any $\mathbb{E}_1, ..., \mathbb{E}_r$.

In CP, the equality constraints are known as channelling constraints, connecting different representations of the same objects. In our case, channelling can be seen as coupling equality constraints. The added flexibility comes at the cost of encoding redundancy:

- we introduce additional encoding variables, $\mathbf{vars}(E(y_i))$;
- we introduce the encoding of additional channelling constraints, $E(x_i{:}\mathbb{E}_i = y_i{:}\mathbb{E}'_i)$; and
- we introduce the encoding of additional consistency constraints, $\mathbf{cons}(E(y{:}\mathbb{E}'))$.

However, depending on $E(c(y_1{:}\mathbb{E}'_1, ..., y_r{:}\mathbb{E}'_r))$, the equality constraints might allow us to omit consistency constraint: either $E(\mathbf{cons}(x{:}\mathbb{E}))$ or $E(\mathbf{cons}(y{:}\mathbb{E}'))$, and sometimes even both.

A potential upside of channelling is that we can avoid using integer encodings which might be unsuited to particular constraints. An interesting example is the SAT encoding of PB encoded linear constraints (see Def. 3.2), which tend to blow up in encoding size for large domains if we use unary integer encodings (e.g. direct or order) instead of binary. For unary encodings, this problem can be somewhat remedied by optimizing the SAT encoding of the PB constraints for sets of terms which originate from integer variables. Still, for many PB constraints, binary encodings are preferable even with this optimization.[6] A second upside of the encoding redundancy is that it has been shown to improve performance in some cases [98].

Encoding unary constraints

When coupling equality constraints, it is very useful to have comprehensive coupling encodings of the following three unary integer constraints for every integer encoding: $x \in D^{\mathbb{Z}}(x) = v$, $x \in D^{\mathbb{Z}}(x) \geq v$ and $\mathrm{bit}(x \in D^{\mathbb{Z}}(x), k)$. Having only one variable, unary constraints are always uniformly encoded. Note that some of the encodings shown here are decompositions (non-SAT encodings), but they are useful as part of *other* SAT encoding.

Depending on the encoding $\mathbb{E}$ of x, the encoding might be trivial, e.g. $E(x{:}\mathbb{D} = v)$ yields a single literal. Others are only slightly more involved, but still clearly correct. The unary equality constraint $x{:}\mathbb{E} = v$ for fixed integer constant v is familiar:

$$E(x{:}\mathbb{D} = v) \equiv [\![x = v]\!] \tag{3.4.1}$$

$$E(x{:}\mathbb{O} = v) \equiv [\![x \geq v]\!] \wedge [\![x < v + 1]\!] \tag{3.4.2}$$

$$E(x{:}\mathbb{B} = v) \equiv \bigwedge_{0 \leq k < \mathcal{B}(x)} ([\![\mathrm{bit}(x, k)]\!] = \mathrm{bit}(v, k)) \tag{3.4.3}$$

As is the unary inequality constraint, $x{:}\mathbb{E} \geq v$:

[6] This optimization is not considered for this chapter, as it requires considerable effort to extend existing works [107] [108], which will be the main topic of Chp. 5.

$$E(x{:}\mathbb{D} \geq v) \equiv \bigwedge_{d \in D(x), d < v} ([\![x \neq d]\!]) \tag{3.5.1}$$

$$E(x{:}\mathbb{O} \geq v) \equiv [\![x \geq v]\!] \tag{3.5.2}$$

$$E(x{:}\mathbb{B} \geq v) \equiv [[\![\mathrm{bit}(x,k)]\!] \mid 0 \leq 1 < \mathcal{B}(x)] \geq [\mathrm{bit}(v,k) \mid 0 \leq 1 < \mathcal{B}(x)] \tag{3.5.3}$$

For Eq. 3.5.3, we encode an inequality constraint between lists $[a_1, a_2, ...] \# [b_1, b_2, ...], \# \in \{\geq, >, <, \leq\}$. This uses some well-known lexicographical comparison encoding, in this case for Booleans variables a_i and Boolean constants b_i. Additional methods exist when both lists contain Boolean variables, in order to encode $E(x{:}\mathbb{B} \leq y{:}\mathbb{B})$ [109].

The unary constraint $\mathrm{bit}(x,k) = v$ for Boolean constant v holds iff bit k in the binary representation of x is v (e.g. the constraint $\mathrm{bit}(x,0) = 1$ holds iff x only takes even values, see Ex. 3.1). This is represented by a single literal for the binary encoding. However, for the direct or order encoding, this leads to a decomposition:

$$E(\mathrm{bit}(x{:}\mathbb{E}, k)), \mathbb{E} \in \{\mathbb{D}, \mathbb{O}\} \equiv x{:}\mathbb{E} \in \{d \mid d \in D(x), \mathrm{bit}(d,k)\} \tag{3.6.1}$$

$$E(\mathrm{bit}(x{:}\mathbb{B}, k) = v) \equiv [\![\mathrm{bit}(x,k)]\!] \tag{3.6.2}$$

For Eq. 3.6.1, to encode the set membership constraint, $(x \subseteq D^{\mathbb{Z}}(x)) \in S$, which restricts integer variable x to take values from set of constant integers, S. For the direct encoding this is a single clause:

$$E((x{:}\mathbb{D} \subseteq D^{\mathbb{Z}}(x{:}\mathbb{D})) \in S) \equiv \bigvee_{v \in S} (E(x = v)) \tag{3.7}$$

Example 3.1 Say we need to encode $\mathrm{bit}(x{:}\mathbb{D} \subseteq [0..6], 2)$. For $D(x) \equiv [0..6]$, we have $\mathcal{B}(x) \equiv 3$. For $k \equiv 1$, Eq. 3.6.1 yields a set membership constraint over $S \equiv \{2, 3, 6\}$, since those are all the values in $D(x)$ for which $\mathrm{bit}(x,1)$ holds:

S	$\mathrm{bit}(x,2)$	$\mathrm{bit}(x,1)$	$\mathrm{bit}(x,0)$
0	0	0	0
1	0	0	1
2	0	**1**	0
3	0	**1**	1
4	1	0	0
5	1	0	1
6	1	**1**	0
7	1	**1**	1

Tab. 3.1: The set $S \equiv \{2, 3, 6\}$ for which $\mathrm{bit}(x \subseteq [0..6], 2)$ holds.

Thus, we use a direct encoding on the following set membership constraint to enforce a direct encoding of the original bit constraint:

$$E(x{:}\mathbb{D} \in \{2,3,6\}) \equiv ([\![x=2]\!] \vee [\![x=3]\!] \vee [\![x=6]\!])$$

For set membership using the order encoding, we can represent a set S as a union of *maximal intervals*:

$$S \equiv \bigcup_{l,r \in S, l \leq r, [l..r] \subseteq S, l-1, r+1 \notin S} [l..r] \tag{3.8}$$

E.g. $S \equiv \{2,3,6\} \equiv [2..3] \cup \{6\}$. Then, $x \in S$ iff x is within the bounds of any maximal interval of S. A disjunction of these *interval membership* constraints forms a Disjunctive Normal Form (DNF) encoding of the set membership constraint.

$$E((x{:}\mathbb{O} \subseteq D^{\mathbb{Z}}(x{:}\mathbb{O})) \in S) \equiv \bigvee_{l,r \in S, l \leq r, [l..r] \subseteq S, l-1, r+1 \notin S} x \in [l..r] \tag{3.9.1}$$

$$E((x{:}\mathbb{O} \subseteq D^{\mathbb{Z}}(x{:}\mathbb{O})) \in [l..r]) \equiv x \geq l \wedge x \leq r \equiv [\![x \geq l]\!] \wedge [\![x \leq r]\!] \tag{3.9.2}$$

We can also encode the negation of the `set_in` constraint:

$$E((x \subseteq D^{\mathbb{Z}}(x)) \notin S) \equiv E(x \in D(x) \setminus S)$$

Again, encodings of this section are evidently correct. However, more complex encodings can be easily proven correct if they are composed of these unary constraints. We will now show that Eq. 3.9.2 and Eq. 3.9.1 are PC for their respective constraints and languages.

Theorem 3.1 UP on the clauses of Eq. 3.9.2 is PC for constraint $x \in D(x) \wedge x \in [l..r]$ and $\mathcal{L} = \text{ORDER}(x)$, thus enforcing domain consistency.

Proof. In Sec. 3.3.1, we have already seen how $E(x \subseteq \{d_1, ..., d_m\} \geq v)$ is PC for $\mathcal{L} = \text{ORDER}(x)$. Thus, UP on $E(x{:}\mathbb{O} \geq l \wedge x{:}\mathbb{O} \leq r)$ sets all $[\![x \geq d_i]\!], d_1 \leq d_i \leq l$ to `true` and $[\![x \geq d_{i'}]\!], r < d_{i'} \leq d_m$ to `false`. These are all the expected consequences of the constraint. □

Theorem 3.2 UP on the clauses of Eq. 3.9.1 is PC for constraint $x \in D(x) \wedge x \in S$ and $\mathcal{L} = \text{ORDER}(y)$, thus enforcing domain consistency.

Proof. The gaps of set S are the maximal intervals of its complement. This means we can encode $x{:}\mathbb{O} \in S$ by negated interval membership constraints on each gap $[l_i..r_i]$ in S:

$$x \in S \equiv \neg\neg(x \in S) \equiv \neg(x \notin S) \equiv \neg(x \in D(x) \setminus S) \equiv \neg \bigvee (x \in [l_i..r_i]) \equiv \bigwedge (x \notin [l_i..r_i])$$

Suppose $[\![x \geq d]\!]$ is set to `true`. If $d \in S$, we expect no propagation from this constraint. If $d \notin S$, we expect to set $[\![x \geq e]\!]$ where $e \equiv \min_{e \in S \cap D(x), d<e}(e)$, or cause failure if there is no e (i.e. if $d \equiv \text{ub}(x)$ or if $d > \max(S)$). Since $d \in D(x)$ and $d \notin S$, we have $d \in D(x) \setminus S$. Consequently, $d \in [l_i..r_i]$ for some i. Thus, UP on $x \notin [l_i..r_i]$ sets $x > r_i$ by Thm. 3.1. This causes failure if $r_i > \text{ub}(x)$ or if $r_i \equiv \max(S)$. If there is no failure, then this sets $[\![x > r_i]\!] \equiv [\![x \geq r_i + 1]\!] \equiv [\![x \geq e]\!]$, since $d < r_i + 1$ and $r_i + 1 \in S \cap D(x)$. A similar argument applies when $[\![x < v]\!]$ is set to `true`. □

Example 3.2 If we encode $E(x{:}\mathbb{O} \subseteq [1..4] \in \{1,4\})$, then we encode $x \notin [1..4] \setminus \{1,4\} \equiv x \notin [2..3] \equiv [\![x < 2]\!] \vee [\![x > 3]\!] \equiv [\![x \geq 2]\!] \rightarrow [\![x \geq 4]\!]$. If we decide $[\![x \geq 3]\!]$, then UP sets $[\![x \geq 4]\!]$:

$$\begin{aligned} ([\![x \geq 3]\!] \rightarrow [\![x \geq 2]\!]) \wedge ([\![x \geq 4]\!] \rightarrow [\![x \geq 3]\!]) \wedge ([\![x \geq 2]\!] \rightarrow [\![x \geq 4]\!]) \quad & \textit{dec. } [\![x \geq 3]\!] \\ [\![x \geq 2]\!] \wedge ([\![x \geq 2]\!] \rightarrow [\![x \geq 4]\!]) \quad & \Longrightarrow \quad [\![x \geq 2]\!] \\ [\![x \geq 4]\!] \quad & \Longrightarrow \quad [\![x \geq 4]\!] \\ \top \quad & \textit{fixp.} \end{aligned}$$

If instead $S \equiv \{1,5\}$, then $([\![x \geq 2]\!] \rightarrow [\![x \geq 5]\!]) \equiv [\![x < 2]\!]$, which sets $[\![x < 3]\!]$, which causes failure by our decision of $[\![x \geq 3]\!]$. If instead $D(x) \equiv [1..3]$, a similar scenario fails by $([\![x \geq 2]\!] \rightarrow [\![x \geq 4]\!]) \equiv [\![x < 2]\!]$.

Coupling $x{:}\mathbb{D} = y{:}\mathbb{O}$

This is the most commonly used channelling constraint in SAT encodings.

Definition 3.3 We define a function to create the inner encoding of the equality constraint:

$$E'(x{:}\mathbb{D} = y{:}\mathbb{O}) \equiv \forall_{d \in D(x) \cup D(y)} (x{:}\mathbb{D} = d) \leftrightarrow (y{:}\mathbb{O} = d) \tag{3.10.1}$$

$$\equiv \bigwedge_{d \in D(x) \cup D(y)} [\![x = d]\!] \leftrightarrow ([\![y \geq d]\!] \wedge [\![y \leq d]\!]) \tag{3.10.2}$$

We can use the inner encoding of Eq. 3.10 in conjunction with just one consistency constraint: either the `AMO` of $E(x{:}\mathbb{D})$ (Eq. 2.18) or the Implication Chain (`IC`) of $E(y{:}\mathbb{O})$ (Eq. 2.20). Either is correct *and* PC.

Theorem 3.3 The following encoding function is PC for $x \in D(x) \wedge x = y \wedge y \in D(y)$ and $\mathcal{L} = \text{DIRECT}(x) \cup \text{ORDER}(y)$:

$$E(y{:}\mathbb{O} = x{:}\mathbb{D}) \equiv E(y{:}\mathbb{O}) \wedge E'(x{:}\mathbb{D} = y{:}\mathbb{O})$$

Thm. 3.3 is already proven [79]. However, instead of omitting the `AMO` constraint, we can also omit the `IC` without losing correctness.

However, it is not PC, since UP sets only the encoding literals of x, *not* of y.

Theorem 3.4 The following encoding function is not PC for $x \in D(x) \wedge x = y \wedge y \in D(y)$ and $\mathcal{L} = \text{DIRECT}(x) \cup \text{ORDER}(y)$:

$$E(x{:}\mathbb{D} = y{:}\mathbb{O}) \equiv E(x{:}\mathbb{D}) \wedge E'(x{:}\mathbb{D} = y{:}\mathbb{O}) \tag{3.11}$$

Proof. Setting $[\![y \geq d]\!]$ is not PC. A counter-example is $E(x \in [0..2] = y \in [0..2])$, which, after setting $[\![y \geq 1]\!]$, does not set the expected $[\![x \neq 0]\!]$ by UP, or any other literal.

$$
\begin{aligned}
&(\llbracket x \neq 0 \rrbracket \vee \llbracket y < 0 \rrbracket) \wedge (\llbracket y \geq 0 \rrbracket \vee \llbracket x = 0 \rrbracket) \wedge (\llbracket x \neq 1 \rrbracket \vee \llbracket y \geq 1 \rrbracket) \\
\wedge\ &(\llbracket x \neq 1 \rrbracket \vee \llbracket y < 2 \rrbracket) \wedge (\llbracket y < 1 \rrbracket \vee \llbracket y \geq 2 \rrbracket \vee \llbracket x = 1 \rrbracket) \wedge (\llbracket x \neq 2 \rrbracket \vee \llbracket y \geq 2 \rrbracket) \\
\wedge\ &(\llbracket y < 2 \rrbracket \vee \llbracket x = 2 \rrbracket) \wedge (\llbracket x \neq 0 \rrbracket \vee \llbracket x \neq 1 \rrbracket) \wedge (\llbracket x \neq 0 \rrbracket \vee \llbracket x \neq 2 \rrbracket) \\
\wedge\ &(\llbracket x \neq 1 \rrbracket \vee \llbracket x \neq 2 \rrbracket) \quad \textit{dec. } \llbracket y \geq 1 \rrbracket \\
&(\llbracket x \neq 0 \rrbracket \vee \llbracket y < 0 \rrbracket) \wedge (\llbracket y \geq 0 \rrbracket \vee \llbracket x = 0 \rrbracket) \wedge (\llbracket x \neq 1 \rrbracket \vee \llbracket y < 2 \rrbracket) \\
\wedge\ &(\llbracket y \geq 2 \rrbracket \vee \llbracket x = 1 \rrbracket) \wedge (\llbracket x \neq 2 \rrbracket \vee \llbracket y \geq 2 \rrbracket) \wedge (\llbracket y < 2 \rrbracket \vee \llbracket x = 2 \rrbracket) \\
\wedge\ &(\llbracket x \neq 0 \rrbracket \vee \llbracket x \neq 1 \rrbracket) \wedge (\llbracket x \neq 0 \rrbracket \vee \llbracket x \neq 2 \rrbracket) \wedge (\llbracket x \neq 1 \rrbracket \vee \llbracket x \neq 2 \rrbracket) \quad \textit{fixp.}
\end{aligned}
$$

A similar counter-example can be constructed for $\llbracket y < d \rrbracket$. □

Coupling $x{:}\mathbb{D} = y{:}\mathbb{B}$

For this constraint, there is a trade-off between propagation strength and the size of the encoding. This encoding was initially devised in the context of encoding `AMO` [16], [52], [55] constraints, where channelling to the binary representation implicitly prevents two $\llbracket x = d \rrbracket$ literals from becoming true:

$$
\bigwedge_{d \in D(x), k \in \mathcal{B}(y)} \llbracket x = d \rrbracket \rightarrow (\llbracket \mathrm{bit}(y,k) \rrbracket = \mathrm{bit}(d,k)) \tag{3.12}
$$

It is extended by [52] to an Exactly-One (`EO`) constraint by adding channelling from the binary representation back to the equality literals:

$$
\bigwedge_{d \in D(x)} \left(\left(\bigwedge_{k \in \mathcal{B}(y)} \llbracket \mathrm{bit}(y,k) \rrbracket = \mathrm{bit}(d,k) \right) \rightarrow \llbracket x = d \rrbracket \right) \tag{3.13}
$$

This encoding does not enforce domain consistency between the two representations, which we can show by counter-example.

Theorem 3.5 UP on the clauses of Eq. 3.13 is *not* PC for constraint $x \in D(x) \wedge x = y \wedge y \in D(y)$ and $\mathcal{L} = \mathrm{DIRECT}(x) \cup \mathrm{BINARY}(y)$.

Proof. By counter-example. Let $y_k \equiv \llbracket \mathrm{bit}(y,k) \rrbracket$ for readability. Consider the case $x{:}\mathbb{D} \subseteq [0..3] = y{:}\mathbb{B} \subseteq [0..3]$ where we have removed all the even values from $D(x)$ by search decisions $\llbracket x \neq 0 \rrbracket$ and $\llbracket x \neq 2 \rrbracket$.

$$
\begin{aligned}
&(\llbracket x = 0 \rrbracket \to \neg y_0) \wedge (\llbracket x = 0 \rrbracket \to \neg y_1) \wedge (y_0 \vee y_1 \vee \llbracket x = 0 \rrbracket) \\
\wedge\ &(\llbracket x = 1 \rrbracket \to y_0) \wedge (\llbracket x = 1 \rrbracket \to \neg y_1) \wedge (\neg y_0 \vee y_1 \vee \llbracket x = 1 \rrbracket) \\
\wedge\ &(\llbracket x = 2 \rrbracket \to \neg y_0) \wedge (\llbracket x = 2 \rrbracket \to y_1) \wedge (y_0 \vee \neg y_1 \vee \llbracket x = 2 \rrbracket) \\
\wedge\ &(\llbracket x = 3 \rrbracket \to y_0) \wedge (\llbracket x = 3 \rrbracket \to y_1) \wedge (\neg y_0 \vee \neg y_1 \vee \llbracket x = 3 \rrbracket) \quad \textit{dec. } \llbracket x \neq 0 \rrbracket \\
&(\neg y_0 \to y_1) \wedge (\llbracket x = 1 \rrbracket \to y_0) \wedge (\llbracket x = 1 \rrbracket \to \neg y_1) \\
\wedge\ &(\neg y_0 \vee y_1 \vee \llbracket x = 1 \rrbracket) \wedge (\llbracket x = 2 \rrbracket \to \neg y_0) \wedge (\llbracket x = 2 \rrbracket \to y_1) \\
\wedge\ &(y_0 \vee \neg y_1 \vee \llbracket x = 2 \rrbracket) \wedge (\llbracket x = 3 \rrbracket \to y_0) \wedge (\llbracket x = 3 \rrbracket \to y_1) \\
&\wedge (\neg y_0 \vee \neg y_1 \vee \llbracket x = 3 \rrbracket) \quad \textit{dec. } \llbracket x \neq 2 \rrbracket \\
&(\neg y_0 \to y_1) \wedge (\llbracket x = 1 \rrbracket \to y_0) \wedge (\llbracket x = 1 \rrbracket \to \neg y_1) \\
\wedge\ &(\neg y_0 \vee y_1 \vee \llbracket x = 1 \rrbracket) \wedge (\neg y_0 \to \neg y_1) \wedge (\llbracket x = 3 \rrbracket \to y_0) \\
&\wedge (\llbracket x = 3 \rrbracket \to y_1) \wedge (\neg y_0 \vee \neg y_1 \vee \llbracket x = 3 \rrbracket) \quad \textit{fixp.}
\end{aligned}
$$

We would expect UP to set y_0, but it does not. □

The underlying reason this encoding is not PC is because it only enforces the following relation:

$$
E\left(\forall_{d \in D(x)}((x{:}\mathbb{D} = d) \leftrightarrow (y{:}\mathbb{B} = d))\right) \equiv \bigwedge_{d \in D(x)} \left(\llbracket x = d \rrbracket \leftrightarrow \left(\bigwedge_{k \in \mathcal{B}(y)} \llbracket \mathrm{bit}(y,k) \rrbracket = \mathrm{bit}(d,k) \right) \right)
$$

These equivalences produce exactly the clauses from Eq. 3.12 and Eq. 3.13. In order to achieve domain consistency, we need to enforce a stronger interaction based on bit constraints.

Definition 3.4 The following function correctly encodes $x{:}\mathbb{D} = y{:}\mathbb{B}$:

$$
\begin{aligned}
& E(x{:}\mathbb{D} = y{:}\mathbb{B}) && 3.14.1 \\
\equiv\ & \forall_{k \in \mathcal{B}(y), v \in \{0,1\}}((\mathrm{bit}(x{:}\mathbb{D}, k) = v) \leftrightarrow (\mathrm{bit}(y{:}\mathbb{B}, k) = v)) && 3.14.2 \\
\equiv\ & \forall_{k \in \mathcal{B}(y), v \in \{0,1\}}(x{:}\mathbb{D} \in \{d \mid d \in D(x), \mathrm{bit}(d,k) = v\} \leftrightarrow (\mathrm{bit}(y{:}\mathbb{B}, k) = v)) && 3.14.3 \\
\equiv\ & \bigwedge_{k \in \mathcal{B}(y), v \in \{0,1\}} \left(\left(\bigvee_{d \in D(x), \mathrm{bit}(d,k) = v} \llbracket x = d \rrbracket \right) \leftrightarrow (\llbracket \mathrm{bit}(y,k) \rrbracket = v) \right) && 3.14.4
\end{aligned}
$$

Note that $\mathrm{bit}(x{:}\mathbb{D}, k) = v$ is encoded as a `set_in` constraint by Eq. 3.6.1, then as clause by Eq. 3.7, and $\mathrm{bit}(y{:}\mathbb{B}, k) = v$ to a literal by Eq. 3.6.2.

Proposition 3.5 The clauses from Eq. 3.12 are included in Eq. 3.14.

Proof. Simplifying Eq. 3.14 yields:

$$\bigwedge_{k\in\mathcal{B}(y),v\in\{0,1\},d\in D(x),\mathrm{bit}(d,k)=v} (\llbracket x=d\rrbracket \to \llbracket \mathrm{bit}(y,k)\rrbracket = v) \tag{3.15.1}$$

$$\wedge \bigwedge_{k\in\mathcal{B}(y),v\in\{0,1\}} \left(\llbracket \mathrm{bit}(y,k)\rrbracket = v \to \bigvee_{d\in D(x),\mathrm{bit}(d,k)=v} (\llbracket x=d\rrbracket) \right) \tag{3.15.2}$$

The first conjunction is equivalent to Eq. 3.12 since its generators yield the same set of values:

$$\prod_{k\in\mathcal{B}(y),v\in\{0,1\},d\in D(x),\mathrm{bit}(d,k)=v} \langle k,d,v\rangle \equiv \prod_{d\in D(x),k\in\mathcal{B}(y)} \langle k,d,\mathrm{bit}(d,k)\rangle$$

□

Example 3.3 Let $y_k \equiv \llbracket \mathrm{bit}(y,k)\rrbracket$ for readability. Adding the new clauses from Eq. 3.15.2 to the already existing clauses from Eq. 3.15.1 in Thm. 3.5:

$$\begin{aligned}
&(\neg y_0 \vee \llbracket x=1\rrbracket \vee \llbracket x=3\rrbracket) \wedge (y_0 \vee \llbracket x=0\rrbracket \vee \llbracket x=2\rrbracket) \\
\wedge\ &(\neg y_1 \vee \llbracket x=2\rrbracket \vee \llbracket x=3\rrbracket) \wedge (y_1 \vee \llbracket x\neq 2\rrbracket \vee \llbracket x\neq 3\rrbracket) && \textit{dec. } \llbracket x\neq 0\rrbracket \\
&(\neg y_0 \vee \llbracket x=1\rrbracket \vee \llbracket x=3\rrbracket) \wedge (y_0 \vee \llbracket x=2\rrbracket) \\
\wedge\ &(\neg y_1 \vee \llbracket x=2\rrbracket \vee \llbracket x=3\rrbracket) \wedge (y_1 \vee \llbracket x\neq 2\rrbracket \vee \llbracket x\neq 3\rrbracket) && \textit{dec. } \llbracket x\neq 2\rrbracket \\
&(\neg y_0 \vee \llbracket x=1\rrbracket \vee \llbracket x=3\rrbracket) \wedge y_0 \wedge (\neg y_1 \vee \llbracket x=3\rrbracket) && \implies \quad y_0 \\
&(\llbracket x=1\rrbracket \vee \llbracket x=3\rrbracket) \wedge (\neg y_1 \vee \llbracket x=3\rrbracket) && \textit{fixp.}
\end{aligned}$$

Now, y_0 is set by UP.

Unlike $x{:}\mathbb{D} = y{:}\mathbb{O}$ (Eq. 3.10), the $x{:}\mathbb{D} = y{:}\mathbb{B}$ constraint allows us to omit both consistency constraints, $E(x{:}\mathbb{D})$ and $E(y{:}\mathbb{B})$. This is because Eq. 3.10 iterates over the domain values of x and y, skipping domain gaps. In other words, Eq. 3.10 does not add clauses for fixed literals, i.e. $\llbracket x=d\rrbracket \notin D(x)$ or $\llbracket y\geq d\rrbracket \notin D(y)$. However, even if a $\llbracket \mathrm{bit}(y,k)\rrbracket$ is fixed (e.g. if $D(y)$ only contains odd values), the encoding $x{:}\mathbb{D} = y{:}\mathbb{B}$ in Eq. 3.14 still adds a clause for it. We speculate that this is what ensures consistency of the $E(x{:}\mathbb{D})$ without `AMO` constraint. For the binary encoding, this approach is more practical, as variables for which the initial domain already satisfies $\mathrm{bit}(y,k)$ are rare.

Theorem 3.6 UP on the clauses of Eq. 3.12 and Eq. 3.14 is PC for constraint $x \in D(x) \wedge x = y \wedge y \in D(y)$ and $\mathcal{L} = \text{DIRECT}(x) \cup \text{BINARY}(y)$. Hence, it enforces domain consistency on x.

Proof. First we show that UP enforces the exactly-one constraint. If we have $[\![x = d]\!]$ and $[\![x = d']\!]$ true simultaneously, then Eq. 3.12 will force one binary variable in y to take two values, thus failing. If all variables $[\![x = d']\!]$ are set false except one $[\![x = d]\!]$, then Eq. 3.14 will set the correct bits of y in the remaining solution (via setting the negation to false). The forward direction will then propagate $[\![x = d]\!]$.

Next we show that the UP is PC with respect to the binary variables and missing values in the original domain. Suppose $S \subseteq D(x)$ are the remaining values in the domain. Then $\neg[\![x = d]\!]$ is set for $d \in D(x) - S$. Suppose that there is no support for $[\![\text{bit}(y, k)]\!]$ in S. Then UP of Eq. 3.14 will set $\neg[\![\text{bit}(y, k)]\!]$.

Finally, we show that UP is PC on the encoding variables. Given the subset of the current assignment literals $L \subseteq \varphi$ restricted to the direct encoding variables for x and binary encoding variables of y. Let $S \subset D(x)$ be the set of possible domain values remaining, i.e. $S = \{d \mid d \in D(x), \neg[\![x = d]\!] \notin L\}$. Clearly the direct encoding variables are consistent with this by definition, and if S is a singleton, by the first argument $[\![x = d]\!]$ is set true. We now consider each bit variable $[\![\text{bit}(y, k)]\!]$: if $[\![\text{bit}(y, k)]\!] \in L$ then by Eq. 3.12 each value in $d \in S$ has $\text{bit}(d, k)$ is true, hence it is supported; similarly if $\neg[\![\text{bit}(y, k)]\!] \in L$. Finally, if variable $[\![\text{bit}(y, k)]\!]$ does not appear in L, then there must be values in S with both truth values, otherwise one of the equations in Eq. 3.14 would have propagated. □

Coupling $x{:}\mathbb{O} = y{:}\mathbb{B}$

We are not aware of any previous channelling between order and binary representations.

Theorem 3.7 The following function correctly encodes $x{:}\mathbb{O} = y{:}\mathbb{B}$:

$$E(x{:}\mathbb{O} = y{:}\mathbb{B}) \equiv E(y{:}\mathbb{O}) \wedge \forall_{k \in \mathcal{B}(y)} (\text{bit}(x{:}\mathbb{O}, k) \leftrightarrow \text{bit}(y{:}\mathbb{B}, k)) \tag{3.16}$$

This encoding is evidently correct since it enforces a simple relationship between the bit constraints on x and y. We have seen how both bit on x and y encodes to formulas which represent the constraint.

To show Eq. 3.16 yields a SAT encoding, we expand its decomposition, before we prove it is PC. In Eq. 3.6.1, we see how $E(\mathrm{bit}(x{:}\mathbb{O}, k))$ yields a set membership constraint of the form $x{:}\mathbb{O} \in S_k$. Thus, the encoding is equivalent to:

$$\forall_{k \in \mathcal{B}(y)}(x{:}\mathbb{O} \in S_k \leftrightarrow \mathrm{bit}(y{:}\mathbb{B}, k)) \tag{3.17.1}$$

$$\equiv \forall_{k \in \mathcal{B}(y)}((x{:}\mathbb{O} \in S_k \rightarrow \mathrm{bit}(y{:}\mathbb{B}, k)) \wedge (x{:}\mathbb{O} \notin S_k \rightarrow \neg\mathrm{bit}(y{:}\mathbb{B}, k))) \tag{3.17.2}$$

Let $[l_{k,i}..r_{k,i}]$ denote the i-th maximal interval of S_k (see Eq. 3.8). The encoding of Eq. 3.16 yields the following clauses:

$$\bigwedge_{k \in \mathcal{B}(y)} \left(\left(\bigvee_{[l_{k,i}..r_{k,i}] \in S_k} x \in [l_{k,i}..r_{k,i}] \right) \rightarrow \mathrm{bit}(y, k) \right)$$
$$\equiv \bigwedge_{k \in \mathcal{B}(y), [l_{k,i}..r_{k,i}] \in S_k} \left(x \notin [l_{k,i}..r_{k,i}] \vee \mathrm{bit}(y, k) \right)$$

Finally, encoding $x \notin [l_{k,i}..r_{k,i}]$ as $[\![x < l_{k,i}]\!] \vee [\![x > r_{k,i}]\!]$, and $\mathrm{bit}(y, k)$ as $[\![\mathrm{bit}(y, k)]\!]$, yields ternary clauses:

$$\bigwedge_{k \in \mathcal{B}(y), [l_{k,i}..r_{k,i}] \in S_k} ([\![x < l_{k,i}]\!] \vee [\![x > r_{k,i}]\!] \vee [\![\mathrm{bit}(y, k)]\!]) \tag{3.18}$$

The other direction, $x{:}\mathbb{O} \notin S_k \rightarrow \neg\mathrm{bit}(y{:}\mathbb{B}, k)$, follows a similar pattern. Thus, the encoding of Eq. 3.16 requires $O(|D(x)|\mathcal{B}(y))$ clauses, each having at most three literals.

Example 3.4 The following $E(x{:}\mathbb{O} \subseteq [0..6] = y{:}\mathbb{B} \subseteq [0..6])$ for $k \equiv 1$ yields the constraint $\mathrm{bit}(x, 1) \leftrightarrow \mathrm{bit}(y, 1)$. Encoding the $\leftarrow$ direction yields unary constraints:

$$\begin{aligned}
&\mathrm{bit}(x, 1) \leftarrow \mathrm{bit}(y, 1) \\
\equiv\ &\neg\mathrm{bit}(x, 1) \rightarrow \neg\mathrm{bit}(y, 1) \\
\equiv\ &x \notin \{2, 3, 6\} \rightarrow \neg\mathrm{bit}(y, 1) \\
\equiv\ &x \in \{0, 1, 4, 5\} \rightarrow \neg\mathrm{bit}(y, 1) \\
\equiv\ &(x \in [0..1] \vee x \in [4..5]) \rightarrow \neg\mathrm{bit}(y, 1) \\
\equiv\ &(x \in [0..1] \rightarrow \neg\mathrm{bit}(y, 1)) \wedge (x \in [4..5] \rightarrow \neg\mathrm{bit}(y, 1)) \\
\equiv\ &(x \notin [0..1] \vee \neg\mathrm{bit}(y, 1)) \wedge (x \notin [4..5] \vee \neg\mathrm{bit}(y, 1)) \\
\equiv\ &(x < 0 \vee x > 1 \vee \neg\mathrm{bit}(y, 1)) \wedge (x < 4 \vee x > 5 \vee \neg\mathrm{bit}(y, 1))
\end{aligned}$$

These unary constraints are then encoded for $x{:}\mathbb{O}$ and $y{:}\mathbb{B}$ by Eq. 3.5.2 and Eq. 3.6.2:

$$([\![x < 0]\!] \vee [\![x > 1]\!] \vee \neg[\![\mathrm{bit}(y, 1)]\!]) \wedge ([\![x < 4]\!] \vee [\![x > 5]\!] \vee \neg[\![\mathrm{bit}(y, 1)]\!])$$

Similarly, encoding the $\rightarrow$ direction, i.e. $x{:}\mathbb{O} \in \{2, 3, 6\} \rightarrow \mathrm{bit}(y{:}\mathbb{B}, 1)$, yields:

$$[\![x < 2]\!] \vee [\![x > 3]\!] \vee [\![\mathrm{bit}(y, 1)]\!] \wedge [\![x < 6]\!] \vee [\![x > 6]\!] \vee [\![\mathrm{bit}(y, 1)]\!]$$

The complete encoding is:

$$\begin{aligned}
&[\![x < 0]\!] \vee [\![x > 1]\!] \vee \neg[\![\mathrm{bit}(y, 1)]\!] \wedge \\
&[\![x < 2]\!] \vee [\![x > 3]\!] \vee \quad [\![\mathrm{bit}(y, 1)]\!] \wedge \\
&[\![x < 4]\!] \vee [\![x > 5]\!] \vee \neg[\![\mathrm{bit}(y, 1)]\!] \wedge \\
&[\![x < 6]\!] \vee [\![x > 6]\!] \vee \quad [\![\mathrm{bit}(y, 1)]\!]
\end{aligned}$$

Theorem 3.8 UP on the clauses of Eq. 3.16 and Eq. 2.20 is PC for constraint $x \in D(x) \wedge x = y$ and language $\mathcal{L} = \mathrm{ORDER}(x) \cup \mathrm{BINARY}(y)$. Hence, it enforces the initial bounds on x.

Proof. Given the subset of the current assignment literals $\mathcal{A} \subseteq \varphi$ restricted to the order encoding variables for x and binary encoding variables for y, we show there is support for each possible value defined by $\mathcal{A}$. Since each bound or bit is supported or propagated, propagation completeness follows. The order literals in $\mathcal{A}$ define a range $[l..u]$ given by $l = \max(\{i \mid [\![x \geq i]\!] \in \mathcal{A}\})$ and $u = \min(\{i - 1 \mid \neg[\![x \geq i]\!] \in \mathcal{A}\})$.

The order consistency of Eq. 2.20 enforces that $\neg[\![x \geq i]\!] \in \mathcal{A}$ for $i > u$ and $[\![x \geq i]\!] \in \mathcal{A}$ for $i < l$. We show l and u are supported. Suppose that $([\![\mathrm{bit}(y, k)]\!] \mapsto \neg\mathrm{bit}(d, k)) \in \mathcal{A}$ for some bit k. Then, l must be in some maximal interval $l \in [l_{k,i}..r_{k,i}] \in S_k$. There exists a corresponding clause $[\![x < l_{k,i}]\!] \vee [\![x > r_{k,i}]\!] \vee [\![\mathrm{bit}(y, k)]\!]$, for which UP will set $[\![x \geq r_{k,i}]\!]$. Hence, l cannot be the lower bound. A similar argument applies to the upper bound.

We now show each possible value for each binary encoding variable $[\![\mathrm{bit}(y, k)]\!]$ is supported in the range $[l..u]$. Suppose to the contrary that w.l.o.g. $[\![\mathrm{bit}(y, k)]\!]$ is not true for any $y \in [l..u]$. Then there is a maximal interval $[l_{k,i}..r_{k,i}]$ at least as large as $[l..u]$ for $\neg[\![\mathrm{bit}(y, k)]\!]$. Then the clause $\neg[\![\mathrm{bit}(y, k)]\!] \vee [\![x < l_{k,i}]\!] \vee [\![x > r_{k,u}]\!]$ will propagate $\neg[\![\mathrm{bit}(y, k)]\!]$, which then must be in $\mathcal{A}$. The same reasoning holds for the negation.

The channelling also enforces the outer bounds on the binary encoding variables. Clearly $d_1 \leq l \leq u \leq d_m$ so the arguments for support of binary encoding variables automatically take into account the initial bounds. □

3.3.4 Coupling inequality constraints

While the channelling constraints above are PC, introducing a dual representation of a variable and its channelling constraint incurs overhead. When we can avoid this and directly encode a constraint that *couples* two representations we can get more compact models.

Coupling binary inequality constraints, $x \leq y$, is a special case of coupling linear constraints, as seen in Def. 3.2. The SAT encoder `diet-sugar` (see Sec. 3.2.2) has an alternative approach for comprehensively coupling $\mathbb{E} \in \{\mathbb{O}, \mathbb{B}\}$. Applied to binary inequality constraints, it encodes $ax + b_1y_1 + \cdots + b_ry_r \leq c$ by essentially encoding the constraints $a \cdot d_i[\![x \geq d_i]\!] + b_1y_1 + \cdots + b_ry_r \leq c$ for each $d_i \in D(x)$ separately, encoding resulting PB using well-known methods. Although they do not prove it, using a PC encoding (such as Binary Decision Diagram (BDD) with the `CompletePath` encoding [110]) will guarantee the encoding is PC on the language $\text{ORDER}(x) \cup \text{BINARY}(y_1) \cup \cdots \cup \text{BINARY}(y_r)$.

However, rather than introducing $O(|D(x_i)|)$ PB constraints, we can couple the encoding variables directly for a binary inequality constraint, $x \leq y$.

Definition 3.5 A correct encoding of $E(x{:}\mathbb{O} \leq y{:}\mathbb{B})$ is:

$$E(x{:}\mathbb{O}) \wedge E(y{:}\mathbb{B}) \wedge E(y{:}\mathbb{B} \geq \text{lb}(x)) \wedge E(x{:}\mathbb{O} \leq \text{ub}(y)) \wedge \tag{3.19.1}$$

$$\bigwedge_{\text{lb}(x) \equiv d+1}^{\text{ub}(x)} \left([\![x \geq d]\!] \rightarrow \bigvee_{k \in \mathcal{B}(x), \neg\text{bit}(d-1,k)} [\![\text{bit}(y,k)]\!] \right) \tag{3.19.2}$$

The intuition behind this is that the binary representation for d satisfies this because adding any positive amount to $d-1$ will flip at least one bit from false to true. The encoding consists of $O(|\text{ub}(x) - \text{lb}(x)|)$ clauses of size $O(|\mathcal{B}(y)|)$. If $D(x)$ is not contiguous, then many clauses within domain gaps become redundant. To complete the encoding we need to enforce the initial bounds on both variables.

Example 3.5 Consider encoding $x{:}\mathbb{O} \leq y{:}\mathbb{B}$ for variables x and y with $D(x) = [1..7]$, $D(y) = [0..6]$. This will result in the following clauses from Eq. 3.19, shown with the relevant bit representation on the left.

$$
\begin{array}{lllll}
001_2 & [\![x \geq 2]\!] \rightarrow [\![\mathrm{bit}(y,2)]\!] \vee [\![\mathrm{bit}(y,1)]\!] & & \\
010_2 & [\![x \geq 3]\!] \rightarrow [\![\mathrm{bit}(y,2)]\!] & & \vee [\![\mathrm{bit}(y,0)]\!] \\
011_2 & [\![x \geq 4]\!] \rightarrow [\![\mathrm{bit}(y,2)]\!] & & \\
100_2 & [\![x \geq 5]\!] \rightarrow & [\![\mathrm{bit}(y,1)]\!] \vee [\![\mathrm{bit}(y,0)]\!] & \\
101_2 & [\![x \geq 6]\!] \rightarrow & [\![\mathrm{bit}(y,1)]\!] &
\end{array}
$$

The encoding also requires $y \geq 1$ encoded as $[\![\mathrm{bit}(y,0)]\!] \vee [\![\mathrm{bit}(y,1)]\!] \vee [\![\mathrm{bit}(y,2)]\!]$ and $x \leq 6$ encoded as $\neg[\![x \geq 7]\!]$. We see that for example $[\![x \geq 4]\!]$, which forces $[\![\mathrm{bit}(y,2)]\!]$ true, means that it never occurs for later literals. This relies on the order encoding, since $[\![x \geq 5]\!]$ implies $[\![x \geq 4]\!]$.

If we suppose instead $D(x) = \{1, 2, 6, 7\}$, then we keep only the first, third and last clauses, since the others are subsumed (i.e., $[\![x \geq 3]\!] \equiv [\![x \geq 4]\!] \equiv [\![x \geq 5]\!] \equiv [\![x \geq 6]\!]$).

Theorem 3.9 UP on the clauses of Eq. 3.19, Eq. 2.20 and clauses for lexicographic encoding of $y \geq \mathrm{lb}(x)$ and together with the clause $\neg[\![x \geq \mathrm{ub}(y) + 1]\!]$ is PC for $x \leq y$ for the language $\mathcal{L} = \mathrm{ORDER}(x) \cup \mathrm{BINARY}(y)$.

Proof. Suppose $\mathcal{D} \wedge x \leq y \rightarrow [\![x \leq d]\!]$; we show that UP will enforce this. The initial case given by $y \leq \mathrm{ub}(x)$ is enforced by the last clause. Let d be the maximum value y can take given $\mathcal{D}$, then we know that $\neg[\![\mathrm{bit}(y,k)]\!] \in \mathcal{D}$ for all $k \in \mathcal{B}(y), \mathrm{bit}(d,k) \equiv 0$ otherwise y could take a larger value. The clause Eq. 3.19 for $[\![x \geq d+1]\!]$ has right-hand side $\bigvee_{k \in \mathcal{B}(y), \neg \mathrm{bit}(d,k)} [\![\mathrm{bit}(y,k)]\!]$ and hence propagates $\neg[\![x \geq d+1]\!]$ as required.

Let Y be the set of all possible values that y can take given $\mathcal{D}$. Suppose $\mathcal{D} \wedge x \leq y \rightarrow [\![\mathrm{bit}(y,k)]\!]$ for some $k \in \mathcal{B}(y)$. We show that UP will enforce this. No other propagation is possible, since the inequality cannot force a bit to be 0. If x still sits at its initial lower bound this will be forced by the encoding of $y \geq \mathrm{lb}(x)$. Otherwise, this propagation was caused by $[\![x \geq d]\!]$ in the current domain, where $d > \min(Y)$

So clearly (A) $\mathrm{bit}(v,k) \equiv 1$ for all $v \in Y \cup [d..\max(y)]$. Let (B) $d' \equiv \max(\{d'' \mid d'' \in Y, \mathrm{bit}(d'',k) \equiv 0\})$. Clearly such a d' exists otherwise we would already have propagated $[\![\mathrm{bit}(y,k)]\!]$, and since $d' < d$, $[\![x \geq d'+1]\!]$ is propagated by Eq. 2.20. We claim the clause Eq. 3.19 for $[\![x \geq d'+1]\!]$ will propagate $[\![\mathrm{bit}(y,k)]\!]$. Suppose to the contrary then there is another bit k' where $\mathrm{bit}(d',k') \equiv 0$ where $\neg[\![\mathrm{bit}(y,k')]\!] \notin \mathcal{D}$. Then $d' + 2^{k'} \in Y$ and either $d' + 2^{k'} \geq d$ contradicting (A), or $d' + 2^{k'} < d$ contradicting (B).

While it is not possible for $\mathcal{D} \wedge x \leq y \rightarrow \neg[\![\mathrm{bit}(y,k)]\!]$ since a lower bound can never force a negative bit (although this can happen subsequently from the clauses for constraint $y \leq \mathrm{ub}(y)$). □

Interestingly, the encoding of `diet-sugar` applied to the constraint $x{:}\mathbb{O} \leq y{:}\mathbb{B}$ produces a strict superset of our encoding clauses, adding many redundant clauses.

We can enforce the reverse $x{:}\mathbb{O} \geq y{:}\mathbb{B}$ similarly. Apart from the initial bounds constraints $x \geq \mathrm{lb}(y)$ and $y \leq \mathrm{ub}(x)$ we generate the clauses for $\mathrm{lb}(y) < d \leq \mathrm{ub}(x)$

$$\neg[\![x \geq d]\!] \rightarrow \bigvee_{k \in \mathcal{B}(y), \mathrm{bit}(d,k)} \neg[\![\mathrm{bit}(y,k)]\!]. \tag{3.20}$$

We can of course use the conjunction of inequalities, $x{:}\mathbb{O} \geq y{:}\mathbb{B} \wedge x{:}\mathbb{O} \leq y{:}\mathbb{B}$, to enforce equality $x{:}\mathbb{O} = y{:}\mathbb{B}$, but this *inequality channel* does not enforce that the encoding is PC.

3.3.5 Coupling element constraints

Element constraints are an important component of many CP models. They enable array look-up using a variable index with $y = A[x]$. This can be expressed using the unary constraints equality constraints of Eq. 3.4:

$$\forall_{d \in D(x)}((x = d) \rightarrow (A[d] = y))$$

Consequently, we can define a coupling encoding of the element constraint for any encoding choices for y and x, and each individual variable in A. We denote $A{:}\mathbb{E}$ if all its variables have encoding $\mathbb{E}$.

Example 3.6 Consider the coupling for $y{:}\mathbb{O} = A{:}\mathbb{B}[x{:}\mathbb{D}]$. The clauses take the form of

$$[\![x \neq d]\!] \vee \left([\![\mathrm{bit}(A[d],k)]\!] = \mathrm{bit}\left(l_{k,i},k\right)\right) \vee [\![y < l_{k,i}]\!] \vee [\![y > r_{k,i}]\!]$$

for $d \in D(x)$, maximal interval $\left[l_{k,i}..r_{k,i}\right] \subseteq \mathcal{R}(y)$, and bit $k \in \mathcal{B}(A_d)$.

When x has the direct encoding and A is fixed, the encoding can be made PC similar to Eq. 3.14 by adding a backwards clauses for every distinct value:

$$\forall_{u \in A}\left((y = u) \rightarrow \bigvee_{d \in D(x), u = A[d]} [\![x = d]\!]\right)$$

3.3.6 Views

Views are a crucial feature in modern CP solvers [111]. For specific constraints, views can simplify propagation construction using an interface to the variable operations. In SAT, whenever an equivalence or negation arises between different expressions, we can represent both with the same Boolean variable. This also allows greater scope for views on SAT encoded integers compared to CP integers.

As mentioned in Sec. 3.2.2, the `BEE` encoder finds views during the encoding process with a technique called Boolean equi-propagation. This technique is more generalized and powerfull than views by itself, since it can find equivalences of literals from different constraints. However, the views discussed in this section apply directly on constraints without the need for in-processing.

Affine transformations

The most important view constraint is the affine transformation $y = ax + b$ for constant integers a and b. For the direct encoding, we have for every $d \in D(x) \cup D(y)$ that $[\![y = ad + b]\!] \equiv [\![x = d]\!]$, so we can use the same Boolean variable to represent both. For the order encoding, we have for non-negative a that $[\![y \geq ad + b]\!] \equiv [\![x \geq d]\!]$ (if a is negative, one $\geq$ flips to $\leq$). For the binary encoding, affine views can be applied in some but not all cases, such as when we multiply by a power-of-two which is equivalent to a left-shift operation on the bits [112].

Minimum/maximum

Boolean encoding also raises the possibility of *partial views* where only some Boolean variables are reused. In $y = \max(x, m)$, enforcing y to take the greater value between integer variable x and integer constant m (and similar for $\min$), after enforcing $y \geq m$ we find a partial view for the order encoding $\forall_{d=m}^{\mathrm{ub}(y)} [\![y \geq d]\!] \equiv [\![x \geq d]\!]$. For the direct encoding, we have one less equivalence: $\forall_{d=m+1}^{\mathrm{ub}(y)} [\![y = d]\!] \equiv [\![x = d]\!]$. We require one new Boolean variable for $[\![y = m]\!]$, which is constrained by $\forall_{d=\mathrm{lb}(x)}^{m} [\![x = d]\!] \rightarrow [\![y = m]\!]$.

Element

The `element` constraint $y = A[x]$ enforces y to take the x-th value of integer array A. For some fixed compositions of A, the constraint allows for views. For $y{:}\mathbb{D} = A[x{:}\mathbb{D}]$, every unique value $A[d]$ has binary backwards clause $[\![y = A[d]]\!] \rightarrow [\![x = d]\!]$, so $[\![x = d]\!] \equiv [\![y = A[d]]\!]$. For $y{:}\mathbb{O} = A[x{:}\mathbb{O}]$, we have for all $d \in D(x)$ that $[\![x \geq d]\!] \rightarrow [\![y \geq A[d]]\!]$ if $A[d] \leq A[e]$ holds for all $e > d$. Similarly, $[\![x \leq d]\!] \rightarrow [\![y \leq A[d]]\!]$ if $A[c] \leq A[d]$ holds for all $c < d$. If both conditions hold, then $[\![x \geq d]\!] \equiv [\![y \geq A[d]]\!]$. If A is strictly monotone, both conditions hold for all elements and y becomes a total view of x. When the inequalities are flipped, we have negated views $[\![x \geq d]\!] \equiv [\![y < A[d]]\!]$, which are total if A is strictly antitone.

Views allow us to extend our coupling encodings straightforwardly. For example, we can encode $x{:}\mathbb{O} + d \leq y{:}\mathbb{B}$ by using a view to construct $(x + d){:}\mathbb{O}$ and then using the inequality coupling.

3.4 Experimental Results

As discussed in Sec. 2.4, we typically express CSPs in a modelling language such as MiniZinc. To illustrate, we show the model of the first case study (Sec. 3.4.1) well-known Knight's tour problem (`knights`) in which we aim to find a trajectory of n^2 legal knight moves (see Fig. 3.1) around an $n \times n$ board. The knight starts and ends at the top-left square (numbered 1), visiting each square exactly once.

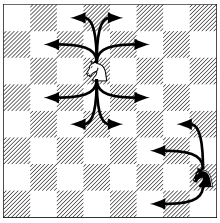


Fig. 3.1: Legal knight moves

```
1  int: n; % size of board
2  array[int] of var int: x = [ let
3    { % legal knight moves from position r, c
4    set of int: neighbours = { n*(r_i-1) + c_i
5      | i in 1..8,
6        r_i in {r + [-1,-2,-2,-1,1,2,2,1][i]}
7               where r_i in 1..n,
8        c_i in {c + [-2,-1,1,2,2,1,-1,-2][i]}
9               where c_i in 1..n };
10   var neighbours: x_r_c;
11   } in x_r_c | r,c in 1..n ];
12 % fix first and last moves (symmetry)
13 constraint x[1] = n+3;
14 constraint x[2*n+2] = 1;
15 include "circuit.mzn";
16 constraint circuit(x);
```

Lst. 3.1: A base MiniZinc model for `knights` (`./knights.mzn`)

The variable `x[p]`, declared in lines 2-11, gives the next position (in $[1..n^2]$) to move to from position `p`. MiniZinc allows us to specify the integer variable domains exactly according to legal knight moves. The symmetry breaking constraints in lines 13 and 14 reduce the domains of first and last variable to their single, fixed values, respectively. The `circuit` global constraint in lines 15-16 constrains the `x` variables to describe a single Hamiltonian circuit traversal of the graph.

To validate the benefit of coupling different integer encodings, we created the MiniZinc-SAT (`mzn-sat`) framework. Through MiniZinc's *annotations*, the user can declare the integer variable encodings. For example in the Knight's tour model, we can choose the direct encoding for each variable in `x` by adding an annotation as follows.

```
var neighbours: x_r_c ::direct_encoded;
```

Since annotations are first class objects, the user can specify more complex encoding schemes based on, for example, the variable's domain size, its existing encodings, or any other contextual rules. If no encodings are specified, default choices are made. During compilation, `mzn-sat` resolves mixed encoding constraints by coupling if possible, or by channelling if necessary. Additionally, views are used whenever the constraint and variable encodings allow it.

Using `mzn-sat`, we create various encodings of realistic problems to see if we can improve solver performance. We also compare with Picat-SAT (`picat-sat`) [90] 3.1, a solver that maps MiniZinc to SAT using binary encoding, and `chuffed` 0.10.4, a lazy clause generation solver, using lazy direct-order encoding.

Results are given for SAT models and MaxSAT models by running `Open-WBO`, version 2.1, on a single-core *Intel Xeon 8260* CPU (non-hyperthreaded) with a 10-minute solving timeout and up to 64 GB of RAM. The results are visualized in cactus plots, in which for each configuration the solved instances are sorted by solve time (measured from when the SAT or LCG solver is started). Instances which are not solved (for SAT) or proved optimal (for MaxSAT) within the time limit are omitted. Source code, models, instances and run logs are available online.[7]

[7] https://github.com/hbierlee/cpaior-2022-coupling-sat

3.4.1 Knight's tour

The first example is the `knights` model shown in Lst. 3.1 for instances $n = \{8, 10, ..., 22\}$. The model contains n^2 variables x with relatively sparse domains of up to size 8, which we can annotate as in Lst. 3.2, from the base `knights.mzn` model. For `mzn-sat`, we adapt the standard library's `circuit` decomposition:

```
1  include "all_different.mzn";
2
3  predicate fzn_circuit(array[int] of var int: x) =
4    let {
5      set of int: S = index_set(x);
6      int: l = min(S);
7      int: n = card(S);
8      array[S] of var 1..n: y::order_encoded;
9    } in forall(i in S)(x[i] != i) /\ alldifferent(x)
10         y[l] = n /\ y[x[l]] = 1 /\
11         forall(i in S diff {l})(y[i] = y[x[i]]-1);
```

Lst. 3.3: A `circuit` decomposition adapted from `std/fzn_circuit.mzn`

To prevent sub-cycles, the decomposition in Lst. 3.3 introduces another n^2 variables, y, with contiguous domains of $[1..n^2]$ that represent the order in which positions are visited. In line 8, we can annotate y's declaration to control its encoding (e.g. `order_encoded`). The encodings of x and y are coupled by a combination of constraints which enforce a bijection between the two. Because of the bijection, only one `all_different` constraint is needed on either x or y. The `all_different` encoding uses disequalities which are stronger for the direct encoding. This allows us to omit the `all_different` constraint for whichever is direct encoded (unless neither are). In this example, we have omitted the `all_different`(y) constraint accordingly. Finally, note that the affine constraint, `y[i]-1`, admits a view for a unary encoding (see Sec. 3.3.6), but any other encoding of x and y are also possible.

The results for the three uniform and three sensible mixed encoding choices for x and y are shown in Fig. 3.2.1. Uniform order and binary encodings do not succeed beyond the two easiest instances. This is unsurprising since the `circuit` constraint strongly prefers x:$\mathbb{D}$ because of its sparse domains, the equalities in `element`, and the disequalities in the `all_different` encoding. However, since the y variables reason about the order of visits, we see that y:$\mathbb{O}$ is clearly preferred over the uniform approach, y:$\mathbb{D}$. The worst choice is y:$\mathbb{B}$, which has the additional disadvantage that it cannot use the affine transformation view. Creating a redundant order encoding for x variables (written x:$\mathbb{D}$:$\mathbb{O}$) does not seem to make much difference. The control solvers `chuffed` and `picat-sat` both far outperform `mzn-sat`, since they have native propagator and encoding [102] respectively.

3.4.2 Orienteering

The `orienteering` problem is a Constrained Optimization Problem (COP) concerning a complete graph with edge distances $d_{i,j}$ and node rewards. The aim is to find a path from start to finish node that maximizes the sum of the rewards of the visited nodes, but which is limited by a linear inequality on the distances of the traversed edges. The principal `subcircuit` constraint is encoded similarly to in `knights` by two sets of variables x and y, coupled through `element` constraints. Given the results from `knights`, we will use $x{:}\mathbb{D}$ and $y{:}\mathbb{O}$.

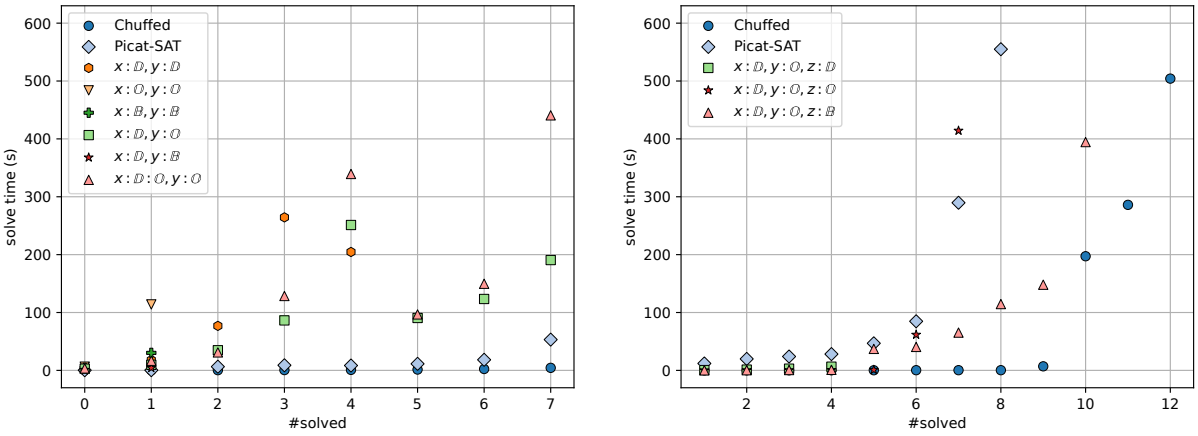
However, in Lst. 3.4, we show how we can experiment with all possible (non-redundant) encodings on a new set of coupled variables, namely the result variables of the element constraint $z_i = d_i[x_i]$. These are used in a linear inequality to impose the maximum distance, `MAX_DISTANCE`. We can annotate the result of the `element` constraint directly (without introducing the z variables in the base `orienteering.mzn` model)

```
% distance from node i to j
array[1..n,1..n] of int: d;
constraint sum(i in 1..n)(
  d[i,x[i]]::binary_encoded % = z[i]
) <= MAX_DISTANCE;
```

Lst. 3.4: A partial `orienteering.mzn` base model, which applies the binary encoding $z{:}\mathbb{B}$ on the result of an `element` constraint.

A cactus plot comparing encodings is shown in Fig. 3.2.2. Of the mixed encodings, $z{:}\mathbb{B}$ is clearly the best, since it makes the path length constraint compact.

While both **`chuffed`** and **`picat-sat`** again have native propagator and encoding, **`picat-sat`** is now outperformed by the best **`mzn-sat`** encoding. It seems that its encoding is less effective than our encoding of the MiniZinc standard decomposition using the element coupling.



3.2.1 Knights 3.2.2 Orienteering

Fig. 3.2: Cactus plots for solved `knights` instances and `orienteering` instances.

3.4.3 Job-shop scheduling with weighted earliness/lateness

In this case study, we examine a job-shop scheduling problem `jsswet` over n jobs, each with m tasks, with start time variables x. Every task j of job i runs on a different machine for its full duration $d_{i,j}$. Tasks of the same job must finish in sequence ($x_{i,j} + d_{i,j} \leq x_{i,j+1}$), and must not overlap (`disjunctive`) with the tasks of other jobs that run on the same machine. The objective is to minimize the sum of the penalties over all jobs. A job's penalty is its final task's earliness or lateness to its deadline t_i, weighted by its earliness e_i or lateness l_i penalty coefficients, respectively. The order encoding combines affine transformations and max views for the objective: $e_i \cdot \max(0, t_i - x_{i,m} + d_i){:}\mathbb{O} + l_i \cdot \max(0, x_{i,m} + d_i - t_i){:}\mathbb{O}$.

In preliminary results not shown here we found that if the schedule's horizon is small, the order encoding works best thanks to its propagation strength. However, when the horizon becomes large this approach will run out of memory due to the sheer number of order encoding variables, and only the binary encoding remains viable. So each encoding has benefits and drawbacks. We now consider two variants of the problem.

In the first variant, we consider a large horizon, but designate the first 40% of jobs as *high priority* jobs which must finish within 500 time steps of their deadline, and have much higher penalties. Effectively, this splits the variables into those with small and large domains. We consider encoding all variables with order, or binary, or a mix which uses order for small and binary for large domains. The modeller can instate a simple rule to dynamically apply the appropriate encoding to variable $x_{i,j}$, based on the size of its domain $D_{i,j}$ given as a set of integer values of each individual integer variable:

```
% Given: D[i,j] = domain of job i, task j
array[1..n,1..m] of var int: x =
[ (i,j): let {
  ann: encoding =
    if card(D[i,j]) <= CUTOFF then  order_encoded
                               else binary_encoded
    endif;
  % start time for job i, task j with domain D[i,j]
  var D[i,j]: x_i_j::encoding;
} in x_i_j | i in 1..n, j in 1..m ];
```

After preliminary experiments, the `CUTOFF` parameter was set to 500. For high priority job-task $x_{a,j}$ and low priority job-task $x_{b,k}$ assigned to the same machine, the `disjunctive` constraint couples the (potentially) mixed encodings:

$$\big((x_{a,j} + d_{a,j}){:}\mathbb{O} \leq x_{b,k}{:}\mathbb{B}\big) \vee \big(x_{b,k}{:}\mathbb{B} \leq (x_{a,j} - d_{b,k}){:}\mathbb{O}\big)$$

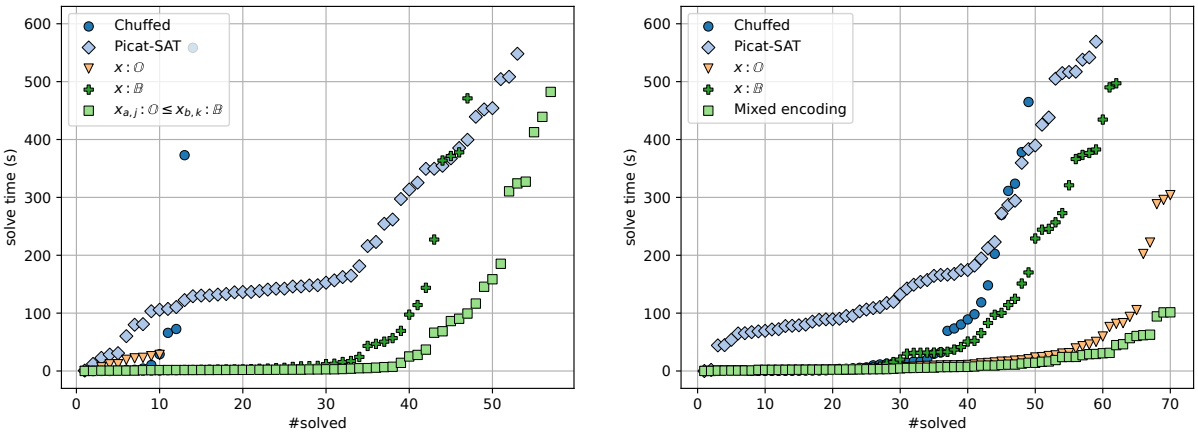
Note that we always resolve the constant $d(a, j)$ or $d(b, k)$ with the order encoded variable, to maximally leverage its support of the affine view. The results in Fig. 3.3.1 show that the mixed encoding outperforms the other solvers and uniform encodings.

For the other variant we consider a smaller horizon, which allows a full order encoding of all tasks. However, now the average run time of all jobs is limited by parameter M (2.5 times the sum of all minimal job durations). To effectively constrain this linear inequality $\sum_{i=1}^{m} x_{i,m} + d_{i,m} - x_{i,1} \leq M$, the mixed encoding adds a redundant binary encoding only to the first and last task of each job in Lst. 3.6.

```
array[1..n,1..m] of var 0..horizon: x::order_encoded;
constraint sum(i in 1..n)(
    x[i,max(tasks)]::binary_encoded + d[i,max(tasks)]
  - x[i,min(tasks)]::binary_encoded
) <= M;
```

Lst. 3.6: A partial `jsswet.mzn` base model, which constrains the maximum (average) runtime of all jobs by applying a redundant binary encoding on the first and last of the order encoded x variables, i.e. $x_{i,1}{:}\mathbb{O}{:}\mathbb{B}, x_{i,2}{:}\mathbb{O}, ..., x_{i,m-1}{:}\mathbb{O}, x_{i,m}{:}\mathbb{O}{:}\mathbb{B}$.

The results in Fig. 3.3.2 show the mixed encoding convincingly outperforms the other solvers and uniform encodings. The redundant encodings are coupled with $x_{i,1}{:}\mathbb{O} = x_{i,1}{:}\mathbb{B}$, but coupling with $x_{i,1}{:}\mathbb{O} \leq x_{i,1}{:}\mathbb{B} \wedge x_{i,1}{:}\mathbb{O} \geq x_{i,1}{:}\mathbb{B}$ or just $x_{i,1}{:}\mathbb{O} \leq x_{i,1}{:}\mathbb{B}$ produces similar results.



3.3.1 Variant where 40% high-priority jobs

3.3.2 Variant with limited average job length

Fig. 3.3: Cactus plots for solved `jsswet` instances

3.4.4 Table Layout

Finally, we consider the `table-layout` problem. For a table composed of $n \times m$ cells, our task is to assign a width-height configuration variable $x_{i,j}$ for each cell at row i, column j. The configuration will determine for cell i, j its cell-width $w_{i,j} = W[x_{i,j}]$ and cell-height $h_{i,j} = H[x_{i,j}]$ through `element` constraints. These variables are combined in n row-height variables

r_i, where $\bigwedge_{j=1}^{m} r_i \geq h_{i,j}$, and m column-width variables c_j, where $\bigwedge_{i=1}^{m} c_j \geq w_{i,j}$. The objective is then to minimize the table's height $\sum_{i=1}^{n} r_i$, without the table's width $\sum_{j=1}^{m} c_j$ exceeding a given maximum width.

We chose this problem to test two unexplored properties. First, $H_{i,j}$ and $W_{i,j}$ are guaranteed to be respectively monotone and antitone, since the width-height configurations represent sorted, optimal text layouts. Consequently, the element constraint establishes a view between the order encoding variables $[\![x_{i,j} \geq k]\!]$, $[\![h_{i,j} \geq H_{i,j,k}]\!]$ and $[\![w_{i,j} \leq W_{i,j,k}]\!]$. Secondly, $D(h_{i,j})$ is sparse whereas $D(r_i)$ is contiguous. Thus, the coupling $h_{i,j}{:}\mathbb{O} \leq r_i{:}\mathbb{B}$ requires far fewer clauses for the domain gaps in $h_{i,j}$, while also skipping a large order encoding of r_i. We compare this very compact encoding against an order encoded objective (using simple binary clauses for $h_{i,j}{:}\mathbb{O} \leq r_i{:}\mathbb{O}$). A third approach creates an order encoding for the width ($w_{i,j}{:}\mathbb{O} \leq c_j{:}\mathbb{O}$) as well, but still redundantly channels $c_j{:}\mathbb{O} = c_j{:}\mathbb{B}$ for the linear inequality.

The results in Fig. 3.4 show that solver performance suffers greatly if we couple $x{:}\mathbb{O}$ to a binary encoded objective $r{:}\mathbb{B}$ rather than using straightforward order encoded objective $r{:}\mathbb{O}$. The coupling perhaps overcomplicates the objective compared to using binary clauses. Furthermore, channelling rather than coupling to $c{:}\mathbb{B}$ seems to be marginally better as well. The domain sizes of the `table-layout` instances are too large for the more compact coupling to pay off (in contrast to `jsswet`). For $r{:}\mathbb{O}$, the PB objective is unweighted, which means `Open-WBO` can use its core-guided Part-MSU3 algorithm. This makes the models solve all except the hardest problems instantly. `chuffed` does very well, solving all instances instantly, while `picat-sat` solves all instances but requires more time.

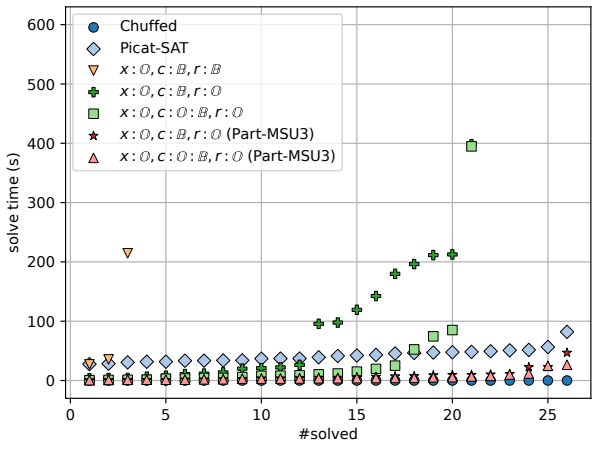


Fig. 3.4: Cactus plots for solved `table-layout` instances.

3.5 Conclusion and future work

In conclusion, while compilation to SAT is a competitive approach to tackling discrete optimization problems, efficient encoding of discrete optimization problems into SAT is challenging. To create the best encodings possible we must allow different representations of integers to be used in the encoding, which means the problem of coupling encodings arises. In this chapter we show how we can create coupled models by using channelling equalities, or directly by using coupled encodings of element or inequality constraints. Total and partial views extend the coupling constraints we can encode. We show that coupling encodings can be required for getting the best resulting SAT encoding.

An important direction of future work is the automated specification of variable encodings. Currently, the system aims to strike a middle ground between experts and non-experts by focusing on user-provided encoding specification, with basic common sense defaults. For non-experts (and in the setting of solver competitions), more autonomy is needed to be usable. In other works, heuristics or approaches based on Machine Learning (ML) have been effective on relatively simpler mixed encodings (e.g. choosing the best PB encoding [66]). Since our mixed encodings have more degrees of freedom, there is an opportunity to expand the existing work. Furthermore, heuristics based on dynamic solver behaviour and statistics (e.g. failures, or structure of learned clauses) seem to be unexplored in the context of encoding decisions.

— Chapter 4 —

Applying Single Constant Multiplication to SAT

This chapter explores the encoding of one particular constraint for one particular integer variable encoding: multiplication of a binary encoded integer variable x by constant c:

$$E(y{:}\mathbb{B} \trianglelefteq D^{\mathbb{Z}}(y{:}\mathbb{B}) = c \cdot (x{:}\mathbb{B} \trianglelefteq D^{\mathbb{Z}}(x{:}\mathbb{B})))$$

This constraint and its encoding are as important in optimization as it is for the hardware circuit design community, where the problem of creating an effective circuit for this constraint is known as the Single Constant Multiplication (SCM) problem. In particular, for hardware circuits it is beneficial to decompose the expensive multiplication circuit into cheaper operations: additions, subtractions, and multiplications by 2^s (i.e. a left-shift of s, see Sec. 3.3.6 on affine total views). For example, a circuit for $5 \cdot x$ is more efficiently constructed with the circuit $4 \cdot x + x$. Consequently, this chapter adapts techniques from circuit design to encode this constraint for Boolean Satisfiability (SAT) to see if benefits carry over from hardware to software.

The original inspiration for this chapter came from a workshop paper [113] at ModRef'20[8] (at CP'20), which proved for the first time (using a SAT solver) that 171,398,453 is the smallest c which requires a decomposition of at least 7 additions/subtractions. The preceding odd number of 171,398,451 still requires 6:

$$\begin{aligned}
2^7 \cdot x + x &= 129x \\
129x - 2^4 \cdot x &= 113x \\
2^4 \cdot 113x + x &= 1{,}809x \\
2^{19} \cdot 113x + 1{,}809x &= 59{,}246{,}353x \\
59{,}246{,}353x - 2^{14} \cdot 129x &= 57{,}132{,}817x \\
2^2 \cdot 57{,}132{,}817x - 57{,}132{,}817x &= 171{,}398{,}451x
\end{aligned}$$

In this chapter, we show how we constructed an optimal and complete database for a realistic range of constants using MiniZinc. Then, we can quickly encode SCM constraints in the

[8] https://modref.github.io/ModRef2020.html (accessed November 2024)

benchmark problems by simply looking up the optimal circuit for the given constant. As we will see in the experimental section, the resulting encoding show significant improvements in both encoding size and solve times.

This chapter is adapted from published and presented work at CPAIOR'24 [19]:

> H. Bierlee, J. J. Dekker, V. Lagoon, P. J. Stuckey, and G. Tack, "Single Constant Multiplication for SAT," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 21st International Conference, CPAIOR 2024, Uppsala, Sweden, May 28-31, 2024, Proceedings, Part I*, B. Dilkina, Ed., in Lecture Notes in Computer Science, vol. 14742. Springer, 2024, pp. 84-98. doi: 10.1007/978-3-031-60597-0_6.

This chapter is organized as follows. In Sec. 4.1, we introduce the problem and our approach. In Sec. 4.2, we give preliminaries specific to this chapter, and related work. In Sec. 4.3, we cover the main theoretical contribution: the various decompositions and encodings of the SCM constraint. In Sec. 4.4, we experimentally evaluate the various approaches. In Sec. 4.5, we conclude our findings and discuss future work.

4.1 Introduction

A fundamental constraint when encoding to SAT is the constant multiplication $y = c \cdot x$ of input and output integer variables x and y, and constant c. For instance, an efficient encoding of this constraint allows a strong encoding of the linear constraint. If x and y are encoded using a unary encoding (e.g. direct or order), then this operation is a view without overhead (see Sec. 3.3.6). However, the binary encoding only has a view for this constraint if the multiplication is by a power-of-two, 2^s, as it can be achieved by a left-shift of s on the bits. If x and y are binary encoded, but $c \neq 2^s$, the multiplication can still be avoided by creating a *decomposition* of *additions*, *subtractions* and 2^s-multiplications.

For example, the multiplication $5x$ can be decomposed into $4 \cdot x + x$. The most basic decomposition for a multiplication $c \cdot x$ therefore simply considers the binary representation of c and introduces one addend for every 1 in this binary representation - such as in the example of $5x = 4 \cdot x + x$ above - or $117x = 64 \cdot x + 32 \cdot x + 16 \cdot x + 4 \cdot x + x$. However, decompositions that reuse intermediate results or use a combination of additions and subtractions can result in fewer additions. For example, we could express $117x$ using multiple constraints with $63x = 64 \cdot x - x \wedge 59x = 63x - 4 \cdot x \wedge 117x = 2 \cdot 59x - x$. This only requires 3 additions/subtractions, compared to the 4 in the original example.

In hardware applications such as an Application-Specific Integrated Circuit (ASIC) or a Field-Programmable Gate Array (FPGA), it is important for the performance and cost of the

application to use small circuits, and therefore there is considerable previous research on finding a circuit for a given constant that minimizes the number k of additions and subtractions. This is known as the Single Constant Multiplication (SCM) problem. SCM is usually said to be NP-hard [17], however, the alternative double-base number system (DBNS) binary schemes admits a sublinear algorithm [114]. They do not address the claim of [17]. Furthermore, SCM has an upper bound of $k \leq \lceil \log(c)/2 \rceil$ [115], but again, an alternative binary scheme Canonical Signed Digit (CSD) can offer a tighter upper bound with $\lceil \log(c)/3 \rceil$. For many practical instances of the problem, however, k is known to be much smaller.

When encoding constant multiplication for SAT, one approach is to construct a SAT encoding directly from the truth table. Up to a limited number of bits, this can be achieved with a zero-variable overhead using Boolean logic minimization tools such as Espresso (`espresso`) [105]. However, the lack of auxiliary variables comes at the cost of an exponential number of literals for larger constants. Instead, we can adopt the SCM approach for SAT, where integer variables are also often represented in binary, and added or subtracted together by the traditional Ripple Carry Adder (RCA) [62] encoding.

Furthermore, the RCA encoding for a single addition or subtraction requires multiple so-called *half adders* and *full adders*, each adding two or three bits together, respectively. The standard encoding of the half adder to SAT requires 7 clauses (consisting of 19 literals), while a full adder requires 14 clauses (consisting of 68 literals) [64]. Thus, the number and type of adders serve as a strong approximation for encoding size and complexity. While the difference between full and half adders is significant, leveraging it is future work. In this chapter, we refer both full and half adders as simply adders.

The required number of adders is determined by the constant as well as the number of bits of the inputs. Recall that the number of bits $\mathcal{B}(x)$ to represent x is not fixed, but depends on the domain of x (e.g., only $\mathcal{B}(x) \equiv 2$ bits are required to represent $x \in [0..2]$ with Boolean variables $[\![\mathrm{bit}(x,0)]\!], [\![\mathrm{bit}(x,1)]\!]$). Consequently, the addition of $5x = 4 \cdot x + x$ requires *no* adders if x has two bits $[\![\mathrm{bit}(x,0)]\!], [\![\mathrm{bit}(x,1)]\!]$, since $5x$ could be represented using the existing bits $[\![\mathrm{bit}(x,0)]\!], [\![\mathrm{bit}(x,1)]\!], [\![\mathrm{bit}(x,0)]\!], [\![\mathrm{bit}(x,1)]\!]$. If x has three bits, multiple adders are required. Consequently, instead of minimizing the number of additions/subtractions k for a given constant c, a better circuit for SAT minimizes the total number of adders a for a given constant c and number of bits, $\mathcal{B}(x)$.

Example 4.1 Consider two circuits for $c \equiv 117$: $17x = 16 \cdot x + x; 19x = 2 \cdot x + 17x; 117x = 8 \cdot 17x (\equiv 136x) - 19x$ and $63x = 64 \cdot x - x; 59x = 63x - 4x; 117x = 2 \cdot 59x (\equiv 118x) - x$. Both are minimal with respect to $k \equiv 3$, but assuming $\mathcal{B}(x) \equiv 4$ bits for $x \in [0..7]$, the first requires $a \equiv 18$ adders, while the second requires $a \equiv 31$. Yet another circuit, $3x = 2 \cdot x + x; 49x = 16 \cdot 3x (\equiv 48x) + x; 17x = 16 \cdot x + x; 117x = 4 \cdot 17x (\equiv 68x) + 49x$, requires only $a \equiv 12$ adders, even though it uses $k \equiv 4$ additions/subtractions. For $\mathcal{B}(x) \equiv 4$, $a \equiv 12$ is optimal for $c \equiv 117$, but for different $\mathcal{B}(x)$ the best circuit changes.

Contributions

The contributions of this chapter are:

- A simple approach, `espresso`, for solving SCM using Boolean logic minimization for SAT with zero-variable overhead.
- A Constraint Programming (CP)/MiniZinc approach to solve classic SCM minimizing additions/subtractions, `min-k`, and an extended approach to solve SCM with an alternative objective minimizing literals by approximation of minimizing adders, `min-a`.[9]
- The construction of a database of optimal decompositions for a useful instance parameter space up to 16 bits, for constants $0 \leq c \leq 2047$.
- We evaluate the application of SCM to SAT to improve both encoding size and solve times by encoding linear constraints of the Multidimensional Bounded Knapsack Problem (MBKP) problem, which are parameterized by bit width (integer variable domain size). We compare all SCM approaches as well as a control SAT encoder, Picat-SAT (`picat-sat`) [90].

The SCM-SAT encoding database is available separately[10] and as part of the released implementation and benchmarks [117].

[9] After publication, we were notified by the first author, Rémi Garcia, of [116], a work published earlier in the same year. It considers the same `min-a` objective in the context of hardware design, which we will discuss in Sec. 4.2.2. We thank Rémi for his detailed remarks which improved the presentation and this chapter.

[10] https://github.com/hbierlee/scm-sat

4.2 Preliminaries and related work

In this section, we cover preliminaries specific to this chapter in Sec. 4.2.1 (mainly formalizing the SCM problem), and we discuss related work in Sec. 4.2.2.

4.2.1 The Single Constant Multiplication problem

The SCM problem can be formalized as follows:

Definition 4.1 Given the multiplier target c, we are to decide for a sequence of up to n equations of the form $c' \cdot x = 2^{s^\ell} \cdot \left(c^\ell \cdot x\right) \pm 2^{s^r} \cdot (c^r \cdot x)$, or more simply: $c' = 2^{s^\ell} \cdot c^\ell \pm 2^{s^r} \cdot c^r$. The left argument c^ℓ is either previously computed, or $c^\ell \equiv 1$, and $s^\ell \in \mathbb{Z}$ is the shift of the left argument. Similarly, c^r is the right argument, and $s^r \in \mathbb{Z}$ is its shift. Finally, we also denote whether the equation is an addition or a subtraction, $\pm \in \{+, -\}$. The final equation's c' should be equal to c.

Example 4.2 The first decomposition for 117 of Ex. 4.1 is:

$$\begin{aligned} 1 \cdot 2^2 \ &+ 1 \cdot 2^0 \ &= 17 \\ \wedge\, 1 \cdot 2^1 \ &+ 17 \cdot 2^0 &= 19 \\ \wedge\, 17 \cdot 2^3 \ &- 19 \cdot 2^0 &= 117 \end{aligned}$$

The second decomposition is defined by:

$$\begin{aligned} 1 \cdot 2^6 \ &- 1 \cdot 2^0 &= 63 \\ \wedge\, 63 \cdot 2^0 \ &- 1 \cdot 2^2 &= 59 \\ \wedge\, 59 \cdot 2^1 \ &- 1 \cdot 2^0 &= 117 \end{aligned}$$

Usually, the goal of the SCM problem is to find the minimal k decomposition for a given c (bounded by $k \leq \lceil \log_2(c)/2 \rceil$).

We can simplify the problem with the following two propositions [118]:

Proposition 4.1 If the target c is even, we can compute c from $c/2$.

By extension, if $c \mid 2^s$, we can compute c from $2^s \cdot c$.

Proposition 4.2 We only need to consider $s^\ell, s^r \geq 0$ for SCM.

4.2.2 Related work

The SCM problem has been studied extensively in the context of hardware. Many approaches for finding optimal circuits have been proposed, using graph based techniques [119], or using Integer Linear Programming (ILP) solvers [120], SAT solvers [121], [122], as well as heuristic approaches [122]. A generalization of SCM is the Multiple Constant Multiplication (MCM), which builds a circuit for multiple target constants. We briefly discuss MCM for SAT as future work in Sec. 4.5. Compared to SCM, this enables further sharing of intermediate results between the circuits for different constants. Interestingly, Prop. 4.2 does not hold for MCM.

Apart from minimizing the number of nodes (additions and subtractions), some of these related works consider other objectives such as minimizing the required surface area, delay, or power consumption of the generated circuits. These metrics are obviously useful when generating hardware circuits, but they are not directly relevant for SAT encodings.

Minimizing the number of adders (`min-a`) in the resulting circuit is a useful approximation in the context of SAT solvers, since it results in a smaller Conjunctive Normal Form (CNF) in terms of the number of variables, clauses, and literals, as we shall see in section Sec. 4.4. As mentioned in the introduction, `min-a` has been formulated in Mixed-Integer Linear Programming (MIP) for hardware design [116]. Furthermore, [118] discussed `min-a`, although they did not optimize for it. They also raised the point that in circuit design the shifts are not free either, but this does not apply to SAT.

4.3 Applying SCM to SAT

In this section, we present different approaches for solving the SCM problem for SAT as defined in Def. 4.1. In Sec. 4.3.1, we establish a popular baseline method. In Sec. 4.3.2, we propose how to encode SCM directly using Boolean minimization `Espresso` with zero-variable overhead. In Sec. 4.3.3, a Constrained Optimization Problem (COP) formulation of SCM is implemented in MiniZinc to find decompositions of minimal additions/subtractions. In Sec. 4.3.4, the decompositions of the SCM problem are encoded to SAT, showing how some additions/subtractions can take fewer literals. In Sec. 4.3.5, an alternative objective minimizes the number of one-bit adders as an approximation for literals.

4.3.1 A baseline algorithm for encoding $y = c \cdot x$

In order to establish a baseline, we will first discuss a well-known, simple solution to the SCM problem that does not optimize the resulting circuit at all.

Multiplication by a constant can always be represented using a combination of shifts and additions. Suppose x is defined as a $\mathcal{B}(x)$ width binary encoded integer. Then we know that $c \cdot x$ can be at most $c \cdot \left(2^{\mathcal{B}(x)} - 1\right)$, thus requiring $\mathcal{B}(c \cdot x) \equiv \left\lceil \log\left(c \cdot \left(2^{\mathcal{B}(x)} - 1\right)\right)\right\rceil$ bits to encode.

A basic shift-and-add algorithm will construct a solution to the above SCM problem by decomposing $c \cdot x$ by a shifted term for every 1 in c's binary representation:

$$c \cdot x = \sum_{0 \leq i < \mathcal{B}(c), \mathrm{bit}(c,i) \equiv 1} 2^i x$$

The number of additions required is equal to the number of 1s in c's binary representation minus one, or at most $\lceil \log_2(c) \rceil - 1$.

Example 4.3 A baseline decomposition for 117 is:

$$\begin{aligned} & 1 \cdot 2^6 && + 1 \cdot 2^5 = 96 \\ \wedge\ & 96 \cdot 2^0 && + 1 \cdot 2^4 = 112 \\ \wedge\ & 112 \cdot 2^0 && + 1 \cdot 2^2 = 116 \\ \wedge\ & 116 \cdot 2^0 && + 1 \cdot 2^0 = 117 \end{aligned}$$

4.3.2 Using Boolean circuit minimization to tackle SCM

Another option for encoding SCM into SAT is to use algorithms such as the Espresso (`espresso`) [105] logic minimizer that directly produce small logic circuits based on truth tables or similar representations of Boolean formulae. These tools are remarkably powerful, and for small enough problems can produce a guaranteed minimal size circuit for a formula *without introducing auxiliary Boolean variables for intermediate results.*

Example 4.4 For example, we can construct a CNF encoding for $y = 117 \cdot x$ with $x \in [0..15]$ by feeding the following truth table between the x and y bits into Espresso:

	x				y										
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	1	1	1	0	1	0	1
2	0	0	1	0	0	0	0	1	1	1	0	1	0	1	0
3	0	0	1	1	0	0	1	0	1	0	1	1	1	1	1
4	0	1	0	0	0	0	1	1	1	0	1	0	1	0	0
5	0	1	0	1	0	1	0	0	1	0	0	1	0	0	1
6	0	1	1	0	0	1	0	1	0	1	1	1	1	1	0
7	0	1	1	1	0	1	1	0	0	1	1	0	0	1	1
8	1	0	0	0	0	1	1	1	0	1	0	1	0	0	0
9	1	0	0	1	1	0	0	0	0	0	1	1	1	0	1
10	1	0	1	0	1	0	0	1	0	0	1	0	0	1	0
11	1	0	1	1	1	0	1	0	0	0	0	0	1	1	1
12	1	1	0	0	1	0	1	0	1	1	1	1	1	0	0
13	1	1	0	1	1	0	1	1	1	1	1	0	0	0	1
14	1	1	1	0	1	1	0	0	1	1	0	0	1	1	0
15	1	1	1	1	1	1	0	1	1	0	1	1	0	1	1

Tab. 4.1: Truth table for $y = 15 \cdot x$ where $x \in [0..3]$ and $y \in [0..15]$

The circuit minimization of this truth table results in a SAT encoding with 15 variables (simply representing the bits of x and y) and 84 clauses.

```
c b=4,c=117
p cnf 15 84
1 2 4 -8 0
1 2 4 -10 0
2 3 4 -12 0
1 3 4 -13 0
1 2 4 -14 0
1 2 -4 8 0
1 2 -4 10 0
1 3 -4 12 0
1 2 -4 13 0
1 2 3 -4 14 0
1 3 -7 0
1 2 3 -9 0
1 2 3 -11 0
1 2 3 -15 0
1 -3 4 12 0
4 -15 0
1 -3 -4 -12 0
-3 -4 15 0
1 -3 7 0
1 2 -3 9 0
2 -3 11 0
1 2 -3 13 0
1 2 -3 -14 0
2 -6 0
1 -2 4 8 0
1 3 4 -9 0
1 -2 4 10 0
-2 3 4 11 0
1 -2 -4 -8 0
1 -2 3 -4 9 0
-2 3 -4 -10 0
3 -4 -11 0
-2 -4 15 0
1 -2 3 12 0
-2 3 -14 0
1 -3 4 9 0
-2 -3 4 -11 0
1 -2 4 12 0
1 -2 -3 -4 -9 0
-3 -4 11 0
1 -2 -3 10 0
-2 -3 14 0
-2 6 0
1 -2 -13 0
1 -5 0
2 3 4 -8 0
-1 2 3 4 10 0
-1 2 4 11 0
2 3 -4 8 0
-1 3 -4 -10 0
-1 -4 15 0
-1 3 7 0
-1 2 3 9 0
-1 3 -12 0
-1 2 3 -13 0
-1 3 -14 0
-1 2 -3 4 8 0
-1 2 -3 4 -9 0
2 -3 4 -10 0
-1 4 -12 0
-1 2 4 -13 0
-1 -3 4 14 0
-1 2 -3 -4 -8 0
-1 2 -4 9 0
2 -3 -4 10 0
-1 -3 -4 12 0
2 -3 -4 13 0
-1 2 -4 -14 0
-1 -3 -7 0
-2 3 4 8 0
-1 -2 4 9 0
-1 3 4 11 0
-2 3 -4 -8 0
-1 -2 3 -4 -9 0
-1 -2 3 -10 0
-1 -2 3 13 0
-1 -2 -3 4 -8 0
-2 -3 4 10 0
-1 -2 4 13 0
-1 -2 -3 -4 8 0
-1 -2 -4 -10 0
-2 -3 -4 -13 0
-1 -2 -3 9 0
-1 5 0
```

Lst. 4.1: The DIMACS file produced by `espresso` given truth table Tab. 4.1, in which variables $1, ..., 4$ correspond to $[\![\text{bit}(x, 0)]\!], ..., [\![\text{bit}(x, 3)]\!]$ and $5, ..., 15$ correspond to $[\![\text{bit}(y, 0)]\!], ..., [\![\text{bit}(y, 10)]\!]$ (`./4_117.dimacs`)

Unfortunately as the bit width $\mathcal{B}(x)$ of the input variable x and the size of the constant c grows, the number of Boolean variables in the formula $y = c \cdot x$ quickly becomes too large for logic minimization. However, for small bit widths $\mathcal{B}(x)$ and constants c, logic minimization can generate the best SAT encoding for $y = c \cdot x$, as demonstrated by our experiments in Sec. 4.4.

4.3.3 Formulating SCM as a COP

We will now introduce a general method for solving the SCM problem, by formulating it as a COP. First, we can reduce the possible forms of the equation to just three. This proof overlaps with Theorem 3 of [118], but also splits off the two subtraction cases since it is important when calculating the number of adders in Sec. 4.3.5.

Proposition 4.3 Given a k equation decomposition to compute c using the general form in Def. 4.1, then there is a k equation decomposition using equations of the form:

$$\texttt{SPLUS}(c_a, s, c_b) \equiv 2^s c_a + c_b \tag{4.1.1}$$

$$\texttt{SMINUS}(c_a, s, c_b) \equiv 2^s c_a - c_b \tag{4.1.2}$$

$$\texttt{MINUSS}(c_a, s, c_b) \equiv c_a - 2^s c_b \tag{4.1.3}$$

Proof. Starting from $c' = 2^{s^\ell} \cdot c^\ell \pm 2^{s^r} \cdot c^r$ from Def. 4.1, we will show that in all cases the equation corresponds to an equation from Eq. 4.1. First, we have two cases depending on which shift is greater, s^ℓ or s^r:

$$c' = \begin{cases} 2^{s^r} \cdot \left(2^{s^\ell - s^r} \cdot c^\ell \pm c^r\right) \text{ if } s^\ell \geq s^r \\ 2^{s^\ell} \cdot \left(c^\ell \pm 2^{s^r - s^\ell} \cdot c^r\right) \text{ otherwise } (s^\ell < s^r) \end{cases}$$

In the first case we have $c' \mid 2^{s^r}$, so we can compute c' from $2^{s^\ell - s^r} \cdot c^\ell \pm c^r$ by Prop. 4.1. The same holds for the other case. After splitting $\pm$ into cases for $+$ and $-$, we have the corresponding equations:

$$c' = \begin{cases} 2^{s^\ell - s^r} \cdot c^\ell \pm c^r \equiv \begin{cases} 2^{s^\ell - s^r} \cdot c^\ell + c^r \equiv \texttt{SPLUS}(c^\ell, s^\ell - s^r, c^r) \\ 2^{s^\ell - s^r} \cdot c^\ell - c^r \equiv \texttt{SMINUS}(c^\ell, s^\ell - s^r, c^r) \end{cases} \\ c^\ell \pm 2^{s^r - s^\ell} \cdot c^r \equiv \begin{cases} c^\ell + 2^{s^r - s^\ell} \cdot c^r \equiv \texttt{SPLUS}(c^r, s^r - s^\ell, c^\ell) \\ c^\ell - 2^{s^r - s^\ell} \cdot c^r \equiv \texttt{MINUSS}(c^\ell, s^r - s^\ell, c^r) \end{cases} \end{cases}$$

□

Example 4.5 Both decompositions for 117 of Ex. 4.2 can be represented equivalently as:

$$17 = \texttt{SPLUS}(1, 4, 1) \wedge 19 = \texttt{SPLUS}(1, 1, 17) \wedge 117 = \texttt{SMINUS}(1, 3, 19)$$

The second decomposition is defined by:

$$63 = \texttt{SMINUS}(1, 6, 1) \wedge 59 = \texttt{MINUSS}(63, 2, 1) \wedge 117 = \texttt{SMINUS}(59, 1, 1)$$

A MiniZinc [3] model for solving SCM as a combinatorial optimization problem is shown in Lst. 4.2. Note that it only makes use of the three decompositions from Prop. 4.3. Each equation is represented by its type `ty`, with `NOP` added for an unused equation; the equation numbers defining its left (`left`) and right (`right`) arguments, and the shift amount `shift`.[11] We use

[11] The careful reader will note that we limit the maximum shift using the target. Relaxing this restriction and never found better decompositions (in terms of equations or adders), but we have no formal proof for it.

equation number 0 to represent $1 \cdot x$. The multiplier resulting from an equation is defined by `mult`. Line 14 sets the dummy equation multiplier to 1, and enforces the final result to be the target. In lines 16-19 we enforce that all equations after `used` are NOPs, and set their other components to dummy values. Line 20 enforces that equations use only earlier defined multiples. The computation of the multiplier for each equation (lines 21-28) follows the definition, where `NOP` just returns the previous multiplier. We minimize the number of used equations. Note that the model "pre-computes" the powers of 2 in `p2` (line 13), since solvers typically propagate poorly for exponential expressions.

```
1  int: xbits; % number of bits to represent x
2  int: n;     % max number of +- equations
3  set of int: EQ = 1..n;
4  set of int: EQ0 = 0..n; % equation number 0 = 1x
5  int: maxsh = ceil(log2(target)); % maximum left-shift
6  int: target;  % target multiplier
7  enum TYPE = { SPLUS, SMINUS, MINUSS, NOP };
8  array[EQ] of var TYPE: ty;   % type of equation
9  array[EQ] of var EQ0: left;  % left input equation numnber
10 array[EQ] of var EQ0: right; % right input equation number
11 array[EQ] of var 0..maxsh: shift; % left-shift left applied
12 array[EQ0] of var 0..infinity: mult; % multiplier value
13 array[1..maxsh+xbits] of int: p2 = [2^i | i in 1..maxsh+xbits];
14 constraint mult[0] = 1 /\ mult[n] = target;
15 var EQ0: used; % number of equations used
16 constraint forall(e in EQ)(e > used <-> ty[e] = NOP);
17 constraint forall(e in EQ)(
18   e > used -> left[e] = 0 /\ right[e] = 0 /\ shift[e] = 1
19 );
20 constraint forall(e in EQ)(left[e] < e /\ right[e] < e);
21 constraint forall(e in EQ)(mult[e] =
22   if ty[e] = SPLUS then
23      p2[shift[e]] * mult[left[e]] + mult[right[e]]
24   elseif ty[e] = SMINUS then
25      p2[shift[e]] * mult[left[e]] - mult[right[e]]
26   elseif ty[e] = MINUSS then
27      mult[left[e]] - p2[shift[e]] * mult[right[e]]
28   else mult[e-1] endif);
29 solve minimize used; % minimize equations
```

Lst. 4.2: A MiniZinc model `scm.mzn` for the SCM problem

4.3.4 Encoding an SCM decomposition

Given a solution (e.g., Ex. 4.2) to the SCM problem, we can encode the shifts and adds of each equation $(c' \cdot x) = 2^{s^\ell} \cdot \left(c^\ell \cdot x\right) \pm 2^{s^r} \cdot (c^r \cdot x)$. We obtain the binary encoding of the output $z = (c' \cdot x)$ from the already computed binary encodings of the inputs $z^\ell = \left(c^\ell \cdot x\right)$ and $z^r = (c^r \cdot x)$.

As mentioned in Sec. 4.1, encoding the affine transformation on the binary encoding of z^ℓ, or $E\left(y^\ell = 2^s z^\ell{:}\mathbb{B}\right)$, is a total view: $[\![\mathrm{bit}\left(y^\ell, i\right)]\!] \equiv 0, 0 \leq i < s$ and $[\![\mathrm{bit}\left(y^\ell, i\right)]\!] \equiv [\![\mathrm{bit}\left(z^\ell, i - s\right)]\!], s \leq i \leq \mathcal{B}\left(z^\ell\right)$, using no additional variables or clauses. We define y^r similarly.

After shifting both inputs, it remains to encode an addition of the form $z = y^\ell \pm y^r$. For addition, $z = y^\ell + y^r$, we can encode an RCA which produces the output (sum) bits $[\![\mathrm{bit}(z, i)]\!]$, and auxiliary carry bits $[\![\mathrm{bit}(c, i)]\!]$. Inputs y^ℓ, y^r might contain fixed literals due to the applied shifts, which in turn leads to fixed outputs, or outputs which are equivalent to inputs, or their negations. The initial carry bit is also fixed as $[\![\mathrm{bit}(w, 0)]\!] \equiv 0$ (which is essentially why a half adder is used to add the first bits). We optimize the RCA encoding to account for this.

The current sum bit $[\![\mathrm{bit}(z, i)]\!]$ is equivalent to the result of a *parity* constraint: a $\oplus$ on a list of up to three input variables, $[\![\mathrm{bit}\left(y^\ell, i\right)]\!] \oplus [\![\mathrm{bit}(y^r, i)]\!] \oplus [\![\mathrm{bit}(w, i)]\!]$, which we reformulate more generically as $\oplus\left([l_1, ..., l_r]\right), r \leq 3$. We remove any fixed variables from the input list, and compute the sum of their values. Then, we apply the $\oplus$ on the remaining non-fixed variables, negating the output if the fixed sum is odd. Given a list L of input bits, some of which are fixed, define $f(L)$ as the sum of the fixed bits in L and $b(L)$ the list of unfixed (variable) bits in L. We can encode $[\![\mathrm{bit}(z, i)]\!] = [\![\mathrm{bit}\left(y^\ell, i\right)]\!] \oplus [\![\mathrm{bit}(y^r, i)]\!] \oplus [\![\mathrm{bit}(w, i)]\!]$ as follows:

$$E(\oplus\,(L)) \equiv \begin{cases} f(L) \bmod 2 & \text{if } b(L) \equiv [] \\ l_1 & \text{if } b(L) \equiv [l_1] \wedge f(L) \equiv 0 \\ \neg l_1 & \text{if } b(L) \equiv [l_1] \wedge f(L) \equiv 1 \\ l_1 \oplus l_2 & \text{if } b(L) \equiv [l_1, l_2] \wedge f(L) \equiv 0 \\ l_1 \oplus l_2 \oplus 1 & \text{if } b(L) \equiv [l_1, l_2] \wedge f(L) \equiv 1 \\ l_1 \oplus l_2 \oplus l_3 & \text{otherwise } (b(L) \equiv [l_1, l_2, l_3]) \end{cases}$$

$$\equiv E(\oplus\,(b(L)) \oplus (f(L) \bmod 2))$$

If $\oplus\,(L)$ is constant or a literal l, the constraint is a partial view since we equate $[\![\mathrm{bit}(z, i)]\!] \equiv l$ without overhead. Otherwise, $[\![\mathrm{bit}(z, i)]\!]$ is equated to a $\oplus$ expression, which can be encoded using a single XOR clause if supported by the SAT solver [123], or encoded with (standard) clauses in a well understood manner.

The next carry bit is set to true if at least 2 of its input variables are. In other words, $[\![\mathrm{bit}(w, i + 1)]\!]$ is true if and only if the cardinality constraint $[\![\mathrm{bit}\left(y^\ell, i\right)]\!] + [\![\mathrm{bit}(y^r, i)]\!] + [\![\mathrm{bit}(w, i)]\!] \geq 2$ holds. Again, we split off and sum the fixed variables, and compute the carry by evaluating the generic version, $l_1 + \cdots + l_r \geq 2, r \leq 3$, written $\sum(L) \geq 2$, of the at-least-2 constraint as follows:

$$E\Big(\sum(L) \geq 2\Big) \equiv \begin{cases} 1 & \text{if } f(L) \geq 2 \\ 0 & \text{if } b(L) \equiv [] \wedge f(L) < 2 \\ 0 & \text{if } b(L) \equiv [l_1] \wedge f(L) \equiv 0 \\ l_1 & \text{if } b(L) \equiv [l_1] \wedge f(L) \equiv 1 \\ l_1 \wedge l_2 & \text{if } b(L) \equiv [l_1, l_2] \wedge f(L) \equiv 0 \\ l_1 \vee l_2 & \text{if } b(L) \equiv [l_1, l_2] \wedge f(L) \equiv 1 \\ (l_1 \wedge l_2) \vee (l_1 \wedge l_3) \vee (l_2 \wedge l_3) & \text{otherwise } (b(L) \equiv [l_1, l_2, l_3]) \end{cases}$$

$$\equiv E\Big(\sum(b(L)) \geq (2 - f(L))\Big)$$

As with the parity constraint, if the result of the sum constraint is a constant or a literal, we can just equate $[\![\mathrm{bit}(w, i+1)]\!]$ with the result without overhead. Otherwise, $[\![\mathrm{bit}(w, i+1)]\!]$ is equated to the CNF expression as normal.

To handle subtraction, we use the fact that RCAs directly handle two's complement. So, $y^\ell + (-y^r) \equiv y^\ell + (\overline{y^r} + 1)$ where the encoding of $\overline{y^r}$ is the complement of y^r, i.e. $[\![\mathrm{bit}(\overline{y^r}, i)]\!] = \neg[\![\mathrm{bit}(y^r, i)]\!]$. To offset the constant 1, we can add an initial carry of 1.

4.3.5 Minimizing the number of adders

The COP formulation of Sec. 4.3.3 minimizes the number of equations, k. However, we show in Sec. 4.3.4 how the true encoding complexity for each equation is more closely related to the number of adders. In this section, we formulate an alternative objective which minimizes the number of adders a instead.

We are now required to consider the bit width $\mathcal{B}(x)$ of the input x being multiplied as an additional parameter. Consider the addition $c' \cdot x = 2^{s^\ell} \cdot \big(c^\ell \cdot x\big) + 2^{s^r} \cdot (c^r \cdot x)$. Assuming the left input $c^\ell \cdot x$ is represented in $\mathcal{B}\big(c^\ell \cdot x\big)$ bits and the right input $c^r \cdot x$ is represented in $\mathcal{B}(c^r \cdot x)$ bits, then the number of adders naïvely required for the addition is the width of the result $\mathcal{B}(c' \cdot x)$.

However, we can omit adders for low bits where one argument is guaranteed 0, and if the non-zero bits never overlap we need no adders. So the number of adders required is 0 if $s^\ell > \mathcal{B}(c^r \cdot x) + s^r$ or if $s^r > \mathcal{B}\big(c^\ell \cdot x\big) + s^\ell$, otherwise the number of adders required is $\max\big(\mathcal{B}\big(c^\ell \cdot x\big) + s^\ell, \mathcal{B}(c^r \cdot x) + s^r\big) - \max\big(s^\ell, s^r\big)$. For subtractions, the only bits not requiring adders are the low bits where both operands are known to be 0, so we require $\mathcal{B}(c' \cdot x) - \min\big(s^\ell, s^r\big)$ adders.

Again, transforming the general form to the simplified equations of Eq. 4.1 does not change the required number of adders required. This is because a right-shifting both inputs by the same amount only removes the low bits which did not require any adders. Furthermore, applying Prop. 4.1 requires no additions, and so no adders.

We can modify our MiniZinc model (see Lst. 4.2) to keep track of the number of bits required for the result of each equation, and count the number of adders used for each equation, and make that the new objective. The additions to the model (and replacement objective) are shown in Lst. 4.3. The number of bits required for the result of each equation are defined by `bits` while the number of adders for each equation are given by `adders`. In lines 3-4 we compute the bit width directly from the multiple (the least power big enough to hold the maximum result). We compute the number of adders required for each decomposition in lines 6-12. Finally, we minimize the sum of adders required in line 13.

```
1  array[EQ] of var 0..infinity: bits; % number of bits
2  array[EQ] of var 0..infinity: adders; % number of adders
3  include "arg_max.mzn";
4  constraint forall(e in EQ)(bits[e] =
5    arg_max([mult[e]*(p2[xbits]-1)<=p2[i]|i in 1..maxsh+xbits ]));
6  constraint forall(e in EQ)(adders[e] =
7    if ty[e] = SPLUS then
8       if shift[e] >= bits[right[e]] then 0
9       else max(bits[left[e]],bits[right[e]]-shift[e]) endif
10   elseif ty[e] = SMINUS then bits[e]
11   elseif ty[e] = MINUSS then bits[e]
12   else 0 endif);
13 solve minimize sum(adders);  % minimize adders
```

Lst. 4.3: An alternative objective for `scm.mzn` (see Lst. 4.2)

Note that for minimizing the SAT model we only minimize an approximation for the size of the resulting SAT model, the number of adders. We experimented with directly minimizing the number of clauses or literals (sum of size of clauses) in the SAT encoding. The resulting models were much harder to solve, let alone optimally. For the small examples where we can get a result, it is rarely better than the SAT encoding resulting from minimizing adders. It remains interesting future work to see if this approach to encoding minimization could be improved.

4.4 Experimental evaluation

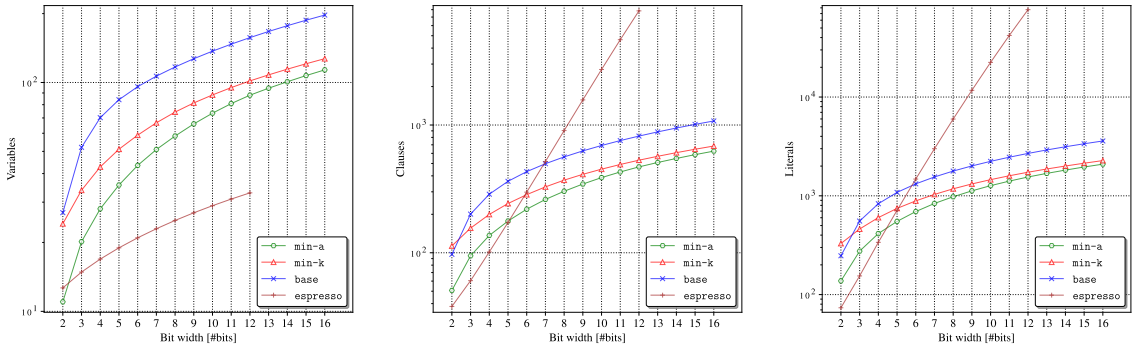
To evaluate the effectiveness of the proposed approach, we compare the size of the encoding of $c \cdot x$ for a range of constants c and bit widths for x in Sec. 4.4.1. In Sec. 4.4.2 we apply our approach to a realistic benchmark, comparing both the encoding size and solve times.

4.4.1 Constructing and analysing the SCM databases

In this section, we compare the different SCM approaches in terms of the size of their respective encodings. For each bit width $2 \leq \mathcal{B}(x) \leq 16$, we average the number of variables, clauses, and literals that are generated when encoding $c \cdot x$ over $1 \leq c \leq 2047$. The four methods are `base`

(the baseline from Sec. 4.3.1), `espresso` (minimizing variables from Sec. 4.3.2, using `espresso` version 2.3), `min-k` (minimizing the number of RCAs) and `min-a` (minimizing the total number of adders). It becomes increasingly hard for Espresso to compute CNF for all constants as the number of input bits grows, so for `espresso` we are limited to $\mathcal{B}(x) \leq 12$. The results are shown in Fig. 4.1.

It is clear from Fig. 4.1.1 that the naive decomposition of `base` produces by far the most variables. In contrast, `min-k` achieves the same multiplication using fewer RCAs. Importantly, `min-a` has even fewer variables than `min-k`, indicating that minimizing the number of adders serves as an approximation for avoiding additional variables. The effect is stronger for smaller bit widths, presumably because then there is more opportunity for completely free additions where one input is shifted by the other input's bit width. The number of variables for `espresso` is always minimal, since no additional variables are ever constructed. In terms of clauses and literals, shown respectively in Fig. 4.1.2 and Fig. 4.1.3, we see a linear relationship between the other methods other than `espresso`, which generates an exponential number of clauses and literals. Over all dimensions, `min-a` is better than `min-k`, which is better than `base`.



4.1.1 Variables 4.1.2 Clauses 4.1.3 Literals

Fig. 4.1: Encoding size of $c \cdot x$ using different SCM approaches

4.4.2 Solving Multidimensional Bounded Knapsack Problem

To evaluate the different approaches, we adapt existing techniques which generate Multidimensional Multiple Choice Knapsack Problems [124]. As these problems are Pseudo-Boolean (PB) (where all domains are [0..1]), we generalize this technique in a straightforward manner to instead generate Multidimensional Bounded Knapsack Problem (MBKP) instances. In these integer linear problems, all domains are $[0..B]$, which allows us to vary the number of bits required to encode the decision variables.

In MBKP, there are N item types having weight in M dimensions given by $w_{i,j}$ for dimension $1 \leq i \leq M$, item $1 \leq j \leq N$. The j-th item also has a profit p_j. We decide how many (between 0 and B) of item type j to pack by decision integer variable x_j. The sum of weights for the i -th dimension cannot exceed the maximimum weight W_i, and the sum of profits must exceed

the minimum profit, P. This yields the following integer linear decision problem, with binary encoded integer variables, requiring $N \cdot (M+1)$ SCM constraints:

$$\bigwedge_{i \equiv 1}^{M} \sum_{j \equiv 1}^{N} w_{i,j} \cdot \left(x_j{:}\mathbb{B} \in [0..B]\right) \leq W_i$$

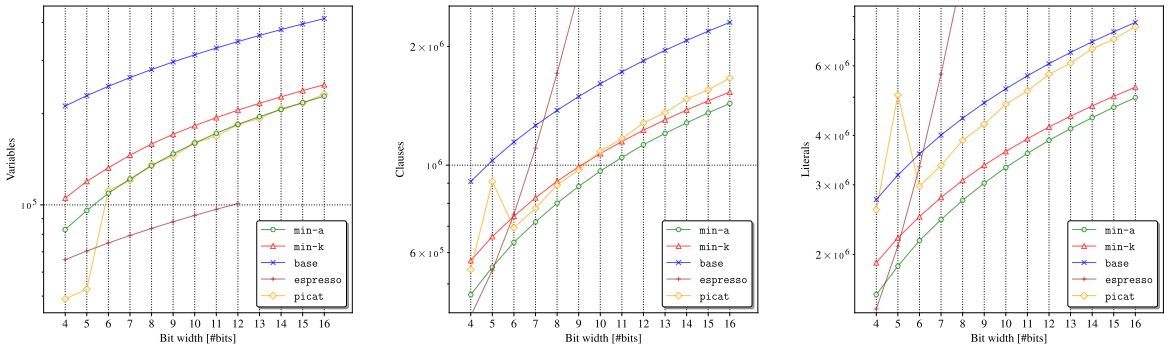
$$\sum_{j \equiv 1}^{N} \left(p_j \cdot x_j{:}\mathbb{B}\right) \geq P$$

Instances can be generated by first generating C coefficient sets for $w_{i,j}$ and p_i, sampling uniformly from $[1..Q]$. Then, for each coefficient set, we choose S capacity sets for $1 \leq s \leq S$ with a capacity factor $f \equiv \frac{s}{S}$: generating $W_i \equiv f \sum_{j \equiv 1}^{N} B \cdot w_{i,j}, 1 \leq i \leq M$, and $P \equiv (1-f) \sum_{j \equiv 1}^{N} B \cdot p_j$. As f increases, the constraints become less strict, and instances turn from unsatisfiable to satisfiable. To avoid an excess of trivial instances on the lower and higher ends of f, we limit f to be within $0.2 \leq f \leq 0.8$. One instance set is generated for each $B \equiv 2^{\mathcal{B}(x)} - 1, 4 \leq \mathcal{B}(x) \leq 16$, with parameters $N \equiv 15$, $M \equiv 100$, $S \equiv 100$, $C \equiv 3$.

The `picat-sat` encodes SCM similar to `base`, but with additional ad-hoc optimizations [112]. Note that for these problems with large coefficients and domain sizes alternate encoding approaches, e.g., domain or order encodings are non-competitive.

All encodings produced in our experiments are solved using the same binary of the SAT solver `CaDiCal` (version 1.9.1) [125] with default parameters, 8 GB memory limit and a time limit of 180 seconds. A PAR2 penalty applies to timeouts.

In Fig. 4.2, we compare the encoding sizes of the MBKP. As we would expect, results are similar to the results from Sec. 4.4.1. The now added `picat-sat` encoding matches `min-a` in the number of variables, as shown in Fig. 4.2.1. However, for clauses, and especially for literals shown in Fig. 4.2.2 and Fig. 4.2.3, larger bit widths result in larger encodings, closer to `base`. For unknown reasons, `picat-sat` shows a temporary spike in clauses and literals at $\mathcal{B}(x) \equiv 5$ (for some constraints, `picat-sat` also relies on Espresso; which it perhaps uses in this case).



4.2.1 Variables 4.2.2 Clauses 4.2.3 Literals

Fig. 4.2: Encoding size of MBKP using different SCM approaches

Comparing solve times in Fig. 4.3, we observe that `espresso` performs best up to a bit width of 10, after which its performance drops off drastically. Even if pre-computing larger input bit widths would be tractable for `espresso`, we can extrapolate that an encoding without auxiliary variables loses its effectiveness. In contrast, `min-k` and `min-a` remain relatively constant, even though the instances grow exponentially in size. Furthermore, `min-a` performs better than `min-k`, which shows the benefit of minimizing adders over RCA, even if some decompositions have suboptimal length (such as in Ex. 4.1). Finally, `picat-sat` performs much better than `base` for smaller bit widths but similar for larger bit widths, suggesting that `picat-sat` is able to apply certain encoding optimizations in the former case.

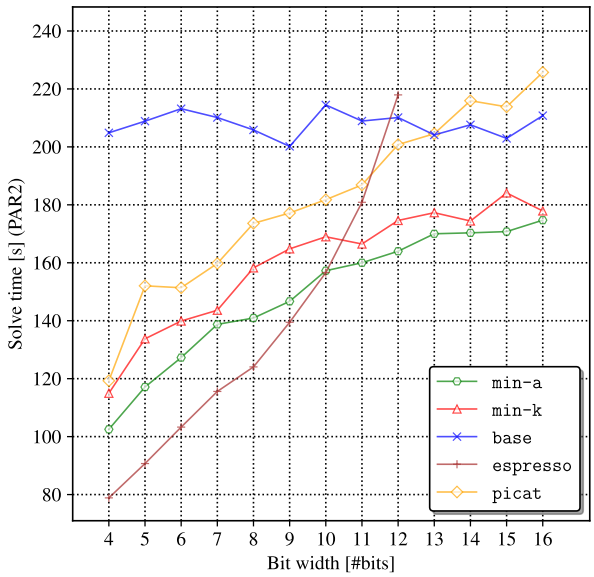


Fig. 4.3: Solve time comparison for MBKP

4.5 Conclusion and Future Work

In conclusion, we have shown how to tackle and apply SCM to SAT. To our knowledge, this is the first work that adapts SCM circuits to encode linear constraints for SAT. Since RCAs are not uniformly encoded in the presence of partially fixed inputs, this gives us the opportunity to develop different optimal circuits compared to the traditional SCM problem. In the experimental evaluation, we have seen how this approach significantly improves both the encoding size of and the solver's performance on linear constraints, which are a key challenge for SAT-based solvers [64]. Finally, we have made the pre-computed SCM encodings freely available.

In future work, we aim to extend our approach by applying MCM to SAT, which - unlike SCM - cannot be comprehensively pre-computed. Instead, we would first have to collect the target constants (e.g., through Common Subexpression Elimination (CSE) [103]) from a given instance and then dynamically solve the MCM problem at encode time. This poses an interesting trade-off between encode and solve time, but has the potential for additional improvements.

Another direction for future work is to improve the ad-hoc baseline method. The advantage of the baseline method from Sec. 4.3.1 is that it can apply to integers with large domains that are not included in the current database. However, its performance is by far the worst. A sensible approach judging by the resuls in Sec. 4.4.2 is to use `espresso` for domains requiring up to 10 bits, `min-a` up to 16, and an improved `base` algorithm for larger domains.

— Chapter 5 —

Revisiting Pseudo-Boolean Encodings from an Integer Perspective

The Pseudo-Boolean (PB) constraint is important and ubiquitous. Consequently, there various Boolean Satisfiability (SAT) encodings, including: the Generalized Totalizer (GT) [59], Sequential Weight Counter (SWC) [61] and Binary Decision Diagram (BDD) [58] encoding. The literature specifies each PB encoding with an initial abstraction (binary trees, weight counters and decision diagrams, respectively). Then, this abstraction is encoded to SAT.

In this chapter, we show how we can unify the various abstractions behind each PB-SAT encoding as distinct Constraint Satisfaction Problem (CSP) *decompositions* (i.e. a non-SAT encoding, see Def. 2.11). By encoding each decomposition according to the same uniform order encoding method, we recover effectively the same set of clauses as the original encoding. In other words, the uniform order encoding of the GT, SWC or BDD decomposition is equivalent to the (unit propagated) GT, SWC or BDD encoding on the same PB-constraint, respectively. We can take advantage of the decompose-and-encode method by exploring various extensions, which both theoretically and practically improve the respective base methods.

This chapter is adapted from the accepted manuscript for CPAIOR'25:

> H. Bierlee, Jip. J. Dekker, and P. J. Stuckey, "Revisiting Pseudo-Boolean Encodings from an Integer Perspective" in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 22nd International Conference, CPAIOR 2025, Melbourne, Victoria, Australia, November 10-13, 2025*

This chapter is organized as follows. In Sec. 5.1, we introduce the problem and our approach. In Sec. 5.2, we cover the main theoretical contribution: a decomposition framework, the decomposition of three PB encodings, and extensions made possible by the method. In Sec. 5.3, we experimentally evaluate the practical extensions. In Sec. 5.4, we conclude our findings and discuss future work.

5.1 Introduction

The most common and important constraint to encode is the linear constraint over Boolean literals, or PB constraints (first discussed in Sec. 2.3.5). A normalized form of a PB constraint can be derived where $\sum_{i\equiv 1}^{n} q_i b_i \leq k$, where q_i and k are positive integer constants and b_i are Boolean literals [58]. We will consider only the normalized form until Sec. 5.2.3. Apart from the aforementioned GT, SWC and BDD encodings, we can also make use of standard circuit encodings of addition, the Ripple Carry Adder (RCA) [62], to encode PB constraints.

Consider the following PB constraint: $2b_1 \in D^{\mathbb{B}}(b_1) + 3b_2 \in D^{\mathbb{B}}(b_2) + 5b_3 \in D^{\mathbb{B}}(b_3) \leq 6$. We can represent it by an equivalent (0,1)-ILP constraint: $2x_1 \in \{0,1\} + 3x_2 \in \{0,1\} + 5x_3 \in \{0,1\} \leq 6$. We can correctly encode this constraint with three distinct CSP decompositions, labeled by their associated PB encoding method:

$$\begin{cases} 2x_1 + 3x_2 \leq y_1 \in \{0,2,3,5\} \wedge y_1 + 5x_3 \leq 6 & \text{GT decomposition} \\ 2x_1 + y_1 \in [-6..0] \leq 0 \quad \wedge 3x_2 + y_2 \in [-6..0] \leq y_1 \wedge 5x_3 - 6 \leq y_2 & \text{SWC decomposition} \\ 2x_1 \leq y_1 \in \{1,2\} \quad \wedge 3x_2 + y_1 \leq y_2 \in \{1,5\} \wedge 5x_3 + y_2 \leq 6 & \text{BDD decomposition} \end{cases}$$

Our central observation is that an uniform order encoding of the each decomposition (e.g. $E((2x_1 + 3x_2 \leq y_1 \in \{0,2,3,5\} \wedge y_1 + 5x_3 \leq 6):\mathbb{O})$) effectively yields the same set of clauses as their original PB-SAT encoding for any PB constraint.

The underlying explanation of this shared behaviour is that each PB encoding implicitly generates encodings of some partial sums $\sum_{i\in J} q_i b_i$ where $J \subseteq \{1,...,n\}$. For example, SWCs and BDDs encode a linear decomposition of the sum: $y_j \geq \sum_{i\equiv 1}^{j} q_i b_i, 2 \leq j \leq n$, using encodings of $q_1 b_1 + q_2 b_2 \leq y_2$, and $y_j + q_{j+1} b_{j+1} \leq y_{j+1}, 2 \leq j < n$, finally adding $y_n \leq k$. Note that it is sufficient to enforce that y_j is greater than or equal to the partial sum, rather than equal to, since the overall inequality constraint is enforced by $y_n \leq k$. A tree decomposition of the sum, used by GTs, encodes the partial sums (assuming $n \equiv 2^k$ for simplicity) as $y_i^1 \geq q_{2i-1} b_{2i-1} + q_{2i} b_{2i}, 1 \leq i \leq 2^{k-1}$ and $y_i^{j+1} \geq y_{2i-1}^j + y_{2i}^j, 1 \leq i \leq 2^{k-j}$, finally adding $y^k \leq k$. The RCA encodings are free to decompose the sum in either way, and use binary encodings of partial sums.

In the original GT, SWC and BDD specifications, the auxiliary Boolean variables that implicitly represent the y variables are not explicitly treated as an integer variable. In the theoretical contribution of this chapter, we make this explicit by introducing a general CSP decomposition method for PB constraints. We show that a straightforward SAT encoding of specific decompositions effectively yields the same clauses as the original encoding. Then we propose the various extensions which theoretically and practically improve the base methods.

Contributions

The contributions of this chapter are:

- An integer-based decomposition method for encoding PB constraint to SAT.
- The equivalence of the encoding of particular decompositions to the GT, SWC and BDD encodings.
- Five extensions which take advantage of the integer viewpoints to theoretically improve our understanding of the base methods, as well as practically in an experimental evaluation. These are:
  - A simplified proof of propagation consistency for the general PB decomposition, which then trivially extends to a proof for the same property for the GT, SWC and BDD encoding.
  - A generalization from encoding PB constraints to Integer Linear (IL) constraints.
  - Support for At-Most-One (`AMO`) and Implication Chain (`IC`) side-constraints, i.e. converting a set of Boolean literals into an additional integer variable if the literals participate in pre-existing constraints.
  - Support for equality PB constraints.
  - Support for the binary encoding of specific integer variables.
- A experimental evaluation of our integer-based implementation of the GT and BDD encoding, the improvement of eaach practical extension, as compared to three baseline encoders.

5.2 PB encodings as CSP decompositions

This section covers the CSP decomposition method in three sub-sections: First, we formalize a general decomposition in Sec. 5.2.1. Then, we show how the order encoding of particular decompositions is effectively equivalent to the GT, SWC or BDD encodings in Sec. 5.2.2. Finally, we take advantage of the decomposed methods by proposing six extensions of the decompositions.

5.2.1 A CSP decomposition method for PB constraints

We now formalize the CSP decomposition method introduced in Sec. 5.1. First, we expand our high-level view of encodings and decompositions from Sec. 2.3:

SAT encoding An encoding (see Def. 2.11) from CSP to SAT.

CSP decomposition An encoding to a CSP (but not necessarily SAT).

PB encoding An encoding from PB to SAT.

GT encoding An encoding from PB to SAT using the GT encoding method.

PB decomposition A CSP decomposition of a PB constraint.

GT decomposition A PB decomposition, which, if encoded to SAT by the uniform order encoding, yields the same set of clauses as the unit propagated GT encoding does for the same PB constraint.

All encodings for the PB constraint $\sum_{i\equiv 1}^{n} q_i(b_i \in D^{\mathbb{B}}(b_i)) \leq k$ decompose the constraint into integer constraints, which implicitly or explicitly constrain integer partial sums using the order encoding. The first step is to decompose the input PB constraint as an integer constraint by introducing integer variables x_i to encode PB term $q_i \cdot (b_i \in D^{\mathbb{B}}(b_i))$ as $x_i \in \{0, q_i\}$. Then, we can encode the PB constraint as a sum of integer variables, $\sum_{i\equiv 1}^{n} x_i \in \{0, q_i\} \leq k$. The interpretation function of this encoding is self-evident.

Example 5.1 Consider the following PB constraint:

$$2b_1 \in D^{\mathbb{B}}(b_1) + 3b_2 \in D^{\mathbb{B}}(b_2) + 5b_3 \in D^{\mathbb{B}}(b_3) \leq 6$$

Its CSP decomposition is:

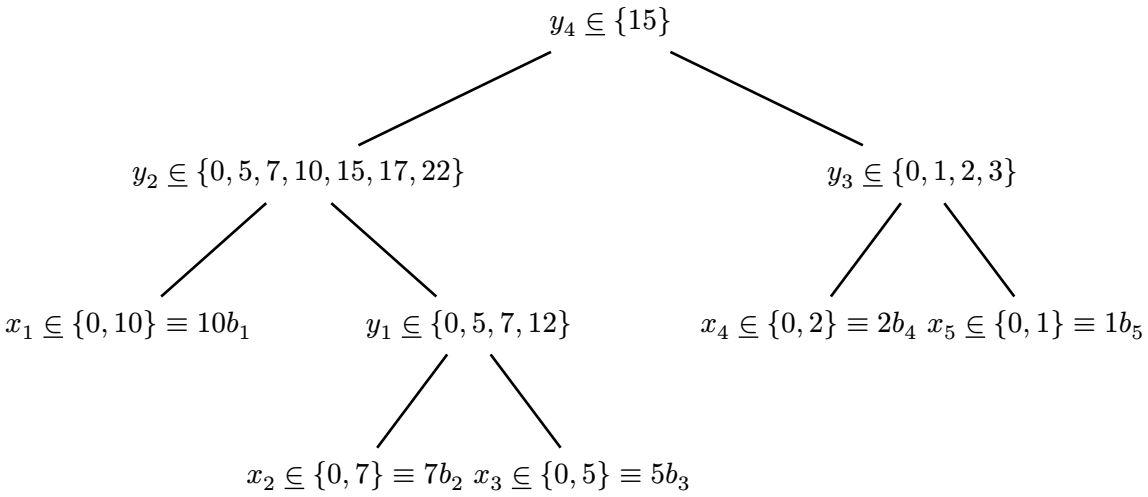
$$2x_1 \in \{0,1\} + 3x_2 \in \{0,1\} + 5x_3 \in \{0,1\} \leq 6 \tag{5.1}$$

The difference between the encodings will be the structure of the decomposition and the domains of the auxiliary integer variables representing the partial sums. The PB encodings discussed here all follow the same binary tree structure, which is either linear or balanced. We can create an arbitrary binary tree with leaf nodes labelled x_i that sums up the n original terms using $n-1$ internal nodes. Each node would be labelled by a new variable $y_i, 1 \leq i < n$ that encodes the partial sum of the tree rooted at internal node i. The final variable y_{n-1} is also denoted as y_{root}.

Each node i creates a *ternary inequality constraint*, $l(i) + r(i) \leq y_i$, where $l(i)$ is either y_l if the left child of node i is an internal node l, or $l(i)$ is x_l if the left child of node i is the leaf node x_l. $r(i)$ is defined similarly. The overall PB constraint is then enforced by restricting the final partial sum in some way, e.g. constraining $y_{\text{root}} \leq k$ (but other approaches are possible).

Different PB encodings create different domains for their auxiliary variables. One example approach would be to restrict the domains $D(y_i) \equiv [0..k]$, since there is no solution where any of them takes a value above k, and to let the root variable $D(y_{\text{root}}) \equiv \{k\}$. Note that the order encoding as specified in Eq. 2.20 does not require any variables or clauses to encode a constant.

Example 5.2 Consider the decomposition shown in Fig. 5.1. It creates the constraints $x_2 + x_3 \leq y_1$, $x_1 + y_1 \leq y_2$, $x_4 + x_5 \leq y_3$, $y_2 + y_3 \leq y_4$, $y_4 \leq 15$. The domains of the y_i variables for a GT encoding are $D(y_1) = \{0, 5, 7, 12\}$, $D(y_2) = \{0, 5, 7, 10, 12, 15\}$, $D(y_3) = \{0, 1, 2, 3\}$, $D(y_4) = \{15\}$.



Once a decomposition has been constructed, the encoding of the PB constraint can be reduced to the repeated encoding of the inequality constraint using *any* integer encoding. The decomposition itself will be an independently correct encoding of the PB constraint. Furthermore, the decomposition is agnostic as to which specific encoding is used for its integer variables x, y and z. One way to encode the ternary inequality constraint is to adapt the uniform order encoding (i.e. $E((x \in D^{\mathbb{Z}}(x) + y \in D^{\mathbb{Z}}(y) \leq z \in D^{\mathbb{Z}}(z)) : \mathbb{O})$) from [42]:

$$\bigwedge_{v \in D(x), w \in D(y)} ([\![x \geq v]\!] \wedge [\![y \geq w]\!]) \rightarrow [\![z \geq v + w]\!] \tag{5.2}$$

Alternative encodings of the ternary inequality constraint can be devised. For example, a recursive definition for linear inequality constraints of arity n, implemented originally for `PBSugar` [96], can generate fewer clauses, and could be used instead. A different integer encoding, such as the binary encoding for some or all variables, is also possible. However, we start from this naive version to show it is the one implicitly used by the PB encodings.

One notable choice of this encoding in Eq. 5.2 is that it does *not* enforce consistency constraints (i.e. `IC` constraints, see Eq. 2.14) on its variables. Consequently, no `IC` constraint is present in the encoding of a PB decomposition. This is in line with the GT, SWC and BDD encoding, which do not encode `IC`s on their auxiliary variables. We will discuss the implications of this further in Sec. 5.2.3.

In each of the following subsections, we first describe such an existing encoding method from the literature. Then, we show a particular decomposition of a PB constraint into ternary inequalities and the domains of the auxiliary integer variables. Finally, we prove how encoding the decomposition using Eq. 5.2 produces an equivalent set of clauses as the original encoding methods for the same PB constraint.

5.2.2 Three PB encoding methods as decompositions

In this section, we explore the GT, SWC and BDD encoding from the integer viewpoint as follows. First, for each PB encoding, we give a specification which is closely adapted from the literature. Then, we specify a PB decomposition. We show an example of its decomposition and eventual SAT encoding on an example PB constraint (namely Eq. 5.1). Finally, we prove that for any PB constraint, the uniform order encoding (specifically Eq. 5.2) of the PB decomposition is equivalent to the clauses of the unit propagated PB encoding for the same PB constraint.

Generalized Totalizer

The Generalized Totalizer (GT) [59] encoding constructs a balanced binary tree bottom-up. Each node Z is associated with a set of auxiliary variables z_v, where v corresponds to the partial sum of the subtree. Each leaf node represents a PB term $q_i b_i$ with singleton set $\{z_{q_i}\}$, where $z_{q_i} \equiv b_i$. From two child nodes, L and R, the parent node is constructed as $Z \equiv \{z_{\min(v+w,k+1)} \mid l_v \in L, r_w \in R\}$. It contains a Boolean variable z_v for every possible sum of the values of the auxiliary variables of child nodes L and R, where any sum greater than k are represented by $k+1$. Then, for every node Z with child nodes L and R, the encoding adds clauses:

$$\bigwedge_{l_v \in L} l_v \rightarrow z_v \tag{5.3.1}$$

$$\wedge \bigwedge_{r_w \in R} r_w \rightarrow z_w \tag{5.3.2}$$

$$\wedge \bigwedge_{l_v \in L, r_w \in R} (l_v \wedge r_w) \rightarrow z_{\min(v+w,k+1)} \tag{5.3.3}$$

If a_{k+1} is an element of the root node A, we add the unary clause $\neg a_{k+1}$ (otherwise the constraint is trivially satisfiable). If an intermediate layer has an odd number of nodes, the last node is lifted to the next layer of the tree.

Definition 5.1 (GT-decomposition) From the integer perspective, the auxiliary Boolean variables of every node Z are represented by a single auxiliary integer variable y_i of a balanced (binary) tree decomposition. The decomposition yields the following ternary inequality constraints:

$$\bigwedge_{i\equiv 1}^{n-1} l(i) + r(i) \leq y_i \tag{5.4}$$

where $D(y_i) \in \{v + w : v \in D(l(i)), w \in D(r(i)), v + w \leq k\}$, except $D(y_{\text{root}}) \equiv \{k\}$.

Example 5.3 The GT decomposition of the example PB constraint from Eq. 5.1 is:

$$\begin{aligned} & 2x_1 \in \{0,1\} + 3x_2 \in \{0,1\} + 5x_3 \in \{0,1\} \leq 6 \\ \equiv\ & x_1 \in \{0,2\} + x_2 \in \{0,3\} \leq y_1 \in \{0,2,3,5\} \wedge y_1 + x_3 \in \{0,5\} \leq y_2 \in \{6\} \end{aligned}$$

The original GT abstraction and its equivalent decomposition are shown in Fig. 5.2. The subsequent encoding of the decomposition yields:

$$\begin{aligned} & ((\llbracket x_1 \geq 0 \rrbracket \wedge \llbracket x_2 \geq 0 \rrbracket) \rightarrow \llbracket y_1 \geq 0 \rrbracket) \wedge ((\llbracket x_1 \geq 0 \rrbracket \wedge \llbracket x_2 \geq 3 \rrbracket) \rightarrow \llbracket y_1 \geq 3 \rrbracket) \wedge \\ & ((\llbracket x_1 \geq 2 \rrbracket \wedge \llbracket x_2 \geq 0 \rrbracket) \rightarrow \llbracket y_1 \geq 2 \rrbracket) \wedge ((\llbracket x_1 \geq 2 \rrbracket \wedge \llbracket x_2 \geq 3 \rrbracket) \rightarrow \llbracket y_1 \geq 5 \rrbracket) \wedge \\ & ((\llbracket x_3 \geq 0 \rrbracket \wedge \llbracket y_1 \geq 0 \rrbracket) \rightarrow \llbracket y_2 \geq 0 \rrbracket) \wedge ((\llbracket x_3 \geq 0 \rrbracket \wedge \llbracket y_1 \geq 2 \rrbracket) \rightarrow \llbracket y_2 \geq 2 \rrbracket) \wedge \\ & ((\llbracket x_3 \geq 0 \rrbracket \wedge \llbracket y_1 \geq 3 \rrbracket) \rightarrow \llbracket y_2 \geq 3 \rrbracket) \wedge ((\llbracket x_3 \geq 0 \rrbracket \wedge \llbracket y_1 \geq 5 \rrbracket) \rightarrow \llbracket y_2 \geq 5 \rrbracket) \wedge \\ & ((\llbracket x_3 \geq 5 \rrbracket \wedge \llbracket y_1 \geq 0 \rrbracket) \rightarrow \llbracket y_2 \geq 5 \rrbracket) \wedge ((\llbracket x_3 \geq 5 \rrbracket \wedge \llbracket y_1 \geq 2 \rrbracket) \rightarrow \llbracket y_2 \geq 7 \rrbracket) \wedge \\ & ((\llbracket x_3 \geq 5 \rrbracket \wedge \llbracket y_1 \geq 3 \rrbracket) \rightarrow \llbracket y_2 \geq 8 \rrbracket) \wedge ((\llbracket x_3 \geq 5 \rrbracket \wedge \llbracket y_1 \geq 5 \rrbracket) \rightarrow \llbracket y_2 \geq 10 \rrbracket) \wedge \\ \equiv\ & (\llbracket x_2 \geq 3 \rrbracket \rightarrow \llbracket y_1 \geq 3 \rrbracket) \wedge (\llbracket x_1 \geq 2 \rrbracket \rightarrow \llbracket y_1 \geq 2 \rrbracket) \wedge ((\llbracket x_1 \geq 2 \rrbracket \wedge \llbracket x_2 \geq 3 \rrbracket) \rightarrow \llbracket y_1 \geq 5 \rrbracket) \wedge \\ & (\llbracket x_3 < 5 \rrbracket \vee \llbracket y_1 < 2 \rrbracket) \wedge (\llbracket x_3 < 5 \rrbracket \vee \llbracket y_1 < 3 \rrbracket) \wedge (\llbracket x_3 < 5 \rrbracket \vee \llbracket y_1 < 5 \rrbracket) \end{aligned}$$

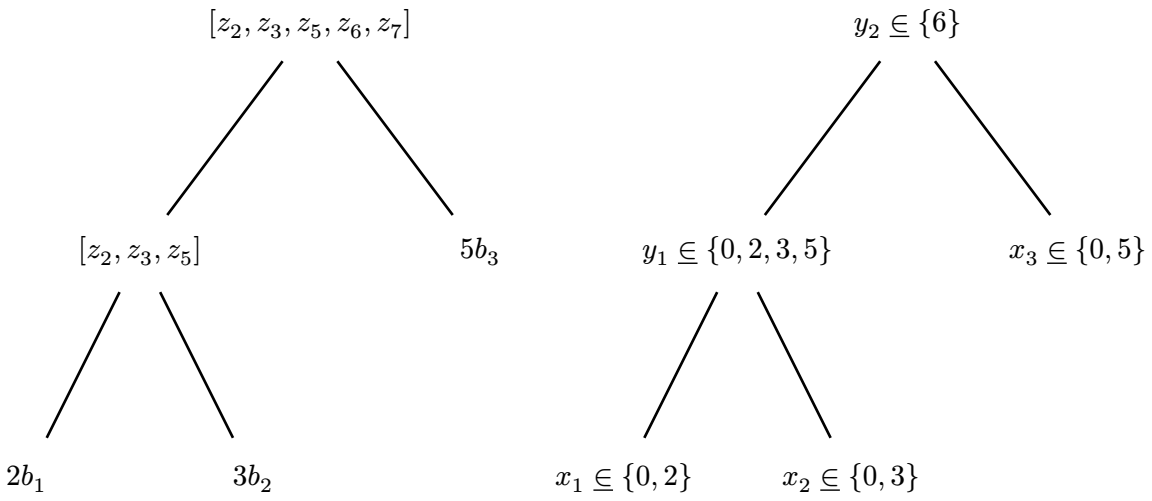


Fig. 5.2: Original GT abstraction and equivalent decomposition.

Theorem 5.1 The encoding of Eq. 5.4 is, after Unit Propagation (UP), equivalent to GT.

Proof. We show that the clauses generated by the balanced tree decomposition are (after closing under UP) the same as those for the GT encodings. We consider a single node Z with child nodes L and R from the GT encoding and corresponding integer constraint $l(i) + r(i) \leq y_i$. The balanced tree of the decomposition is the same as in the GT encoding, where:

$$[\![y_i \geq v]\!] \equiv z_v, [\![l(i) \geq v]\!] \equiv l_v, [\![r(i) \geq v]\!] \equiv r_v$$

Encoding the ternary inequality of Eq. 5.4 yields the following clauses.

$$\begin{aligned}
& E\left(\left(\bigwedge_{i\equiv 1}^{n-1} l(i) + r(i) \leq y_i\right) :\mathbb{O}\right) \\
\equiv & \bigwedge_{v\in D(l(i)), w\in D(r(i))} ([\![l(i) \geq v]\!] \wedge [\![r(i) \geq w]\!]) \rightarrow [\![y_i \geq v + w]\!] \\
\equiv & \left(\bigwedge_{v\in D(l(i))} ([\![l(i) \geq v]\!] \wedge [\![r(i) \geq 0]\!]) \rightarrow [\![y_i \geq v]\!]\right) \\
\wedge & \left(\bigwedge_{w\in D(r(i))} ([\![l(i) \geq 0]\!] \wedge [\![r(i) \geq w]\!]) \rightarrow [\![y_i \geq w]\!]\right) \\
\wedge & \left(\bigwedge_{v\in D(l(i))\setminus\{0\}} \bigwedge_{w\in D(r(i))\setminus\{0\}} ([\![l(i) \geq v]\!] \wedge [\![r(i) \geq w]\!]) \rightarrow [\![y_i \geq v + w]\!]\right)
\end{aligned}$$

This split into multiple conjuncts is correct since 0 is the lower bound of every internal variable y_i, except for y_{root} which is fixed to k. We show these are the same clauses as generated for a GT node. The literals $[\![l(i) \geq 0]\!] \equiv 1$ and $[\![r(i) \geq 0]\!] \equiv 1$ by Eq. 2.27.2. We can skip iterations where $[\![y_i \geq 0]\!] \equiv 1$, since the resulting clause is satisfied. In the GT encoding, the unary clause $\neg a_{k+1}$ will fix $z_{k+1} \equiv 0$ if present in its child node Z, by $z_{k+1} \rightarrow a_{k+1}$. This propagates through the entire tree, so that $z_{k+1} \equiv 0$ for all nodes. Consequently, the first two conjuncts are equivalent to the binary clauses of the GT, Eq. 5.3.1 and Eq. 5.3.2, since $D(l(i)) \equiv \{v \mid l_v \in L \setminus \{l_{k+1}\}\} \cup \{0\}$ and $D(r(i)) \equiv \{v \mid r_v \in R \setminus \{r_{k+1}\}\} \cup \{0\}$:

$$\left(\bigwedge_{v\in D(r(i))\setminus\{0\}} [\![l(i) \geq v]\!] \rightarrow [\![y_i \geq v]\!]\right) \wedge \left(\bigwedge_{w\in D(l(i))\setminus\{0\}} [\![r(i) \geq w]\!] \rightarrow [\![y_i \geq w]\!]\right)$$

The remaining (third case) clauses are UP equivalent to the ternary clauses of the GT decomposition. Since, for any $v + w > k$, the literal $[\![y_i \geq v + w]\!] \equiv 0$, which is simply

falsified (leading to a binary clause, rather than a ternary clause including z_{k+1} in the original encoding).

For the root node, if $v + w \leq k$ then the literal $[\![y_i \geq v + w]\!] \equiv 1$ by Eq. 2.27.2, and so the resulting clauses are trivial. Otherwise, they agree with those in GT. The original GT encoding adds a number of Boolean variables $\{a_v \mid v \leq k\}$ for the root node but since they only appear positively they can be trivially assigned to 1 by preprocessing. □

Sequential Weight Counter

In the Sequential Weight Counter (SWC) [61] encoding, a series of n weight counters is introduced. The i-th counter enforces $\sum_{j\equiv 1}^{i} q_i b_i \leq k$ by introducing auxiliary variables $z_{i-1,j}$ for $1 \leq j \leq k$ where $z_{i,j}$ represents $\sum_{j\equiv 1}^{i} q_i b_i \geq j$. It creates the following clauses.

$$z_{i-1,j} \rightarrow z_{i,j} \qquad 1 \leq j \leq k \tag{5.5.1}$$

$$\neg\big(z_{i-1,k+1-q_i} \wedge b_i\big) \tag{5.5.2}$$

$$\big(z_{i-1,j} \wedge b_i\big) \rightarrow z_{i,j+q_i} \qquad 1 \leq j \leq k - q_i \tag{5.5.3}$$

$$b_i \rightarrow z_{i,j} \qquad 1 \leq j \leq q_i \tag{5.5.4}$$

Note that $z_{0,j} \equiv 0$, for $0 \leq j \leq k$

Definition 5.2 (SWC-decomposition) The SWC encoding can be described using a linear (binary) tree decomposition that introduces $n + 1$ auxiliary integer variables $y_i \in [-k..0]$ except $y_0 \equiv 0$ and $y_n \equiv -k$, and the following ternary inequalities.

$$\bigwedge_{i\equiv 1}^{n} x_i + y_i \leq y_{i-1}$$

These inequalities are derived from $x_i + y'_{i-1} \leq y'_i$ where $y'_i \in [0..k]$, which is equivalent from an integer perspective to $x_i - y'_i \leq -y'_{i-1}$ assuming $y_i \equiv -y'_i$. While the decomposition $x_i + y'_{i-1} \leq y'_i$ would be similar, and simpler, it does lead to a subtle difference in clauses compared to the original SWC encoding.

Example 5.4 The SWC decomposition of the example PB constraint from Eq. 5.1 is:

$$
\begin{aligned}
& 2x_1 \in \{0,1\} + 3x_2 \in \{0,1\} + 5x_3 \in \{0,1\} \leq 6 \\
\equiv\ & x_1 \in \{0,2\} + y_1 \in [-6..0] \leq y_0 \in \{0\} \wedge \\
& x_2 \in \{0,3\} + y_2 \in [-6..0] \leq y_1 \in [-6..0] \wedge \\
& x_3 \in \{0,5\} + y_3 \in \{-6\} \leq y_2 \in [-6..0]
\end{aligned}
$$

The original SWC abstraction and its equivalent decomposition are shown in Fig. 5.3. The subsequent encoding of the decomposition yields:

$$((\llbracket x_1 \geq 0 \rrbracket \wedge \llbracket y_1 \geq -6 \rrbracket) \rightarrow \llbracket y_0 \geq -6 \rrbracket) \wedge \cdots \wedge ((\llbracket x_1 \geq 0 \rrbracket \wedge \llbracket y_1 \geq 0 \rrbracket) \rightarrow \llbracket y_0 \geq 0 \rrbracket) \wedge$$

$$((\llbracket x_1 \geq 2 \rrbracket \wedge \llbracket y_1 \geq -6 \rrbracket) \rightarrow \llbracket y_0 \geq -4 \rrbracket) \wedge ((\llbracket x_1 \geq 2 \rrbracket \wedge \llbracket y_1 \geq -5 \rrbracket) \rightarrow \llbracket y_0 \geq -3 \rrbracket) \wedge \ldots \wedge$$
$$((\llbracket x_1 \geq 2 \rrbracket \wedge \llbracket y_1 \geq -2 \rrbracket) \rightarrow \llbracket y_0 \geq 0 \rrbracket) \wedge ((\llbracket x_1 \geq 2 \rrbracket \wedge \llbracket y_1 \geq -1 \rrbracket) \rightarrow \llbracket y_0 \geq 1 \rrbracket) \wedge$$
$$((\llbracket x_1 \geq 2 \rrbracket \wedge \llbracket y_1 \geq 0 \rrbracket) \rightarrow \llbracket y_0 \geq 2 \rrbracket) \wedge$$

$$((\llbracket x_2 \geq 0 \rrbracket \wedge \llbracket y_2 \geq -6 \rrbracket) \rightarrow \llbracket y_1 \geq -6 \rrbracket) \wedge ((\llbracket x_2 \geq 0 \rrbracket \wedge \llbracket y_2 \geq -5 \rrbracket) \rightarrow \llbracket y_1 \geq -5 \rrbracket) \wedge \ldots \wedge$$
$$((\llbracket x_2 \geq 0 \rrbracket \wedge \llbracket y_2 \geq -1 \rrbracket) \rightarrow \llbracket y_1 \geq -1 \rrbracket) \wedge ((\llbracket x_2 \geq 0 \rrbracket \wedge \llbracket y_2 \geq 0 \rrbracket) \rightarrow \llbracket y_1 \geq 0 \rrbracket) \wedge$$

$$((\llbracket x_2 \geq 3 \rrbracket \wedge \llbracket y_2 \geq -6 \rrbracket) \rightarrow \llbracket y_1 \geq -3 \rrbracket) \wedge ((\llbracket x_2 \geq 3 \rrbracket \wedge \llbracket y_2 \geq -5 \rrbracket) \rightarrow \llbracket y_1 \geq -2 \rrbracket) \wedge$$
$$((\llbracket x_2 \geq 3 \rrbracket \wedge \llbracket y_2 \geq -4 \rrbracket) \rightarrow \llbracket y_1 \geq -1 \rrbracket) \wedge ((\llbracket x_2 \geq 3 \rrbracket \wedge \llbracket y_2 \geq -3 \rrbracket) \rightarrow \llbracket y_1 \geq 0 \rrbracket) \wedge \ldots \wedge$$
$$((\llbracket x_2 \geq 3 \rrbracket \wedge \llbracket y_2 \geq 0 \rrbracket) \rightarrow \llbracket y_1 \geq 3 \rrbracket) \wedge$$

$$((\llbracket x_3 \geq 0 \rrbracket \wedge \llbracket y_3 \geq -6 \rrbracket) \rightarrow \llbracket y_2 \geq -6 \rrbracket) \wedge ((\llbracket x_3 \geq 5 \rrbracket \wedge \llbracket y_3 \geq -6 \rrbracket) \rightarrow \llbracket y_2 \geq -1 \rrbracket)$$

Finally, this is equivalent to:

$$(\llbracket x_1 < 2 \rrbracket \vee \llbracket y_1 < -1 \rrbracket) \wedge (\llbracket x_1 < 2 \rrbracket \vee \llbracket y_1 < 0 \rrbracket) \wedge$$

$$(\llbracket y_2 \geq -5 \rrbracket \rightarrow \llbracket y_1 \geq -5 \rrbracket) \wedge (\llbracket y_2 \geq -5 \rrbracket \rightarrow \llbracket y_1 \geq -4 \rrbracket) \wedge$$
$$(\llbracket y_2 \geq -5 \rrbracket \rightarrow \llbracket y_1 \geq -3 \rrbracket) \wedge (\llbracket y_2 \geq -5 \rrbracket \rightarrow \llbracket y_1 \geq -2 \rrbracket) \wedge$$
$$(\llbracket y_2 \geq -1 \rrbracket \rightarrow \llbracket y_1 \geq -1 \rrbracket) \wedge$$

$$(\llbracket x_2 \geq 3 \rrbracket \rightarrow \llbracket y_1 \geq -3 \rrbracket) \wedge ((\llbracket x_2 \geq 3 \rrbracket \wedge \llbracket y_2 \geq -5 \rrbracket) \rightarrow \llbracket y_1 \geq -2 \rrbracket) \wedge$$
$$((\llbracket x_2 \geq 3 \rrbracket \wedge \llbracket y_2 \geq -4 \rrbracket) \rightarrow \llbracket y_1 \geq -1 \rrbracket) \wedge$$

$$(\llbracket x_3 \geq 5 \rrbracket \rightarrow \llbracket y_2 \geq -1 \rrbracket)$$

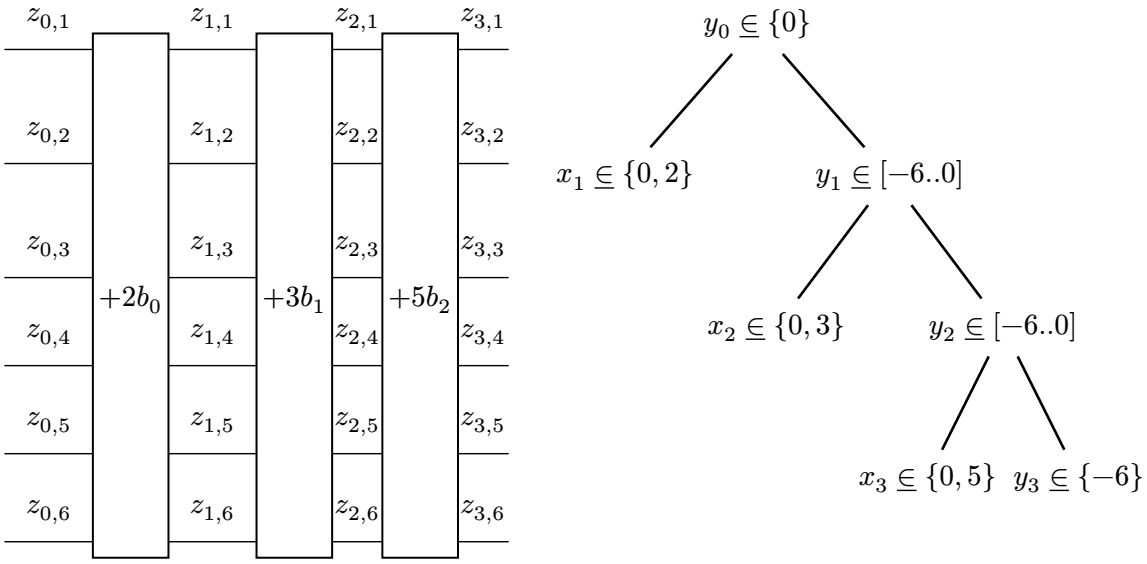


Fig. 5.3: Original SWC abstraction and equivalent decomposition. Note that the orientation of a SWC is traditionally left-to-right.

Theorem 5.2 The encoding of Eq. 5.5 is, after UP, equivalent to SWC.

Proof. Encoding Eq. 5.5 yields the following clauses:

$$
\begin{aligned}
& E\left(\left(\bigwedge_{i\equiv 1}^{n} x_i + y_i \leq y_{i-1}\right) : \mathbb{O}\right) \\
\equiv & \bigwedge_{v\in D(x_i)} \bigwedge_{u\in D(y_i)} (\llbracket x_i \geq v \rrbracket \wedge \llbracket y_i \geq w \rrbracket) \rightarrow \llbracket y_{i-1} \geq v + u \rrbracket \\
\equiv & \bigwedge_{v\in D(x_i)} \bigwedge_{u\in D(y_i)} (\llbracket x_i \geq v \rrbracket \wedge \llbracket y_{i-1} < v + u \rrbracket) \rightarrow \llbracket y_i < u \rrbracket
\end{aligned}
$$

We split the conjunctions:

$$\left(\bigwedge_{u\equiv -k}^{0}([\![x_i \geq 0]\!] \wedge [\![y_{i-1} < 0+u]\!]) \rightarrow [\![y_i < u]\!]\right) \tag{5.6.1}$$

$$\wedge\,([\![x_i \geq q_i]\!] \wedge [\![y_{i-1} < q_i - k]\!] \rightarrow [\![y_i < -k]\!]) \tag{5.6.2}$$

$$\wedge\left(\bigwedge_{u\equiv -k+1}^{-q_i}([\![x_i \geq q_i]\!] \wedge [\![y_{i-1} < q_i+u]\!]) \rightarrow [\![y_i < u]\!]\right) \tag{5.6.3}$$

$$\wedge\left(\bigwedge_{u\equiv -q_i+1}^{0}([\![x_i \geq q_i]\!] \wedge [\![y_{i-1} < q_i+u]\!]) \rightarrow [\![y_i < u]\!]\right) \tag{5.6.4}$$

When we equate $[\![y_i < u]\!] \equiv z_{i,-u+1}$ for $1 \leq i \leq n, -k \leq u \leq 0$ we can show that the four cases above correspond to the four cases in the SWC encoding. The first case Eq. 5.6.1 $[\![x_i \geq 0]\!] \equiv 1$, and $[\![y_i - 1 < -k]\!] \equiv 0$, and the resulting clauses are equivalent after UP to Eq. 5.5.1 and removing the trivial clause (when $u \equiv -k$). The second case Eq. 5.6.2 the clauses are equivalent once we note that $[\![y_i < -k]\!] \equiv 0$. The third case Eq. 5.6.3, produces exactly the clauses from Eq. 5.5.3, generated in reverse order, i.e. $\bigwedge_{u\equiv -k+1}^{-q_i}\big(\big(b_i \wedge z_{i-1,-q_i-u)}\big) \rightarrow z_{i,-u}\big) \equiv \bigwedge_{l=1}^{k-q_i}\big(\big(b_i \wedge z_{i-1,k-q_i-l}\big) \rightarrow z_{i,k-l}\big)$, where $l \equiv u+k$. Finally, in Eq. 5.6.4 $[\![y_{i-1} < q_i+u]\!] \equiv 1$ (since q_i+u is greater than the upper bound 0 of y_{i-1}, by $q_i+u \geq q_i - q_i + 1 > 0$ the upper bound of y_{i-1}). □

Binary Decision Diagram

A Binary Decision Diagram (BDD) [58] is a rooted, directed, acyclic graph which models a Boolean function over n Boolean variables, $b_1, ..., b_n$. There are two terminal nodes, $z_\bot$ and $z_\top$, representing `false` and `true`. Every other node has two outgoing edges labeled 0 and 1, and a selector variable which determines whether a path follows either the node's zero or one edge. When the path reaches a terminal node, the function returns zero or one, respectively. In an ordered BDD, selector variables appear in the same order on all paths from the root, organizing the BDD into one layer per selector variable. A Quasi-reduced Ordered BDD (QOBDD) has no isomorphic sub-BDDs. A Reduced Ordered BDD (ROBDD) is quasi-reduced and has no *identity* nodes, i.e. nodes of which all edges target the same child. A construction algorithm exists which produces ROBDDs that model the satisfaction of PB constraints [91].

A constructed ROBDD can be encoded to SAT as follows. Layer i models the decision of Boolean variable b_i with multiple nodes (i, j). Layer $n+1$ represents the terminal nodes $(n+1, 1)$ is $z_\bot$ and $(n+1, 2)$ is $z_\top$. Each node is also associated with an auxiliary Boolean variable, $z_{i,j}$. Let E be the set of edges of the ROBDD of the form $((i, j), (i', j'), p) \in E$, where (i, j) is the source node, (i', j') is the target node, and $p \in \{0, 1\}$ is the label. We call an edge $((i, j), (i', j'), p)$ a p -edge. The encoding adds a clause $z_{i,j} \rightarrow z_{i',j'}$ for every `false` edge $((i, j), (i', j'), 0)$ where $i <$

i', and $\big(b_i \wedge z_{i,j}\big) \rightarrow z_{i'',j''}$ for every `true` edge $((i,j),(i'',j''),1)$ where $i < i''$. The constraint is enforced by adding unary clauses $z_{1,1}$ (root), $\neg z_{\bot}$ and $z_{\top}$.

Before moving to the integer perspective, we establish two general properties of BDDs. First, a ROBDD can be converted to an equivalent QOBDD by replacing any *long* edges $((i,j),(i',j'),v)$ where $i' - i > 1$ by chains of $i' - i - 1$ identity nodes, so that $i' \equiv i + 1$ for all edges. There is no encoding overhead, since we can reuse $z_{i,j}$ for the identity nodes, and satisfy the tautological clauses from edges between them which will have $z_{i,j}$ as both antecedent and consequent. Consequently, in a QOBDD, $((i,j),(i',j'),p) \in E$ has $i \equiv i' - 1$.

Second, the above ROBDD construction algorithm uses so-called *PB intervals*, which we adapt here. A path from root node $(1,1)$ to node (i,j) is associated with a partial assignment of $b_1, ..., b_{i-1}$, which yields a partial sum. Let $W_{i,j}$ be the set of all partial sums given by all possible paths to node (i,j). In a ROBDD, the sub-ROBDD rooted at node (i,j) covers all PB constraints $w + \sum_{i' \equiv i}^{n} q_i b_i \leq k$ in the interval $\min W_{i,j} \leq w \leq \max W_{i,j}$. Hence, intervals on the same layers do not overlap, as that would produce identical ROBDDs, which would contradict the isomorphism property.

Definition 5.3 (BDD-decomposition) Like the original encoding, the integer perspective also starts from a constructed ROBDD, which we then convert to a QOBDD without long edges. Then, we represent each layer i with a single integer variable y_i which acts as the partial sum $\sum_{j \equiv 1}^{i} x_j \leq y_i$ to construct a *linear* (binary) tree. This is enforced by the following linear tree of ternary inequalities:

$$\bigwedge_{i \equiv 1}^{n} x_i + y_{i-1} \leq y_i \tag{5.7}$$

The domain values 0 and q_i of x_i represent the zero and one edges at layer i, respectively. The j-th domain value $d_{i,j}$ of y_i is represented by node (i,j) and is equal to $\max W_{i,j}$, except for $y_0 \equiv 0$ and $y_n \equiv k$. The domain values can also be recursively calculated, bottom-up:

$$d_{i,j} \equiv \max\big\{d_{i-1,j} + vq_i \mid ((i-1,j'),(i,j),v) \in E\big\} \tag{5.8}$$

Example 5.5 The BDD decomposition of the example PB constraint from Eq. 5.1 is:

$$
\begin{aligned}
& 2x_1 \subseteq \{0,1\} + 3x_2 \subseteq \{0,1\} + 5x_3 \subseteq \{0,1\} \leq 6 \\
\equiv\ & x_1 \subseteq \{0,2\} + y_0 \subseteq \{0\} \leq y_1 \subseteq \{1,2\} \wedge \\
& x_2 \subseteq \{0,3\} + y_1 \subseteq \{1,2\} \leq y_2 \subseteq \{1,5\} \wedge \\
& x_3 \subseteq \{0,5\} + y_2 \subseteq \{1,5\} \leq y_3 \subseteq \{6\}
\end{aligned}
$$

The original BDD abstraction and its equivalent decomposition are shown in Fig. 5.4. The subsequent encoding of the decomposition yields:

$$
\begin{aligned}
& ((\llbracket x_1 \geq 0 \rrbracket \wedge \llbracket y_0 \geq 0 \rrbracket) \rightarrow \llbracket y_1 \geq 0 \rrbracket) \wedge ((\llbracket x_1 \geq 2 \rrbracket \wedge \llbracket y_0 \geq 0 \rrbracket) \rightarrow \llbracket y_1 \geq 2 \rrbracket) \wedge \\
& ((\llbracket x_2 \geq 0 \rrbracket \wedge \llbracket y_1 \geq 1 \rrbracket) \rightarrow \llbracket y_2 \geq 1 \rrbracket) \wedge ((\llbracket x_2 \geq 0 \rrbracket \wedge \llbracket y_1 \geq 2 \rrbracket) \rightarrow \llbracket y_2 \geq 2 \rrbracket) \wedge \\
& ((\llbracket x_2 \geq 3 \rrbracket \wedge \llbracket y_1 \geq 1 \rrbracket) \rightarrow \llbracket y_2 \geq 4 \rrbracket) \wedge ((\llbracket x_2 \geq 3 \rrbracket \wedge \llbracket y_1 \geq 2 \rrbracket) \rightarrow \llbracket y_2 \geq 5 \rrbracket) \wedge \\
& ((\llbracket x_3 \geq 0 \rrbracket \wedge \llbracket y_2 \geq 1 \rrbracket) \rightarrow \llbracket y_3 \geq 1 \rrbracket) \wedge ((\llbracket x_3 \geq 0 \rrbracket \wedge \llbracket y_2 \geq 5 \rrbracket) \rightarrow \llbracket y_3 \geq 5 \rrbracket) \wedge \\
& ((\llbracket x_3 \geq 5 \rrbracket \wedge \llbracket y_2 \geq 1 \rrbracket) \rightarrow \llbracket y_3 \geq 6 \rrbracket) \wedge ((\llbracket x_3 \geq 5 \rrbracket \wedge \llbracket y_2 \geq 5 \rrbracket) \rightarrow \llbracket y_3 \geq 10 \rrbracket) \\
\equiv\ & (\llbracket x_1 \geq 2 \rrbracket \rightarrow \llbracket y_1 \geq 2 \rrbracket) \wedge (\llbracket y_1 \geq 2 \rrbracket \rightarrow \llbracket y_2 \geq 2 \rrbracket) \wedge (\llbracket x_2 \geq 3 \rrbracket \rightarrow \llbracket y_2 \geq 4 \rrbracket) \wedge \\
& ((\llbracket x_2 \geq 3 \rrbracket \wedge \llbracket y_1 \geq 2 \rrbracket) \rightarrow \llbracket y_2 \geq 5 \rrbracket) \wedge (\llbracket x_3 < 5 \rrbracket \vee \llbracket y_2 < 5 \rrbracket)
\end{aligned}
$$

Finally, we also apply Eq. 2.27.3 resolving $\llbracket y_2 \geq 4 \rrbracket \equiv \llbracket y_2 \geq 5 \rrbracket$ in the 3rd clause.

$$
\begin{aligned}
\equiv\ & (\llbracket x_1 \geq 2 \rrbracket \rightarrow \llbracket y_1 \geq 2 \rrbracket) \wedge (\llbracket y_1 \geq 2 \rrbracket \rightarrow \llbracket y_2 \geq 2 \rrbracket) \wedge (\llbracket x_2 \geq 3 \rrbracket \rightarrow \llbracket \boldsymbol{y_2 \geq 5} \rrbracket) \wedge \\
& ((\llbracket x_2 \geq 3 \rrbracket \wedge \llbracket y_1 \geq 2 \rrbracket) \rightarrow \llbracket y_2 \geq 5 \rrbracket) \wedge (\llbracket x_3 < 5 \rrbracket \vee \llbracket y_2 < 5 \rrbracket)
\end{aligned}
$$

Crucially, this shows how multiple edges to the same node in the next BDD layer is analogous to the addition of domain values coinciding in the same domain gap.

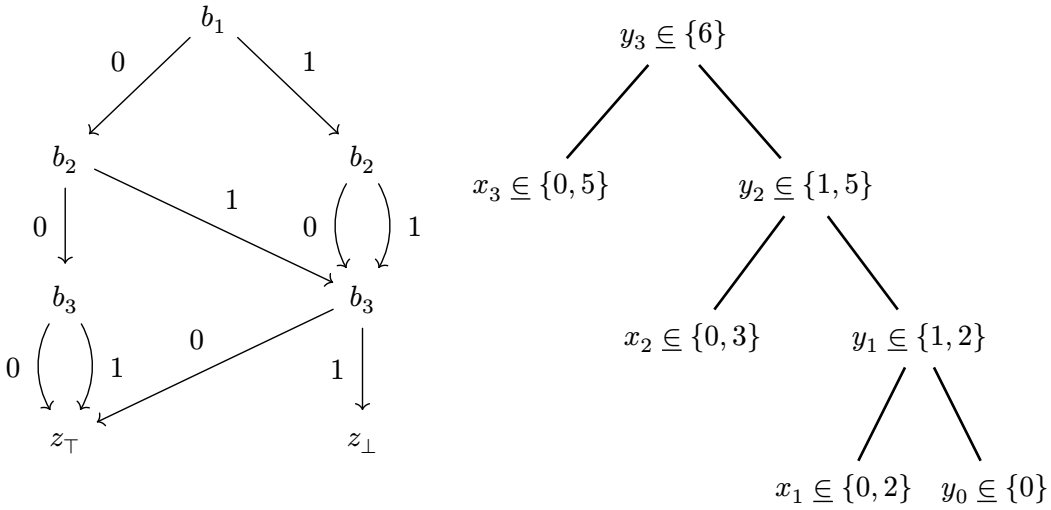


Fig. 5.4: Original BDD abstraction and equivalent decomposition. Note that the orientation of a BDD is traditionally top-to-bottom.

We note that:

- some non-reduced BDDs cannot be represented by our encoding of the ternary inequality constraints; and
- other BDD encodings exist might require other integer variable encodings. The `CompletePath` encoding includes the `EO` constraint [110], suggesting an underlying direct encoding.

Theorem 5.3 The encoding of Eq. 5.7 is, after UP, equivalent to BDD.

Proof. The encoding $E((x_i + y_{i-1} \leq y_i){:}\mathbb{O})$ yields clauses:

$$\bigwedge_{v \in D(x_i), d_{i-1,j} \in D(y_{i-1})} (\llbracket x_i \geq v \rrbracket \wedge \llbracket y_{i-1} \geq d_{i-1,j} \rrbracket) \rightarrow \llbracket y_i \geq d_{i-1,j} + v \rrbracket \tag{5.9}$$

Every edge $((i-1,j),(i,j'),p) \in E$ of the QOBDD corresponds to exactly one clause in Eq. 5.9 with antecedents $\llbracket x_i \geq v \rrbracket$ (where $v \equiv pq_i$) and $\llbracket y_{i-1} \geq d_{i-1,j} \rrbracket$. It remains to be shown that the clause's consequent $\llbracket y_i \geq d_{i-1,j} + v \rrbracket$ matches the literal $\llbracket y_i \geq d_{i,j'} \rrbracket$ of target node (i,j'). If $d_{i-1,j} \equiv \max W_{i-1,j}$, and there is a p-edge between $(i-1,j)$ and (i,j'), then $\max W_{i,j'-1} < d_{i,j} + v \leq \max W_{i,j'}$. By the domain construction of Eq. 5.8, we have $d_{i,j'-1} < w \leq d_{i,j'}$. Consequently, $\llbracket y_i \geq w \rrbracket \equiv \llbracket y_i \geq d_{i,j'} \rrbracket$ by Eq. 2.27.3. If edge is a 0 edge, i.e. $p \equiv 0$, then $\llbracket x_i \geq v \rrbracket$ by Eq. 2.27.2, which simplifies to the binary clause $\llbracket y_{i-1} \geq d_{i-1,j} \rrbracket \rightarrow \llbracket y_i \geq d_{i-1,j} \rrbracket$. □

5.2.3 Extensions

Now that we have specified the PB encodings as decomposition, we can introduce various extensions of the decomposition, which will apply to each encoding. The extensions are a theoretical proof of propagation strength, support for IL constraints, and support for IL, for side-constraints, for equality comparators, for domain reductions, and for the binary encoding of integer variables.

Proving consistency of all encodings

As shown in their seminal papers, the GT, SWC and BDD encodings maintain domain consistency. Instead of proving domain consistency on every encoding individually, we can prove it for the ternary inequality, and use this to prove domain consistency (see Def. 2.7) of every decomposition.

As mentioned in Sec. 5.2.1, no `IC` is added by the original encoding or ternary inequality constraint encoding of Eq. 5.2. Consequently, we can construct a ternary inequality constraint $x + y \leq z$ for which Eq. 5.2 does not maintain domain consistency. However, we will show that the full decomposition does maintain domain consistency.

Proposition 5.1 The encoding of $x + y \leq z$ given by Eq. 5.2 does not maintain domaion consistency (see Def. 2.8).

Proof. By counterexample. Consider the domains $D(x) \equiv \{0, 4, 8\}$, $D(y) \equiv \{0, 7, 10\}$ and $D(z) \equiv \{0, 4, 7, 8, 10, 11\}$ then suppose we have $[\![y \geq 7]\!]$ and $[\![z < 10]\!]$. By $y \geq 7$ and $z \leq 9$, we hope to propagate $x \leq 2$, or equivalently the literal $[\![x < 4]\!]$. But unless z has the `IC` constraints this may not occur, since the clause which does this propagation is $[\![x < 4]\!] \vee [\![y < 7]\!] \vee [\![z \geq 11]\!]$, and without the `IC` clause $[\![z \geq 11]\!] \rightarrow [\![z \geq 10]\!]$ we have no guarantee that $\neg[\![z \geq 11]\!]$ holds.

$$
\begin{aligned}
&(\![x < 8]\!] \vee [\![x \geq 4]\!]) \wedge ([\![y < 10]\!] \vee [\![y \geq 7]\!]) \wedge ([\![x < 4]\!] \vee [\![z \geq 4]\!]) \\
&\wedge ([\![x < 4]\!] \vee [\![y < 7]\!] \vee [\![z \geq 11]\!]) \wedge ([\![x < 4]\!] \vee [\![y < 10]\!]) \wedge ([\![x < 8]\!] \vee [\![z \geq 8]\!]) \\
&\qquad \wedge ([\![x < 8]\!] \vee [\![y < 7]\!]) \wedge ([\![x < 8]\!] \vee [\![y < 10]\!]) \quad \textit{dec.} \quad [\![y \geq 7]\!] \\
&([\![x < 8]\!] \vee [\![x \geq 4]\!]) \wedge ([\![x < 4]\!] \vee [\![z \geq 4]\!]) \wedge ([\![x < 4]\!] \vee [\![z \geq 11]\!]) \\
&\qquad \wedge ([\![x < 4]\!] \vee [\![y < 10]\!]) \wedge ([\![x < 8]\!] \vee [\![z \geq 8]\!]) \wedge [\![x < 8]\!] \\
&\qquad \wedge ([\![x < 8]\!] \vee [\![y < 10]\!]) \quad \Longrightarrow \quad [\![x < 8]\!] \\
&([\![x < 4]\!] \vee [\![z \geq 4]\!]) \wedge ([\![x < 4]\!] \vee [\![z \geq 11]\!]) \wedge ([\![x < 4]\!] \vee [\![y < 10]\!]) \quad \textit{dec.} \quad [\![z < 10]\!] \\
&([\![x < 4]\!] \vee [\![z \geq 4]\!]) \wedge ([\![x < 4]\!] \vee [\![z \geq 11]\!]) \wedge ([\![x < 4]\!] \vee [\![y < 10]\!]) \quad \textit{fixp.}
\end{aligned}
$$

If we add the IC constraint on z to the resulting formula at fixpoint: and try to propagate $[\![z < 10]\!]$ again:

$$
\begin{aligned}
&([\![x < 4]\!] \vee [\![z \geq 4]\!]) \wedge ([\![x < 4]\!] \vee [\![z \geq 11]\!]) \wedge ([\![x < 4]\!] \vee [\![y < 10]\!]) \\
&\wedge ([\![z < 7]\!] \vee [\![z \geq 4]\!]) \wedge ([\![z < 8]\!] \vee [\![z \geq 7]\!]) \wedge ([\![z < 10]\!] \vee [\![z \geq 8]\!]) \\
&\qquad\qquad \wedge ([\![z < 11]\!] \vee [\![z \geq 10]\!]) \quad \textit{dec. } [\![z < 10]\!] \\
&([\![x < 4]\!] \vee [\![z \geq 4]\!]) \wedge ([\![x < 4]\!] \vee [\![z \geq 11]\!]) \wedge ([\![x < 4]\!] \vee [\![y < 10]\!]) \\
&\qquad \wedge ([\![z < 7]\!] \vee [\![z \geq 4]\!]) \wedge ([\![z < 8]\!] \vee [\![z \geq 7]\!]) \wedge [\![z < 11]\!] \implies [\![z < 11]\!] \\
&\qquad ([\![x < 4]\!] \vee [\![z \geq 4]\!]) \wedge [\![x < 4]\!] \wedge ([\![x < 4]\!] \vee [\![y < 10]\!]) \\
&\qquad\qquad \wedge ([\![z < 7]\!] \vee [\![z \geq 4]\!]) \wedge ([\![z < 8]\!] \vee [\![z \geq 7]\!]) \implies [\![x < 4]\!] \\
&\qquad\qquad ([\![z < 7]\!] \vee [\![z \geq 4]\!]) \wedge ([\![z < 8]\!] \vee [\![z \geq 7]\!]) \quad \textit{fixp.}
\end{aligned}
$$

Then, UP does set $\neg[\![z \geq 11]\!]$, followed by $[\![x < 4]\!]$. □

However, we can prove something slightly weaker: if an IC constraint is somehow enforced on z (when z is said to be closed under IC), then the encoding of the ternary inequality constraint does maintain domain consistency, and it closes x and y under the IC as well.

Theorem 5.4 Encoding $x + y \leq z$ with Eq. 5.2 ensures that lower bounds on x and y are propagated correctly to z (or cause failure), and when an upper bound on z is closed under the IC, then propagation of upper bounds on x and y are correctly propagated, and these propagations are also closed under the IC.

Proof. No assumptions are made on the domains, except that all are non-empty (otherwise the problem would be trivially unsatisfiable). However, we do know that the clause $([\![x \geq v]\!] \wedge [\![y \geq w]\!]) \rightarrow [\![z \geq v + w]\!]$ exists if and only if $v \in D(x)$ and $w \in D(y)$.

Lower Bound Propagation Given a fixed lower bound v for x and lower bound w for y, then since these bounds are in $D(x)$ and $D(y)$ respectively we have $[\![x \geq v]\!]$ and $[\![y \geq w]\!]$ hold. Hence, the clause $[\![x \geq v]\!] \wedge [\![y \geq w]\!] \rightarrow [\![z \geq v + w]\!]$ will propagate the correct bound (or cause failure if $v + w > ub(z)$).

Upper Bound Propagation Suppose we have fixed upper bound u for z, then we have $[\![z \leq u]\!] \equiv \neg[\![z \geq u']\!], d_i^z \leq u < u' \equiv d_{i+1}^z$ for some adjacent domain values $d_i^z, d_{i+1}^z \subseteq D(z)$. So $\neg[\![z \geq u']\!]$ holds. By the closure assumption, $\neg[\![z \geq u'']\!], u'' > u'$ also holds.

Suppose we have fixed lower bound $w \in D(y)$ for y (which might be the original lower bound), then we know that $[\![y \geq w]\!]$ holds. Propagation should enforce that $x \leq u - w$. Now $[\![x \leq u - w]\!] \equiv \neg[\![x \geq u - w + 1]\!] \equiv \neg[\![x \geq v]\!], d^x_j < u - w + 1 \leq v \equiv d^x_{j+1}$ for some adjacent domain values $d^x_i, d^x_{j+1} \subseteq D(x)$. Consider the clause $[\![x \geq v]\!] \wedge [\![y \geq w]\!] \rightarrow [\![z \geq v + w]\!]$. Since $v + w \geq (u - w + 1) + w \equiv u + 1 \geq u'$ we have that $\neg[\![z \geq v + w]\!]$ holds by the closure assumption, and the clause propagates $\neg[\![x \geq v]\!]$. Note also for any value $v' > v, v' \in D(x)$ we have the clause $[\![x \geq v']\!] \wedge [\![y \geq w]\!] \rightarrow [\![z \geq v' + w]\!]$, which propagates $\neg[\![x \geq v']\!]$ again by the closure assumption on z. This proves that upper bounds are correctly propagated, and any propagated upper bound also satisfies the closure condition. □

A consequence of Thm. 5.4 is that the encoding of a PB constraint into a tree decomposition of ternary inequalities does maintain domain consistency.

Theorem 5.5 The decomposition of a PB constraint $\sum_{i \equiv 1}^{n} q_i b_i \leq k$ (without `ICs`) into ternary inequalities encoded using Eq. 5.2 enforces domain consistency of the constraint.

Proof. First, domain consistency on linear inequality constraints is equivalent to bounds($\mathbb{R}$) consistency. Next, (non-zero) lower bounds on the leaf integers $x_i \equiv q_i b_i$ are set by the fact that $[\![x_i \geq q_i]\!] \equiv b_i$. Thm. 5.4 shows that the lower bounds are correctly propagated up the tree. For the top most inequality $l(\text{root}) + r(\text{root}) \leq y_{\text{root}}$ the closure condition holds automatically since y_{root} has a singleton domain, thus all literals mentioning y_{root} are either true or false by definition. This means upper bounds propagate to the children of the root correctly and are closed under the chain condition. By induction, using Thm. 5.4 all upper bounds propagated by the tree are correct and always satisfy the closure condition. Finally, if an upper bound propagated on the leaf integer $x_i \equiv q_i b_i$ removes value q_i then it sets $\neg[\![x_i \geq q_i]\!] \equiv \neg b_i$. □

Note that the counterexample of Prop. 5.1 does not conflict with the above theorem, since it cannot occur, unless we branch on intermediate literals, in this case by setting $\neg[\![z \geq 10]\!]$. Note the proof given for this result for GT by [59] seems to never mention the chain condition explicitly, which suggests it may actually be incomplete.

In order to ensure domain consistency even when branching on intermediate variables, `ICs` (see Eq. 2.14) can be added to Boolean representations of intermediate variables y_i (or $-w_i$). While this does not change the meaning of the constraint, it allows more propagation when the solver makes decisions about these variables.

Using Integer Decision Variables

An IL constraint $\sum_i^n q_i x'_i \leq k$ over order encoded integer variables x'_i and constants q_i, k can be converted to a PB constraint by splitting out every IL term $q_i x'_i$ into multiple PB terms using $\sum_{v \in D(x'_i)} q_i [\![x'_i \geq v]\!]$. However, encoding the resulting PB constraint leads to a high degree of redundancy, since it introduces a GT leaf node, SWC counter or BDD layer per domain value, for every IL term.

Since our ternary inequalities are defined over integers, the PB encoding methods are easily extended to IL encodings. As in Sec. 5.2, we use views $x_i \equiv q_i x'_i$ where $D(x_i) \equiv \{q_i v \mid v \in D(x'_i)\}$ to establish a linear constraint with unit coefficients $\sum_{i \equiv 1}^n x_i \leq k$. Encoding the ternary inequalities results in just a single leaf node, counter, or layer per term. In particular, BDDs are thus generalized to MDDs, as every node on layer i will have one edge per domain value of x_i (see e.g. [64]).

Finding more integers

Even if a constraint is not explicitly modelled using integer variables, we can find more integers. For this, we can make use of work that discovers `AMO` or `IC` side constraints (see Eq. 2.12 and Eq. 2.14, respectively), since sets of literals which are subject to such side-constraints can be replaced by an integer variable with little or no overhead.

Given a PB constraint $\sum_{i \equiv 1}^n q_i b_i \leq k$, suppose there is an `AMO` side-constraint $\texttt{AMO}([b_1, ..., b_l]), l \leq n$ (w.l.o.g. we assume the first l terms are constrained so that at most one is true). This constraint is either explicitly encoded using any of the `AMO` constraint encodings [16], or it can be inferred [104]. [107] explore how to use this to improve various PB encodings. We can recreate the improvements by replacing the term $q_1 b_1 + \cdots + q_1 b_l$ by an integer x with domain $D(x) \equiv \{0\} \cup \{q_i \mid 1 \leq i \leq l\}$ since these are the only values that it can take. We can then encode $q_1 b_1 + \cdots + q_1 b_l \leq x$ succinctly. If q_i is a unique coefficient in the first l terms then $[\![x \geq q_i]\!] \equiv b_i$. Otherwise, we introduce a new Boolean $[\![x \geq q_i]\!]$ and add the channelling constraint $b_i \rightarrow [\![x \geq q_i]\!]$. This partial view encoding of x is very compact (and, if all q_i are unique, introduces no new Booleans).

> **Example 5.6** Suppose $2b_1 + 3b_2 + 3b_3 + ... \leq k \wedge \texttt{AMO}([b_1, b_2, b_3])$, then we can instead encode $x{:}\mathbb{O} \subseteq \{0, 2, 3\} + ... \leq k$ with order encoded x variable $E(x{:}\mathbb{O} \subseteq \{0, 2, 3\}) \equiv (b_1 \equiv [\![x \geq 2]\!]) \wedge b_2 \rightarrow [\![x \geq 3]\!] \wedge b_3 \rightarrow [\![x \geq 3]\!]$. Note there is no need for further consistency constraints on this order encoded x variable (i.e. $[\![x \geq 3]\!] \rightarrow [\![x \geq 2]\!]$).

Given a PB constraint $\sum_{i \equiv 1}^n q_i b_i \leq k$, suppose there is an `IC` side-constraint $\texttt{IC}([b_l, ..., b_1]), l \leq n$ (w.l.o.g. we assume the first l terms contain the chain). We can replace the term $q_1 b_1 +$

$\cdots + q_1 b_l$ by an integer x with domain $D(x) \equiv \{0, q_1, q_1 + q_2, ..., q_1 + \cdots + q_l\}$, since these are the only values the term can take. Even better, the existing Boolean variables already give an order encoding of x since $[\![x \geq \sum_{j \equiv 1}^{v} q_j]\!] \equiv b_v$ for $1 \leq v \leq l$. Hence, we avoid any encoding of the addition of the first l PB terms. For BDDs, this results in the same encoding as [108], but creates novel encodings for GT and SWC. An alternate, more redundant approach is to convert the `IC` group to an `AMO` group, so we can apply the PB(AMO) constraint [126].

Example 5.7 Suppose $2b_1 + 3b_2 + 3b_3 + ... \leq k \wedge (b_1 \leftarrow b_2 \leftarrow b_3))$, then we can instead encode $x{:}\mathbb{O} \subseteq \{0, 2, 3, 6\}{:}\mathbb{O} + ... \leq k$ with order encoded x variable $E(x{:}\mathbb{O} \subseteq \{0, 2, 3, 6\}) \equiv (b_1 \equiv [\![x \geq 2]\!]) \wedge (b_2 \equiv [\![x \geq 3]\!]) \wedge (b_3 \equiv [\![x \geq 6]\!])$.

Given a PB constraint $\sum_{i \equiv 1}^{n} q_i b_i \leq k$, suppose that there are shared coefficients $q_1 \equiv \cdots \equiv q_l, l \leq n$ (w.l.o.g. we assume the first l coefficients are the same). We can replace the term $q_1 b_1 + \cdots + q_1 b_l$ by an integer term $q_1 x$ where $D(x) \equiv [0..l]$ and encode the constraint $b_1 + b_2 + ... \leq x$ using a cardinality network (e.g. using [65]). This performs the addition in a more compact way, and generates new auxiliary variables for x that recognize symmetric situations, and has been shown to improve encodings [64]. Note that integer term $q_1 x$ is simply a view on the x variable, so that we do not need to introduce additional Booleans to encode it [108].

Example 5.8 Suppose $2b_1 + 2b_2 + 2b_3 + ... \leq k$, then we can instead encode $2 \cdot (x{:}\mathbb{O} \in [0..3]) + ... \leq k$ (using an affine view, see Sec. 3.3.6) with order encoded x variable $b_1 + b_2 + b_3 \leq x{:}\mathbb{O} \in [0..3]$ using a cardinality encoding.

Encoding equality constraints

Usually, an equality PB constraint $\sum_{i \equiv 1}^{n} q_i b_i = k$ is split into two inequalities $\sum_{i \equiv 1}^{n} q_i b_i \leq k$ and $\sum_{i \equiv 1}^{n} q_i b_i \geq k$. However, this creates two sets of auxiliary variables. Encoding the equality constraint prevents this. We can do so straightforwardly by decomposing the equality PB constraint into ternary *equalities* of the form $x + y = z$, and then generating an encoding of these directly. Since domain consistency is NP-hard (by reduction to subset sum), we will only enforce bounds($\mathbb{R}$) consistency, which is equivalent to the consistency enforced by splitting into two inequalities [95].

The encoding simply reuses the PB encoding methods for $\sum_{i \equiv 1}^{n} q_i b_i \leq k$ with the following changes. Instead of decomposing to ternary inequalities $x + y \leq z$, we decompose to ternary equalities $x + y = z$. The domain computation for each introduced integer variable remains the same. The given ROBDD should model the equality constraint, which can be achieved by changing the base case of the construction algorithm [91].

The encoding of the ternary equality constraint $x + y = z$ is given by the usual encoding of $x + y \leq z$ from Eq. 5.2 in conjunction with the following encoding on the analogous $x + y \geq z$:

$$\bigwedge_{v \in D(x), w \in D(y)} (\llbracket x \leq v \rrbracket \wedge \llbracket y \leq w \rrbracket) \rightarrow \llbracket z \leq v + w \rrbracket \tag{5.10}$$

The advantage of the equality encoding over splitting into two inequalities is that it reuses the same variables for the intermediate sums, rather than building two copies of the intermediate sums.

Example 5.9 Consider encoding the equation $10b_1 + 7b_2 + 5b_3 + 2b_4 + b_5 = 15$, using the same tree decomposition as in Ex. 5.2. The constraints are $x_2 + x_3 = y_1$, $x_1 + y_2 = y_2$, $x_4 + x_5 = y_3$, $y_2 + y_3 = y_4$, $y_4 = 15$, with the same domains as shown in Ex. 5.2. We encode each ternary equation as defined above, e.g. $y_2 + y_3 = y_4$ generates (trivial) clauses such as $\neg\llbracket y_2 \geq 5 \rrbracket \vee \neg\llbracket y_3 \geq 2 \rrbracket \vee \llbracket y_4 \geq 7 \rrbracket \equiv \texttt{true}$ from $(\llbracket y_2 \geq 5 \rrbracket \wedge \llbracket y_2 \geq 2 \rrbracket) \rightarrow \llbracket y_4 \geq 7 \rrbracket$ and more interesting clauses such as $\llbracket y_2 \geq 7 \rrbracket \vee \llbracket y_3 \geq 3 \rrbracket \vee \neg\llbracket y_4 \geq 8 \rrbracket$ from $(\llbracket y_2 \leq 5 \rrbracket \wedge \llbracket y_3 \leq 2 \rrbracket) \rightarrow \llbracket y_4 \leq 7 \rrbracket$, which is equivalent to just $\llbracket y_2 \geq 7 \rrbracket \vee \llbracket y_3 \geq 3 \rrbracket$.

We can prove results analogous to Thm. 5.4 and Thm. 5.5 about the encoding of $x + y \geq z$ generated by Eq. 5.10.

Theorem 5.6 The decomposition of a PB constraint $\sum_{i \equiv 1}^{n} q_i b_i \geq k$ into ternary inequalities encoded using Eq. 5.10 enforces domain consistency of the constraint.

The encodings from Eq. 5.2 and Eq. 5.10 of the decompositions of ternary $\leq$- and $\geq$-inequalities enforce bounds($\mathbb{R}$) consistency on $\sum_{i \equiv 1}^{n} q_i b_i \leq k$ and $\sum_{i \equiv 1}^{n} q_i b_i \geq k$, respectively, and consequently, together they also enforce bounds($\mathbb{R}$) consistency on $\sum_{i \equiv 1}^{n} q_i b_i = k$ [127].

Mixed Integer Variable Encodings of the Decompositions

By Thm. 5.5, order encoding the integer variables in a decomposition enforces domain consistency of the inequality constraint. However, the number of clauses of Eq. 5.2 grows cubically with the domains of the variables. This makes the encoding too large to be effective as domain sizes grow. Alternatively, a binary encoding of the integer variables and applying the RCA encoding on the constraints (see Sec. 4.3.4) generates fewer clauses, but trades in the consistency guarantees.

The appropriate constraint encoding in that case is the RCA encoding. However, the RCA encodes equality rather than inequality, which means it cannot be applied to the inequalities

in the decomposition without further modifications. For example, we can rewrite $x + y \leq z$ by introducing an auxiliary variable z'. Note how the domain of the new auxiliary variable z' is chosen so that no solutions are removed by this rewrite.

$$x + y \leq z \equiv x + y = z' \in [\mathrm{lb}(x) + \mathrm{lb}(y)..\mathrm{ub}(z)] \wedge z' \leq z$$

The encoding of this (further) decomposition of the inequality constraint requires one RCA and one lexicographic encoding. However, we can avoid z' and the lexicographic constraint if we use equality comparators for all intermediate constraints, and only bound $y_{\mathrm{root}} \leq k$. The binary encoding of the BDD and SWC decomposition is similar (apart from some domain changes) to the RCA, while GT has its alternative, balanced tree structure.

Depending on the chosen decomposition, the domains of the different auxiliary variables vary in size. In the SWC, the domains of (non-constant) auxiliary variables are of the same size. In the GT, the size of auxiliary domains grows at each subsequent layer. In the BDD decomposition, the auxiliary domain values directly correspond with the nodes at each layer of a given QOBDD. They start with one node at the root layer, grow in number until the middle layer, and then shrink again until the terminal layer. Since the decomposition is agnostic as to which variable encoding is used for which variable, we can choose the order encoding for domains up to size c, and binary for larger domains. We will denote this simple heuristic by $\mathbb{O} \geq c$.

Consider the equality constraint $x + y = z$ in a decomposition of an inequality constraint. Suppose we are given a pre-determined choice of order or binary encoding for each variable indicated by $\mathbb{O}$ or $\mathbb{B}$, respectively. So far, the encoding choices have always been uniform: $x{:}\mathbb{O} + y{:}\mathbb{O} = z{:}\mathbb{O}$, using Eq. 5.2 after relaxing to an inequality constraint, or $x{:}\mathbb{B} + y{:}\mathbb{B} = z{:}\mathbb{B}$, using the RCA as discussed at the start of this section. Unfortunately, for mixed encodings such as $x{:}\mathbb{O} + y{:}\mathbb{B} = z{:}\mathbb{B}$, often there are no (efficient) encodings. However, there are efficient encodings for $x{:}\mathbb{O} \# x{:}\mathbb{B}, \# \in \{\leq, \geq, =\}$ through coupling (the main topic of Chp. 5). To encode decompositions of equality constraints (see Sec. 5.2.3), we apply $x{:}\mathbb{O} = x{:}\mathbb{B}$. We can introduce $x{:}\mathbb{B}$ as an additional integer variable and decompose the constraint to $x{:}\mathbb{O} \leq x{:}\mathbb{B} \wedge x{:}\mathbb{B} + y{:}\mathbb{B} = y{:}\mathbb{B}$.

Example 5.10 Consider the PB constraint $8b_1 + 6b_2 + 3b_3 + 3b_4 + 2b_5 \leq 10$. After constructing a BDD, we decompose using Eq. 5.7 and order encode all variables with domain size up to 3 (i.e. $\mathbb{O} \leq 3$):

$$
\begin{aligned}
x_1{:}\mathbb{O} \trianglelefteq \{0, 8\} \qquad & \leq y_1{:}\mathbb{O} \trianglelefteq \{1, 8\} \\
x_2{:}\mathbb{O} \trianglelefteq \{0, 6\} + y_1{:}\mathbb{O} & \leq y_2{:}\mathbb{O} \trianglelefteq \{2, 7, 8\} \\
x_3{:}\mathbb{O} \trianglelefteq \{0, 3\} + y_2{:}\mathbb{O} & \leq y_3{:}\mathbb{B} \trianglelefteq \{5, 7, 8, 10\} \\
x_4{:}\mathbb{O} \trianglelefteq \{0, 3\} + y_3{:}\mathbb{B} & \leq y_4{:}\mathbb{O} \trianglelefteq \{8, 10\} \\
x_5{:}\mathbb{O} \trianglelefteq \{0, 2\} + y_4{:}\mathbb{O} & \leq 10
\end{aligned}
$$

This requires further decomposition of the two mixed encoding constraints:

$$
\begin{aligned}
x_3{:}\mathbb{O} \trianglelefteq \{0, 3\} &\leq x_3{:}\mathbb{B} \trianglelefteq \{0, 3\} \\
y_2{:}\mathbb{O} \trianglelefteq \{2, 7, 8\} &\leq y_2{:}\mathbb{B} \trianglelefteq \{2, 7, 8\} \\
x_3{:}\mathbb{B} + y_2{:}\mathbb{B} &= y_3{:}\mathbb{B} \trianglelefteq [2..10] \\
x_4{:}\mathbb{O} \trianglelefteq \{0, 3\} &\leq x_4{:}\mathbb{B} \trianglelefteq \{0, 3\} \\
x_4{:}\mathbb{B} + y_3{:}\mathbb{B} &= y_4{:}\mathbb{B} \trianglelefteq [2..10] \\
y_4{:}\mathbb{B} &\geq y_4{:}\mathbb{O} \trianglelefteq \{8, 10\}
\end{aligned}
$$

To support binary encoded integer decision variables (see Sec. 5.2.3), representing a term $x_i = q_i x_{i'}{:}\mathbb{B}$ is not without overhead (unlike $x_i = q_i x_{i'}{:}\mathbb{O}$). Here we reuse the pre-computed Single Constant Multiplication (SCM) encodings, which was the main topic of Chp. 4. By using common sub-expression elimination of terms, we avoid introducing the same encoding twice [128].

5.3 Experimental Evaluation

In this section, we evaluate the GT and BDD decomposition, and apply each practical extension. First, we test the PB encoding in its base form by converting each integer decision variable into PB terms using the order encoding. The next configuration (e.g. GT) supports the integer linear constraint directly (see Sec. 5.2.3). Next, we add direct support for equality constraints (e.g. GT + eq + $\mathbb{O}$) rather than splitting each in two inequalities (see Sec. 5.2.3). This is only applicable for the equality constraints tested in Sec. 5.3.2. Finally, we mix in the binary encoding for (auxiliary) integer variables (e.g. GT + $\mathbb{O} \leq c$) using two cut-offs (see Sec. 5.2.3), as well as a uniform binary approach (e.g. GT + $\mathbb{B}$). We omit the SWC decomposition because its constant size auxiliary variable domains are not as interesting for the mixed encoding.

As baseline, we include various configurations of three established SAT encoders which can encode integer linear constraints. Savile Row (`savile-row`) [103] (version 1.10.1) implements the fundamental PB encodings that we study in this paper, as well as binary encoding approaches such as Generalized n-Level Modulo Totalizer (GMTO) [60] and Local and Global Polynomial Watchdog (LPW/GPW) [63]. These PB encodings are generalized for `AMO` side-constraints, allowing for principal integer variables, the GT encoding is enhanced by a BDD-like reduction algorithm [107]. Equality constraints are separately encoded as two inequalities. Picat-SAT (`picat-sat`) [90] (version 3.5) primarily uses the binary encoding with RCA and ad-hoc optimization. Diet-Sugar/Fun-sCOP (`diet-sugar`) [101] (version `20230601-13h09m`) hybridizes order and binary encodings of the principal integer variables. Yet, its binary encoding employs PB encodings such as BDD, which still order encode the auxiliary variables uniformly. Consequently, there was no difference between their binary and mixed approach. We have omitted PBLib (`pblib`) [88], another common PB encoding library, as it does not support side constraints or integer variables, and we have found that it is not competitive in an integer setting.

All encodings are solved using the same executable of `CaDiCal` [125] (v2.1.0) with a 3 GB memory limit and a time limit of 60 seconds. For two different IL problems, we generate three instance sets: one with 01-bounds for the integers (PB), others with larger domains. Our implementation of the encoding methods, the generated benchmark instances, and the benchmark scripts are available [129].

5.3.1 Multidimensional Bounded Knapsack Problem

The Multidimensional Bounded Knapsack Problem (MBKP) instances are generated as discussed in Sec. 4.4.2.

In the results shown in Tab. 5.1, the effects of using integer decision variables ($+\,\mathbb{O}$) is difficult to compare due to encoding memory- and timeouts. Using the uniform binary encoding ($+\,\mathbb{B}$) improves results in every case, presumably because the auxiliary domains are of considerable size. A relatively small cut-off value, with only a minority of all domain values order encoded, improves the number of solved instances for both GT and BDD on all instance set, with the former outperforming the control encoders. In experiments not shown here, a larger cut-off value of 150 or 300 was found to be less effective.

5.3.2 Multidimensional Bounded Subset Sum Problem

The Multidimensional Bounded Subset Sum Problem (MBSSP) is similar to MBKP, but requires packed items to sum up to an exact capacity using equality constraints. We decide for item types $1 \leq i \leq N$ how many x_i items to select (up to bound B) such that an exact subset sum k_i is reached for each $1 \leq j \leq M$ dimension given the weight $q_{i,j} \in [1..Q]$ of item j in dimension i. I.e. $\bigwedge_{i\equiv 1}^{M} \sum_{j\equiv 1}^{N} q_{i,j}x_j \equiv k_i$. Instances are generated by uniformly sampling $q_{i,j}$ from $[0..Q]$ for $1 \leq i \leq M, 1 \leq j \leq N$. Then, to guarantee a solution exists, we uniformly sample an assignment a_j for every x_j, yielding feasible values for $k_i \equiv \sum_{j\equiv 1}^{N} q_{i,j}a_j$ for $1 \leq i \leq M$. The experiments are run with a memory and solver time limit of 6 GB and 180 s.

The results in Tab. 5.2 show an improvement when using integer variables ($+\,\mathbb{O}$) on non-PB instances. The support for equality constraints predictably halves the number of auxiliary variables, and makes a positive effect on solver performance overall. Again, the binary encoding approach is better than the order-based ones, and equals the best control for two out of three instance sets. Mixing in the order encoding does not improve performance for MBSSP, as it perhaps suits the equality constraints less.

(N, B, M, Q)	$(50, 1, 25, 50)$		$(10, 10, 200, 50)$		$(10, 10, 200, 250)$	
	solved	vars./cl.	solved	vars./cl.	solved	vars./cl.
GT + $\mathbb{B}$	84 (6.0)	15/76	85 (11.5)	36/224	74 (16.5)	52/332
GT + $\mathbb{O} \leq 25$	88 (5.7)	14/54	**94 (10.9)**	39/238	87 (14.9)	58/337
GT + $\mathbb{O} \leq 75$	**89 (5.7)**	14/54	93 (11.8)	43/252	**91 (16.6)**	58/340
GT + $\mathbb{O}$	67 (3.5)	84/7852	0	701/159425	—	—
GT	68 (3.9)	84/7852	0	985/164102	—	—
BDD + $\mathbb{B}$	78 (3.3)	23/99	78 (10.1)	40/234	68 (12.0)	56/343
BDD + $\mathbb{O} \leq 25$	75 (8.9)	26/113	93 (13.4)	42/244	86 (15.1)	58/334
BDD + $\mathbb{O} \leq 75$	76 (10.2)	29/114	90 (13.8)	43/252	88 (16.1)	58/335
BDD + $\mathbb{O}$	75 (6.4)	351/694	61 (17.4)	775/7983	0	2723/28503
BDD	74 (5.2)	351/694	0	8313/16588	0	30312/60561
`savile-row` (GT)	82 (2.3)	61/5946	0	610/108981[56]	—	—
`savile-row` (SWC)	74 (13.9)	795/1561	9 (13.9)	2244/22459	0	10566/105809[30]
`savile-row` (BDD)	86 (0.8)	230/446	44 (2.8)	582/5535	40 (2.2)	2019/19390
`savile-row` (GPW)	87 (3.5)	22/117	74 (12.1)	83/288	72 (13.1)	111/394
`savile-row` (GMTO)	87 (5.1)	20/43	89 (8.0)	50/179	84 (10.9)	63/234
`diet-sugar`	83 (0.2)	232/667	58 (8.5)	1252/3512	44 (9.0)	5405/15092
`picat-sat`	81 (6.5)	10/62	70 (16.7)	27/211	47 (19.6)	42/359

Tab. 5.1: Encoding size and solve results for 3 sets of 100 Multidimensional Bounded Knapsack Problem (MBKP) instances ($C = 4, S = 25, 0.4 \leq f \leq 0.6$). The solve result is the number of instances solved followed by their average solve time. The encoding size is the number of thousands of variables and clauses, with the number of memory- and timeouts (if any) indicated by a superscript, or — if all failed to encode.

(N, B, M, Q)	$(50, 1, 25, 50)$		$(10, 10, 200, 50)$		$(10, 10, 200, 250)$	
	solved	vars./cl.	solved	vars./cl.	solved	vars./cl.
GT + eq + $\mathbb{B}$	100 (2.5)	7/34	100 (1.2)	2/14	**100 (1.4)**	3/19
GT + eq + $\mathbb{O} \leq 25$	100 (2.4)	6/55	100 (7.0)	2/58	99 (10.3)	3/253
GT + eq + $\mathbb{O} \leq 75$	100 (2.3)	6/55	100 (9.4)	3/59	100 (11.7)	3/253
GT + eq + $\mathbb{O}$	6 (36.8)	32/5794	0	38/22574	0	$103/177961^{96}$
GT + $\mathbb{O}$	4 (46.4)	68/6446	0	81/23596	0	$251/207563^{99}$
GT	2 (37.9)	68/6446	0	101/25768	0	$291/196827^{97}$
BDD + eq + $\mathbb{B}$	99 (9.8)	10/44	**100 (1.1)**	2/14	**100 (1.4)**	3/20
BDD + eq + $\mathbb{O} \leq 25$	—	—	100 (6.1)	3/59	100 (10.2)	3/254
BDD + eq + $\mathbb{O} \leq 75$	—	—	100 (8.1)	3/60	100 (10.1)	3/254
BDD + eq + $\mathbb{O}$	35 (35.7)	116/458	7 (42.6)	49/1024	0	192/4045
BDD + $\mathbb{B}$	29 (35.3)	232/458	5 (34.3)	99/1024	1 (17.3)	385/4048
BDD	32 (36.3)	232/458	0	1037/2068	0	4117/8225
`savile-row` (GT)	**100 (0.2)**	47/3190	0	67/15607	—	—
`savile-row` (SWC)	33 (48.3)	597/1166	1 (24.7)	269/2728	0	1322/13412
`savile-row` (BDD)	34 (28.0)	153/294	0	75/725	0	289/2816
`savile-row` (GPW)	98 (12.4)	19/89	15 (25.6)	9/31	9 (25.2)	11/42
`savile-row` (GMTO)	94 (13.7)	18/39	11 (31.7)	5/18	8 (32.1)	6/23
`diet-sugar`	26 (38.3)	156/442	5 (25.6)	155/435	1 (46.7)	688/1930
`picat-sat`	81 (16.1)	5/29	100 (1.6)	2/13	100 (1.8)	2/19

Tab. 5.2: Encoding size and solve results for 3 sets of 100 Multidimensional Bounded Subset Sum Problem (MBSSP) instances. The solve result is the number of instances solved followed by their average solve time. The encoding size is the number of thousands of variables and clauses, with the number of memory- and timeouts (if any) indicated by a superscript, or — if all failed to encode.

5.4 Conclusion and future work

In this chapter, we have shown that by thinking of encodings of PB constraints as computing partial sums, we can reframe the fundamental GT, SWC, and BDD encodings as integer decompositions. But once we have the *integer viewpoint*, any improvement on the decomposition automatically improves each encoding method. We extend the decomposition by supporting linear constraints, equality constraints, and mixed encodings for each individual integer variable. We show how each extension practically improves each method across two types of benchmarks and compared to three strong baseline encoders.

However, we note that there are various aspects to be improved in the experimental evaluation to show that the observed effect holds in practce. A better evaluation would test multiple SAT solvers over multiple seeds, as well as use more established non-generated benchmarks sets, such as those from the PB competition.

In future work, additional PB encodings may be understood from the same viewpoint. Some methods (e.g. Sorting Network (SN) [58]) may also use an implicit order encoding for their auxiliary variables, while others may use the binary or direct encoding (e.g. alternative BDD encodings [110]), or the binary encoding such as in the LPW/GPW. Further extensions of the decomposition (e.g. by improving its encoding) could also be of interest. In the experiments, `savile-row`'s implementation of GMTO has a strong performance. Capturing this sophisticated encoding in the integer viewpoint will also require the integration of mixed-radix integer variable encodings. In future work, we also wish to structurally investigate novel PB encodings which mix variable encodings within the same decomposition.

— Chapter 6 —

Conclusion

This thesis has aimed to improve the theoretical understanding and practical effectiveness of Boolean Satisfiability (SAT) encoding methods. The three technical chapters reflect the paths we have gone down to this end, namely a) the coupling of different integer variable encodings, b) the encoding of the Single Constant Multiplication (SCM) constraint on binary encoded variables, and c) the understanding and extension of Pseudo-Boolean (PB) encodings through the lens of Constraint Satisfaction Problem (CSP) decompositions. This concluding chapter summarizes the contribution towards each aim and discusses potential future work.

6.1 Coupling integer variable encodings

Existing approaches to encoding CSP to SAT often fix the integer variable encoding. This avoids the complexity of *coupling*, i.e. encoding a constraint with an arbitrary mix of encodings for its integer variables. However, different constraints prefer different variable encodings. If such constraints appear together in complex problems, we miss out on potential performance improvements.

The first main contribution of the thesis that addresses this challenge are the various novel coupling encodings which are proposed, implemented as part of the MiniZinc-SAT (`mzn-sat`) framework and evaluated theoretically and experimentally. The coupling encodings cover three fundamental integer encodings, namely the direct, order and binary encoding. We can couple equality constraints to channel integer encodings, such as order and binary, which is very flexible, although it introduces a redundant encoding. Non-redundant encodings can be made by coupling directly, such as linear constraints, especially binary inequality constraints or element constraints. Some of these encodings are also expressed as decompositions, since they rely on high-level yet simple constraints such as unary equality, inequality, bit representation, or set membership constraints. The availability of total and partial views on certain encodings and constraints can point an expert modeller to the right direction for the most effective SAT encoding for a given problem. In five case studies, we find a use case for most coupling encodings on realistic benchmark problems.

Future work

We find the following directions of future work of special interest:

- For some constraints and integer encodings we have no efficient approach to coupling. Consequently, we need to resort to either channelling or inefficient coupling techniques (e.g. representing the constraint as a linear).
- The `mzn-sat` framework is currently expert-guided and assumes the encoding specifications (or annotations) are all known. Heuristics to guide this could a) improve the accessibility of the framework by automatically suggesting better defaults, and b) allow a wider experimental evaluation. This could expand the work on existing heuristics approaches, either those computed directly like in `PBSugar` [96] or based on Machine Learning (ML) [66].

6.2 SCM for SAT

Chp. 4 studies how to best encode the SCM constraint for binary encoded variables. This can then be applied to encoding Integer Linear (IL) constraints, which are important to encode well due to their ubiquity and the complexity they add to problems. In the hardware design community, circuits for the same constraint are found by solving SCM problem, often minimizing the number of additions and subtractions. More recently, the complexity of the additions/subtractions is also taken into account by minimizing the number of adders in the additions/subtractions [116].

The second main contribution of the thesis adapts SCM techniques from hardware to software (i.e. SAT solvers). Since solving SCM is NP-hard, it cannot be done at encode time without likely losing benefits at solve time. To avoid this issue, we pre-compute databases of SCM circuits using MiniZinc which cover a realistic range of SCM constants. Another approach is to use Boolean logic minimizers, which solve SCM for SAT without any additional variables, but this comes at the cost of an exponential number of clauses/literals. These databases are complete and optimal for the chosen range.

An experimental evaluation on generated benchmarks shows that for variables with large domains, minimizing the number of additions/subtractions is helpful over a pure baseline, or one with ad-hoc optimizations implemented by Picat-SAT (`picat-sat`). Minimizing the total complexity of additions/subtractions further improves SAT solver performance. For smaller domains, however, the Boolean logic minimization approach is best, despite requiring a larger number of literals than the other approaches.

Future work

We see various avenues for future work:

- A natural extension would be to tackle Multiple Constant Multiplication (MCM), which can reuse parts of the single optimal circuit for all coefficients of an integer variables which appears in multiple IL constraint. However, this cannot be easily pre-computed, so a trade-off between encoding and solve times bring another challenge.

- Experiments show that minimizing adders (literals) using the SCM MiniZinc model is good for small domains (and by extension, coefficients), while minimizing literals using the Boolean logic minimizer is good for small domains and coefficients. For problems where both types of domains/coefficients occur, a mix of approaches should apply. However, we recognize that the SCM approach decomposes its problem over a large domain and coefficient into many sub-problems of smaller domains and coefficients. By switching to Boolean minimization as domains and coefficients are reduced lower down the tree, we could see improvements across the upper half of the parameter space of the benchmarks.
- Pre-computing SAT encodings for other constraints by Boolean logic minimization or by more specific techniques such as SCM could be practically applied.

6.3 SAT encodings as CSP decompositions

Once again, PB constraints (a special kind of IL constraints) are among the most well-studied encodings to SAT. Various approaches are possible, fundamental of which are the Generalized Totalizer (GT), Sequential Weight Counter (SWC) and Binary Decision Diagram (BDD) encoding. The literature specifies each approach using a distinct abstraction, which has some distinct encoding to SAT. However, there appears to be functional overlap between the abstractions and their encodings.

The third main contribution of the thesis is a decomposition approach to encoding PB constraints, the equivalence of the encoding of particular decompositions to the original PB encodings, and extensions of the decomposition method which improve the base methods in six ways. On one hand, the extensions complete a picture of the original and additional literature, such as unified proofs of propagation strength, or support for side-constraints and equality comparators. On the other hand, it also allows for novel encodings. Furthermore, coupling constraints and mixed encoding find a place not only within different constraints of the same CSP, but also within a decomposition of a single constraint.

Practically, the debugging process is also improved. For small instances (e.g. in unit tests), we can check the correctness of a particular decomposition or encoding by comparing its set of solutions with the set of solutions of the original constraint. First, an extra or missing solution in the decomposition is caught before it is encoded to SAT. Then, an extra or missing solution in the encoding is isolated to the encoding of a ternary inequality constraint rather than the full IL constraint. In this sense, we see the decomposition method also as a design philosophy, which uses Constraint Programming (CP) to reason about the complex integer constraints, while focusing the complexity of SAT encoding on simpler constraints in a highly optimized way.

The decomposition method also allows us to understand our own work better. In Chp. 2, we needed to formalize a working definition of encodings (see Def. 2.11) to apply to techniques

from the literature and this thesis. We found that the central requirement for one CSP encoding another is that the encoding CSP must be interpretable, i.e. allow a mapping from its solutions back to the solutions of the original. However, this definition not only describes encodings to SAT, but also encodings to other types of CSPs. These non-SAT encodings are the CSP decompositions in question and tie in with CP (e.g. the decompositions in MiniZinc's global constraint library).

In Chp. 3, we have retroactively rewritten some of the coupling encodings as decompositions as well. For instance, the order-binary channelling has its origin in my MSc. thesis [85], using an abstraction called *segments* of the binary representation, later rebranded as *stable ranges* in [18]. In this thesis, we recognize that both integer variables in the coupling encoding of $E(x{:}\mathbb{O} = y{:}\mathbb{B})$ rely on the bit-representation property, i.e. the unary integer constraint, bit. For the binary encoded variable, $E(\mathrm{bit}(y{:}\mathbb{B}, k))$ is encoded by a single literal, while for the order encoded variable, $E(\mathrm{bit}(x{:}\mathbb{O}, k))$ is encoded by a chain of decompositions: set membership constraints of range membership constraints of inequality constraints. The final decomposition in the chain is trivial to encode, being the order encoding of an inequality constraint $E(x{:}\mathbb{O} \geq d)$ as $[\![x \geq d]\!]$. In Chp. 4, the solution of the SCM model is also a decomposition of ternary integer equality constraints. Through a complex optimization process which involves its own MiniZinc model solved by SAT, the decomposition is computed to favour the binary encoding with its power-of-two coefficients using affine views.

Future work

We see various avenues for future work:

- Additional encoding methods of IL constraints and its specializations (PB, cardinality, `AMO` s) might permit decompositions as well. A complex example is the Sorting Network (SN), while a trivial example is the *ladder* encoding of the `AMO` constraint discussed in Sec. 2.3.5. Some might require integer variable encodings other than order for its decomposition, such as binary for the Ripple Carry Adder (RCA) and direct for the `CompletePath` encoding (as it requires an `EO` constraint) [110]. Fundamentally, any type of integer constraint, presumably outside the realm of IL constraints as well, cannot avoid representing auxiliary integer variables by related sets of literals.
- Rather than reframing existing encodings as decompositions, we can look to create new decompositions which combine the lessons from the existing ones.

Glossary

***AC* – Arc Consistency** 21

***AI* – Artificial Intelligence** vi

`ALO` **– At-Least-One** 30, 44, 52

`AMO` **– At-Most-One** 11, 30, 43, 44, 52, 58, 70, 71, 73, 109, 125, 126, 130, 138

***ASIC* – Application-Specific Integrated Circuit** 90

bounds(ℝ) consistency 20, 21, 124, 126, 127

***BDD* – Binary Decision Diagram** 44, 77, 107, 108, 109, 111, 112, 118, 119, 120, 121, 122, 125, 126, 128, 129, 130, 131, 134, 137

`BEE` 60, 61, 80

***BnB* – Brand and Bound** 21, 25

***BT* – backtracking** 19, 20, 21, 23, 25

`CaDiCal` 48, 49, 104, 130

***CDCL* – Conflict-driven Clause Learning** 21, 25

`chuffed` **– Chuffed** 9, 60, 82, 83, 84, 87

***CNF* – Conjunctive Normal Form** 21, 22, 26, 27, 30, 94, 96, 101, 103

***CO* – Combinatorial Optimization** ii, 1, 2, 5, 8, 9, 11, 12, 15, 16

***COP* – Constrained Optimization Problem** 17, 18, 21, 22, 25, 54, 84, 94, 97, 101

***CP* – Constraint Programming** ii, 2, 4, 6, 7, 8, 9, 11, 12, 13, 15, 19, 23, 45, 46, 51, 54, 55, 59, 66, 79, 80, 92, 137, 138

***CSD* – Canonical Signed Digit** 91

***CSE* – Common Subexpression Elimination** 53, 106

***CSP* – Constraint Satisfaction Problem** 15, 16, 17, 18, 19, 20, 21, 22, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 41, 42, 43, 44, 45, 46, 48, 50, 51, 54, 55, 58, 60, 63, 65, 81, 107, 108, 109, 110, 135, 137, 138

***DBNS* – double-base number system** 91

`diet-sugar` **– Diet-Sugar/Fun-sCOP** 60, 77, 79, 130, 132, 133

***DNF* – Disjunctive Normal Form** 22, 68

***DP* – Davis-Putnam** 23, 25

***DPLL* – Davis-Putnam-Logemann-Loveland** 23, 24, 25

`EO` **– Exactly-One** 30, 33, 44, 71

`espresso` **– Espresso** 91, 96, 97, 103

***FPGA* – Field-Programmable Gate Array** 90

`FznTini` 58, 60

***GAC* – Generalized Arc Consistency** 21

***SCM* – Single Constant Multiplication** 12, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 102, 103, 104, 105, 106, 129, 135, 136, 137, 138

***SMT* – Satisfiability Modulo Theories** 55

***SN* – Sorting Network** 44, 134, 138

`Sp-Or` 60

***SWC* – Sequential Weight Counter** 44, 107, 108, 109, 111, 112, 115, 116, 117, 118, 122, 125, 126, 128, 130, 134, 137

***UP* – Unit Propagation** 23, 24, 59, 60, 62, 63, 68, 69, 70, 71, 72, 73, 74, 76, 78, 114, 117, 118, 121, 123

Bibliography

[1] Wikipedia, "Instructions per second — Wikipedia, the Free Encyclopedia" (accessed November 2024). [Online]. Available at: http://en.wikipedia.org/w/index.php?title=Instructions%20per%20second&oldid=1244641293

[2] L. J. Stockmeyer, "Planar 3-Colorability Is Polynomial Complete," *SIGACT News*, vol. 5, no. 3, pp. 19–25, 1973, doi: 10.1145/1008293.1008294.

[3] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "MiniZinc: Towards a Standard CP Modelling Language," in *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, C. Bessiere, Ed., in Lecture Notes in Computer Science, vol. 4741. Springer, 2007, pp. 529–543. doi: 10.1007/978-3-540-74970-7_38.

[4] S. A. Cook, "The Complexity of Theorem-Proving Procedures," in *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, M. A. Harrison, R. B. Banerji, and J. D. Ullman, Eds., ACM, 1971, pp. 151–158. doi: 10.1145/800157.805047.

[5] P. J. Stuckey, T. Feydy, A. Schutt, G. Tack, and J. Fischer, "The MiniZinc Challenge 2008-2013," *Ai Magazine*, vol. 35, no. 2, pp. 55–60, 2014, doi: 10.1609/AIMAG.V35I2.2539.

[6] P. J. Stuckey, R. Becket, and J. Fischer, "Philosophy of the MiniZinc Challenge," *Constraints An Int. J.*, vol. 15, no. 3, pp. 307–316, 2010, doi: 10.1007/S10601-010-9093-0.

[7] L. Perron and F. Didier, "CP-SAT." [Online]. Available at: https://developers.google.com/optimization/cp/cp_solver/

[8] J. K. Fichte, M. Hecher, and F. Hamiti, "The Model Counting Competition 2020," *ACM Journal of Experimental Algorithmics*, vol. 26, Oct. 2021, doi: 10.1145/3459080.

[9] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, 2nd ed. IOS Press, 2021.

[10] A. Biere, M. Fleury, N. Froleyks, and M. J. H. Heule, "The SAT Museum," in *Proceedings of the 14th Internantional Workshop on Pragmatics of SAT Co-Located with the 26th International Conference on Theory and Applicationas of Satisfiability Testing (SAT 2003), Alghero, Italy, July, 4, 2023*, M. Järvisalo and D. Le Berre, Eds., in CEUR Workshop Proceedings, vol. 3545. CEUR-WS.org, 2023, pp. 72–87. [Online]. Available at: http://ceur-ws.org/Vol-3545/paper6.pdf

[11] P. J. Heawood, "Map-Colour Theorem," *Proceedings of The London Mathematical Society*, pp. 161–175, 1949, [Online]. Available at: https://api.semanticscholar.org/CorpusID:121562895

[12] K. Appel and W. Haken, "Every Planar Map Is Four Colorable. Part I: Discharging," *Illinois Journal of Mathematics*, vol. 21, pp. 429–490, 1977, [Online]. Available at: https://api.semanticscholar.org/CorpusID:118172389

[13] K. Appel, W. Haken, and J. Koch, "Every Planar Map Is Four Colorable. Part II: Reducibility," *Illinois Journal of Mathematics*, vol. 21, pp. 491–567, 1977, [Online]. Available at: https://api.semanticscholar.org/CorpusID:117990672

[14] B. Subercaseaux and M. J. H. Heule, "The Packing Chromatic Number of the Infinite Square Grid Is 15," in *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part I*, S. Sankaranarayanan and N. Sharygina, Eds., in Lecture Notes in Computer Science, vol. 13993. Springer, 2023, pp. 389–406. doi: 10.1007/978-3-031-30823-9_20.

[15] A. V. Gelder, "Another Look at Graph Coloring via Propositional Satisfiability," *Discret. Appl. Math.*, vol. 156, no. 2, pp. 230–243, 2008, doi: 10.1016/J.DAM.2006.07.016.

[16] V.-H. Nguyen and S. T. Mai, "A New Method to Encode the At-Most-One Constraint into SAT," in *Proceedings of the Sixth International Symposium on Information and Communication Technology, Hue City, Vietnam, December 3-4, 2015*, H. Q. Thang, L. A. Phuong, L. D. Raedt, Y. Deville, M. Bui, T. T. D. Linh, N. Thi-Oanh, D. V. Sang, and N. B. Ngoc, Eds., ACM, 2015, pp. 46–53. doi: 10.1145/2833258.2833293.

[17] P. Cappello and K. Steiglitz, "Some Complexity Issues in Digital Signal Processing," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 32, no. 5, pp. 1037–1041, 1984.

[18] H. Bierlee, G. Gange, G. Tack, J. J. Dekker, and P. J. Stuckey, "Coupling Different Integer Encodings for SAT," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 19th International Conference, CPAIOR 2022, Los Angeles, CA, USA, June 20-23, 2022, Proceedings*, P. Schaus, Ed., in Lecture Notes in Computer Science, vol. 13292. Springer, 2022, pp. 44–63. doi: 10.1007/978-3-031-08011-1_5.

[19] H. Bierlee, J. J. Dekker, V. Lagoon, P. J. Stuckey, and G. Tack, "Single Constant Multiplication for SAT," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 21st International Conference, CPAIOR 2024, Uppsala, Sweden, May 28-31, 2024, Proceedings, Part I*, B. Dilkina, Ed., in Lecture Notes in Computer Science, vol. 14742. Springer, 2024, pp. 84–98. doi: 10.1007/978-3-031-60597-0_6.

[20] F. Rossi, P. van Beek, and T. Walsh, Eds., *Handbook of Constraint Programming*, vol. 2. in Foundations of Artificial Intelligence, vol. 2. Elsevier, 2006. [Online]. Available at: https://www.sciencedirect.com/science/bookseries/15746526/2

[21] A. Ek, "High-Level Modelling and Solving for Online, Real-Time, and Multiagent Combinatorial Optimisation" (accessed November 2024) Ph.D. dissertation, Monash University, Melbourne, 2022.

[22] F. Ulrich-Oltean, "Learning SAT Encodings for Constraint Satisfaction Problems" (accessed November 2024) Ph.D. dissertation, University of York, York, 2023.

[23] T. Soh, "Studies on Applying Incremental Sat Solving to Optimization and Enumeration Problems" (accessed November 2024) Ph.D. dissertation, Graduate University for Advanced Studies (Sokendai), Hayama, 2011.

[24] J. J. Dekker, "A Modern Architecture for Constraint Modelling Languages" (accessed November 2024) Ph.D. dissertation, Monash University, Melbourne, 2021.

[25] P. van Beek, "Backtracking Search Algorithms," *Handbook of Constraint Programming*, vol. 2. in Foundations of Artificial Intelligence, vol. 2. Elsevier, pp. 85–134, 2006. doi: 10.1016/S1574-6526(06)80008-8.

[26] A. K. Mackworth, "Consistency in Networks of Relations," *Artificial Intelligence*, vol. 8, no. 1, pp. 99–118, 1977, doi: 10.1016/0004-3702(77)90007-8.

[27] A. H. Land and A. G. Doig, "An Automatic Method for Solving Discrete Programming Problems," *50 Years of Integer Programming 1958-2008 - from the Early Years to the State-of-the-Art*. Springer, pp. 105–132, 2010. [Online]. Available at: https://doi.org/10.1007/978-3-540-68279-0_5

[28] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *Journal of The Acm*, vol. 7, no. 3, pp. 201–215, 1960, doi: 10.1145/321033.321034.

[29] M. Davis, G. Logemann, and D. W. Loveland, "A Machine Program for Theorem-Proving," *Communications of The Acm*, vol. 5, no. 7, pp. 394–397, 1962, doi: 10.1145/368273.368557.

[30] R. M. Stallman and G. J. Sussman, "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis," *Artificial Intelligence*, vol. 9, no. 2, pp. 135–196, 1977, doi: 10.1016/0004-3702(77)90029-7.

[31] R. Dechter, "Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition," *Artificial Intelligence*, vol. 41, no. 3, pp. 273–312, 1990, doi: 10.1016/0004-3702(90)90046-3.

[32] K. A. Sakallah and J. P. Marques-Silva, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Trans. Computers*, vol. 48, no. 5, pp. 506–521, 1999, doi: 10.1109/12.769433.

[33] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "Generic ILP versus Specialized 0-1 ILP: An Update," in *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2002, San Jose, California, USA, November 10-14, 2002*, L. T. Pileggi and A. Kuehlmann, Eds., ACM / IEEE Computer Society, 2002, pp. 450–457. doi: 10.1145/774572.774638.

[34] N. Eén and N. Sörensson, "An Extensible SAT-solver," in *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, E. Giunchiglia and A. Tacchella, Eds., in Lecture Notes in Computer Science, vol. 2919. Springer, 2003, pp. 502–518. doi: 10.1007/978-3-540-24605-3_37.

[35] Z. Fu and S. Malik, "On Solving the Partial MAX-SAT Problem," in *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, A. Biere and C. P. Gomes, Eds., in Lecture Notes in Computer Science, vol. 4121. Springer, 2006, pp. 252–265. doi: 10.1007/11814948_25.

[36] C. R. Codel, J. Avigad, and M. J. H. Heule, "Verified Encodings for SAT Solvers," in *Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023*, A. Nadel and K. Y. Rozier, Eds., IEEE, 2023, pp. 141–151. doi: 10.34727/2023/ISBN.978-3-85448-060-0_22.

[37] C. R. Codel, "Verifying SAT Encodings in Lean" (accessed November 2024) M.S. Thesis, Carnegie Mellon University, Pittsburgh, PA, 2022.

[38] G. S. Tseitin, "On the Complexity of Derivation in Propositional Calculus," *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pp. 466–483, 1983.

[39] D. A. Plaisted and S. Greenbaum, "A Structure-preserving Clause Form Translation," *Journal of Symbolic Computation*, vol. 2, no. 3, pp. 293–304, Sep. 1986, doi: 10.1016/S0747-7171(86)80028-1.

[40] T. Walsh, "SAT v CSP," in *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, R. Dechter, Ed., in Lecture Notes in Computer Science, vol. 1894. Springer, 2000, pp. 441–456. doi: 10.1007/3-540-45349-0_32.

[41] C. Ansótegui and F. Manyà, "Mapping Problems With Finite-Domain Variables to Problems with Boolean Variables," in *Theory and Applications of Satisfiability Testing*, H. H. Hoos and D. G. Mitchell, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1–15.

[42] J. M. Crawford and A. B. Baker, "Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems," in *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 2*, B. Hayes-Roth and R. E. Korf, Eds., AAAI Press / The MIT Press, 1994, pp. 1092–1097. [Online]. Available at: http://www.aaai.org/Library/AAAI/1994/aaai94-168.php

[43] T. Tanjo, N. Tamura, and M. Banbara, "A Compact and Efficient SAT-encoding of Finite Domain CSP," in *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, K. A. Sakallah and L. Simon, Eds., in Lecture Notes in Computer Science, vol. 6695. Springer, 2011, pp. 375–376. doi: 10.1007/978-3-642-21581-0_36.

[44] K. Iwama and S. Miyazaki, "SAT-variable Complexity of Hard Combinatorial Problems," in *Technology and Foundations - Information Processing '94, Volume 1, Proceedings of the IFIP 13th World Computer Congress, Hamburg, Germany, 28 August - 2 September, 1994*, B. Pehrson and I. Simon, Eds., in IFIP Transactions. North-Holland, 1994, pp. 253–258.

[45] M. D. Ernst, T. D. Millstein, and D. S. Weld, "Automatic SAT-compilation of Planning Problems," in *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, Morgan Kaufmann, 1997, pp. 1169–1177. [Online]. Available at: http://ijcai.org/Proceedings/97-2/Papers/055.pdf

[46] I. P. Gent, "Arc Consistency in SAT," in *Proceedings of the 15th Eureopean Conference on Artificial Intelligence, ECAI'2002, Lyon, France, July 2002*, F. van Harmelen, Ed., IOS Press, 2002, pp. 121–125.

[47] S. Kasif, "On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks," *Artificial Intelligence*, vol. 45, no. 3, pp. 275–286, 1990, doi: 10.1016/0004-3702(90)90009-O.

[48] M. N. Velev, "Exploiting Hierarchy and Structure to Efficiently Solve Graph Coloring as SAT," in *2007 International Conference on Computer-Aided Design, ICCAD 2007, San Jose, CA, USA, November 5-8, 2007*, G. G. E. Gielen, Ed., IEEE Computer Society, 2007, pp. 135–142. doi: 10.1109/ICCAD.2007.4397256.

[49] V.-H. Nguyen, M. N. Velev, and P. Barahona, "Application of Hierarchical Hybrid Encodings to Efficient Translation of CSPs to SAT," in *25th IEEE International*

Conference on Tools with Artificial Intelligence, ICTAI 2013, Herndon, VA, USA, November 4-6, 2013, IEEE Computer Society, 2013, pp. 1028–1035. doi: 10.1109/ICTAI.2013.154.

[50] P. Barahona, S. Hölldobler, and V.-H. Nguyen, "Representative Encodings to Translate Finite CSPs into SAT," in *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings*, H. Simonis, Ed., in Lecture Notes in Computer Science, vol. 8451. Springer, 2014, pp. 251–267. doi: 10.1007/978-3-319-07046-9_18.

[51] M. Gavanelli, "The Log-Support Encoding of CSP into SAT," in *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, C. Bessiere, Ed., in Lecture Notes in Computer Science, vol. 4741. Springer, 2007, pp. 815–822. doi: 10.1007/978-3-540-74970-7_59.

[52] A. M. Frisch and T. J. Peugniez, "Solving Non-Boolean Satisfiability Problems with Stochastic Local Search," in *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, B. Nebel, Ed., Morgan Kaufmann, 2001, pp. 282–290.

[53] S. D. Prestwich, "Full Dynamic Substitutability by SAT Encoding," in *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, M. Wallace, Ed., in Lecture Notes in Computer Science, vol. 3258. Springer, 2004, pp. 512–526. doi: 10.1007/978-3-540-30201-8_38.

[54] I. P. Gent and P. Nightingale, "A New Encoding of AllDifferent into SAT," in *International Workshop on Modelling and Reformulating Constraint Satisfaction*, 2004, pp. 95–110.

[55] S. D. Prestwich, "Finding Large Cliques Using SAT Local Search," *Trends in Constraint Programming*. John Wiley & Sons, Ltd, pp. 269–274, 2007.

[56] N.-F. Zhou, "Yet Another Comparison of SAT Encodings for the At-Most-k Constraint," *CoRR*, 2020, [Online]. Available at: https://arxiv.org/abs/2005.06274

[57] V.-H. Nguyen, V.-Q. Nguyen, K. Kim, and P. Barahona, "Empirical Study on SAT-encodings of the at-Most-One Constraint," in *SMA 2020: The 9th International Conference on Smart Media and Applications, Jeju, Republic of Korea, September 17 - 19, 2020*, ACM, 2020, pp. 470–475. doi: 10.1145/3426020.3426170.

[58] N. Eén and N. Sörensson, "Translating Pseudo-Boolean Constraints into SAT," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, no. 1–4, pp. 1–26, 2006, doi: 10.3233/sat190014.

[59] S. Joshi, R. Martins, and V. M. Manquinho, "Generalized Totalizer Encoding for Pseudo-Boolean Constraints," in *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, G. Pesant, Ed., in Lecture Notes in Computer Science, vol. 9255. Springer, 2015, pp. 200–209. doi: 10.1007/978-3-319-23219-5_15.

[60] A. Zha, M. Koshimura, and H. Fujita, "N-Level Modulo-Based CNF Encodings of Pseudo-Boolean Constraints for MaxSAT," *Constraints*, vol. 24, no. 2, pp. 133–161, Apr. 2019, doi: 10.1007/s10601-018-9299-0.

[61] S. Hölldobler, N. Manthey, and P. Steinke, "A Compact Encoding of Pseudo-Boolean Constraints into SAT," in *KI 2012: Advances in Artificial Intelligence - 35th Annual German Conference on AI, Saarbrücken, Germany, September 24-27, 2012. Proceedings*, B. Glimm and A. Krüger, Eds., in Lecture Notes in Computer Science, vol. 7526. Springer, 2012, pp. 107–118. doi: 10.1007/978-3-642-33347-7_10.

[62] J. P. Warners, "A Linear-Time Transformation of Linear Inequalities into Conjunctive Normal Form," *Inf. Process. Lett.*, vol. 68, no. 2, pp. 63–69, 1998, doi: 10.1016/S0020-0190(98)00144-6.

[63] O. Bailleux, Y. Boufkhad, and O. Roussel, "New Encodings of Pseudo-Boolean Constraints into CNF," in *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, O. Kullmann, Ed., in Lecture Notes in Computer Science, vol. 5584. Springer, 2009, pp. 181–194. doi: 10.1007/978-3-642-02777-2_19.

[64] I. Abío, V. Mayer-Eichberger, and P. J. Stuckey, "Encoding Linear Constraints into SAT," *CoRR*, 2020, [Online]. Available at: https://arxiv.org/abs/2005.02073

[65] I. Abío, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell, "A Parametric Approach for Smaller and Better Encodings of Cardinality Constraints," in *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, C. Schulte, Ed., in Lecture Notes in Computer Science, vol. 8124. Springer, 2013, pp. 80–96. doi: 10.1007/978-3-642-40627-0_9.

[66] F. Ulrich-Oltean, P. Nightingale, and J. A. Walker, "Learning to Select SAT Encodings for Pseudo-Boolean and Linear Integer Constraints," *Constraints An Int. J.*, vol. 28, no. 3, pp. 397–426, 2023, doi: 10.1007/S10601-023-09364-1.

[67] P. Surynek, "Linear Ordering in the SAT Encoding of the All-Different Constraint over Bit-Vectors," *ALP Newsletter, March/April*, 2014.

[68] N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit, "Global Constraint Catalogue: Past, Present and Future," *Constraints An Int. J.*, vol. 12, no. 1, pp. 21–62, 2007, doi: 10.1007/S10601-006-9010-8.

[69] R. Fourer, D. M. Gay, and B. W. Kernighan, "AMPL: A Mathematical Programming Language," *Management Science*, vol. 36, no. 5, pp. 519–554, 1990.

[70] F. Boussemart, C. Lecoutre, and C. Piette, "XCSP3: An Integrated Format for Benchmarking Combinatorial Constrained Problems," *CoRR*, 2016, [Online]. Available at: http://arxiv.org/abs/1611.03398

[71] A. M. Frisch, W. Harvey, C. Jefferson, B. M. Hernández, and I. Miguel, "Essence : A Constraint Language for Specifying Combinatorial Problems," *Constraints An Int. J.*, vol. 13, no. 3, pp. 268–306, 2008, doi: 10.1007/S10601-008-9047-Y.

[72] P. V. Hentenryck and L. Michel, "OPL Script: Composing and Controlling Models," in *New Trends in Contraints, Joint ERCIM/Compulog Net Workshop, Paphos, Cyprus, October 25-27, 1999, Selected Papers*, K. R. Apt, A. C. Kakas, É. Monfroy, and F. Rossi, Eds., in Lecture Notes in Computer Science, vol. 1865. Springer, 1999, pp. 75–90. doi: 10.1007/3-540-44654-0_4.

[73] T. 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 at CP (Modref 2019)*, 2019.

[74] N.-F. Zhou, H. Kjellerstrand, and J. Fruhman, *Constraint Solving and Planning with Picat*. in Springer Briefs in Intelligent Systems. Springer, 2015. doi: 10.1007/978-3-319-25883-6.

[75] D. S. Johnson and M. A. Trick, Eds., *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*, vol. 26. in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 26. DIMACS/AMS, 1996. doi: 10.1090/DIMACS/026.

[76] G. Belov, P. J. Stuckey, G. Tack, and M. Wallace, "Improved Linearization of Constraint Programming Models," in *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, M. Rueher, Ed., in Lecture Notes in Computer Science, vol. 9892. Springer, 2016, pp. 49–65. doi: 10.1007/978-3-319-44953-1_4.

[77] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual" (accessed November 2024). [Online]. Available at: https://www.gurobi.com/

[78] S. Bolusani *et al.*, "The SCIP Optimization Suite 9.0" (accessed November 2024), Feb. 2024. [Online]. Available at: https://optimization-online.org/2024/02/the-scip-optimization-suite-9-0/

[79] O. Ohrimenko, P. J. Stuckey, and M. Codish, "Propagation via Lazy Clause Generation," *Constraints: An International Journal*, vol. 14, no. 3, pp. 357–391, 2009, doi: 10.1007/s10601-008-9064-x.

[80] G. Chu, "Improving Combinatorial Optimization" (accessed November 2024) Ph.D. dissertation, University of Melbourne, Melbourne, 2011. [Online]. Available at: http://hdl.handle.net/11343/36679

[81] T. Feydy and P. J. Stuckey, "Lazy Clause Generation Reengineered," in *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, I. P. Gent, Ed., in Lecture Notes in Computer Science, vol. 5732. Springer, 2009, pp. 352–366.

[82] G. Gange, J. Berg, E. Demirovic, and P. J. Stuckey, "Core-Guided and Core-Boosted Search for CP," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 17th International Conference, CPAIOR 2020, Vienna, Austria, September 21-24, 2020, Proceedings*, E. Hebrard and N. Musliu, Eds., in Lecture Notes in Computer Science, vol. 12296. Springer, 2020, pp. 205–221.

[83] S. A. Wolfman and D. S. Weld, "The LPSAT Engine & Its Application to Resource Planning," in *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 Pages*, T. Dean, Ed., Morgan Kaufmann, 1999, pp. 310–317. [Online]. Available at: http://ijcai.org/Proceedings/99-1/Papers/046.pdf

[84] L. M. de Moura and N. S. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, C. R. Ramakrishnan and J. Rehof, Eds., in Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 337–340. doi: 10.1007/978-3-540-78800-3_24.

[85] H. Bierlee, "The MiniZinc-SAT Compiler" (accessed November 2024) M.S. Thesis, Uppsala University, Uppsala, 2021.

[86] T. Soh, M. Banbara, and N. Tamura, "A Hybrid Encoding of CSP to SAT Integrating Order and Log Encodings," in *27th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2015, Vietri Sul Mare, Italy, November 9-11, 2015*, IEEE Computer Society, 2015, pp. 421–428. doi: 10.1109/ICTAI.2015.70.

[87] A. Ignatiev, J. P. Marques-Silva, and A. Morgado, "A Python Library for Prototyping with SAT Oracles" (accessed November 2024). 2021.

[88] T. Philipp and P. Steinke, "PBLib - A Library for Encoding Pseudo-Boolean Constraints into CNF," in *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, M. J. H. Heule and S. A. Weaver, Eds., in Lecture Notes in Computer Science, vol. 9340. Springer, 2015, pp. 9–16. doi: 10.1007/978-3-319-24318-4_2.

[89] J. Huang, "Universal Booleanization of Constraint Models," in *International Conference on Principles and Practice of Constraint Programming*, Springer, 2008, pp. 144–158.

[90] N.-F. Zhou and H. Kjellerstrand, "The Picat-SAT Compiler," in *Practical Aspects of Declarative Languages - 18th International Symposium, PADL 2016, St. Petersburg, FL, USA, January 18-19, 2016. Proceedings*, M. Gavanelli and J. H. Reppy, Eds., in Lecture Notes in Computer Science, vol. 9585. Springer, 2016, pp. 48–62. doi: 10.1007/978-3-319-28228-2_4.

[91] I. Abío, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and V. Mayer-Eichberger, "A New Look at BDDs for Pseudo-Boolean Constraints," *Journal of Artificial Intelligence Research*, vol. 45, pp. 443–480, 2012, doi: 10.1613/jair.3653.

[92] C. Sinz, "Towards an Optimal CNF Encoding of Boolean Cardinality Constraints," in *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, P. van Beek, Ed., in Lecture Notes in Computer Science, vol. 3709. Springer, 2005, pp. 827–831. doi: 10.1007/11564751_73.

[93] N. Tamura, M. Banbara, and T. Soh, "Compiling Pseudo-Boolean Constraints to SAT with Order Encoding," in *25th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2013, Herndon, VA, USA, November 4-6, 2013*, IEEE Computer Society, 2013, pp. 1020–1027. doi: 10.1109/ICTAI.2013.153.

[94] M. J. Maher, "Propagation Completeness of Reactive Constraints," in *Logic Programming, 18th International Conference, ICLP 2002, Copenhagen, Denmark, July 29 - August 1, 2002, Proceedings*, P. J. Stuckey, Ed., in Lecture Notes in Computer Science, vol. 2401. Springer, 2002, pp. 148–162. doi: 10.1007/3-540-45619-8_11.

[95] C. Choi, W. Harvey, J. Lee, and P. J. Stuckey, "Finite Domain Bounds Consistency Revisited," in *Proceedings of the Australian Conference on Artificial Intelligence 2006*, in LNCS, vol. 4304. Springer-Verlag, 2006, pp. 49–58.

[96] N. Tamura and M. Banbara, "Sugar: A CSP to SAT Translator Based on Order Encoding," *Proceedings of the Second International CSP Solver Competition*, pp. 65–69, 2008.

[97] A. Metodi, M. Codish, and P. J. Stuckey, "Boolean Equi-Propagation for Concise and Efficient SAT Encodings of Combinatorial Problems," *Journal of Artificial Intelligence Research*, vol. 46, pp. 303–341, 2013, doi: http://dx.doi.org/10.1613/jair.3809.

[98] P. Barahona, S. Hölldobler, and V.-H. Nguyen, "Efficient SAT-encoding of Linear CSP Constraints," in *International Symposium on Artificial Intelligence and*

Mathematics, ISAIM 2014, Fort Lauderdale, FL, USA, January 6-8, 2014, 2014. [Online]. Available at: http://www.cs.uic.edu/pub/Isaim2014/WebPreferences/ISAIM2014_Barahona%20_etal.pdf

[99] B. Hurley, L. Kotthoff, Y. Malitsky, and B. A. O'Sullivan, "Proteus: A Hierarchical Portfolio of Solvers and Transformations," in *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings*, H. Simonis, Ed., in Lecture Notes in Computer Science, vol. 8451. Springer, 2014, pp. 301–317. doi: 10.1007/978-3-319-07046-9_22.

[100] H. Gorjiara, G. H. Xu, and B. Demsky, "Satune: Synthesizing Efficient SAT Encoders," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 1–32, 2020.

[101] T. Soh, D. Le Berre, M. Banbara, and N. Tamura, "sCOP: SAT-based Constraint Programming System," *Proceedings of XCSP3 Competition 2018 (XCSP18)*, pp. 93–94, 2018.

[102] N.-F. Zhou, "In Pursuit of an Efficient SAT Encoding for the Hamiltonian Cycle Problem," in *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-La-Neuve, Belgium, September 7-11, 2020, Proceedings*, H. Simonis, Ed., in Lecture Notes in Computer Science, vol. 12333. Springer, 2020, pp. 585–602. doi: 10.1007/978-3-030-58475-7_34.

[103] P. Nightingale, P. Spracklen, and I. Miguel, "Automatically Improving SAT Encoding of Constraint Problems Through Common Subexpression Elimination in Savile Row," in *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, G. Pesant, Ed., in Lecture Notes in Computer Science, vol. 9255. Springer, 2015, pp. 330–340. doi: 10.1007/978-3-319-23219-5_23.

[104] C. Ansótegui *et al.*, "Automatic Detection of At-Most-One and Exactly-One Relations for Improved SAT Encodings of Pseudo-Boolean Constraints," in *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, T. Schiex and S. de Givry, Eds., in Lecture Notes in Computer Science, vol. 11802. Springer, 2019, pp. 20–36. doi: 10.1007/978-3-030-30048-7_2.

[105] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, vol. 2. in The Kluwer International Series in Engineering and Computer Science, vol. 2. Springer, 1984. doi: 10.1007/978-1-4613-2821-6.

[106] N. Eén and A. Biere, "Effective Preprocessing in SAT through Variable and Clause Elimination," in *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, F. Bacchus and T. Walsh, Eds., in Lecture Notes in Computer Science, vol. 3569. Springer, 2005, pp. 61–75. doi: 10.1007/11499107_5.

[107] M. Bofill, J. Coll, P. Nightingale, J. Suy, F. Ulrich-Oltean, and M. Villaret, "SAT Encodings for Pseudo-Boolean Constraints Together with at-Most-One Constraints," *Artificial Intelligence*, vol. 302, p. 103604, 2022, doi: 10.1016/j.artint.2021.103604.

[108] I. Abío, V. Mayer-Eichberger, and P. J. Stuckey, "Encoding Linear Constraints with Implication Chains to CNF," in *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015,*

Proceedings, G. Pesant, Ed., in Lecture Notes in Computer Science, vol. 9255. Springer, 2015, pp. 3–11. doi: 10.1007/978-3-319-23219-5_1.

[109] W. Zhao, "Encoding Lexicographical Ordering Constraints in SAT," 2017.

[110] I. Abío, G. Gange, V. Mayer-Eichberger, and P. J. Stuckey, "On CNF Encodings for Decision Diagrams," in *Thirteenth International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming*, C.-G. Quimper, Ed., in LNCS. 2016, pp. 1–17.

[111] C. Schulte and G. Tack, "Views and Iterators for Generic Constraint Implementations," in *Recent Advances in Constraints*, B. Hnich, M. Carlsson, F. Fages, and F. Rossi, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 118–132.

[112] N.-F. Zhou and H. Kjellerstrand, "Optimizing SAT Encodings for Arithmetic Constraints," in *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, J. C. Beck, Ed., in Lecture Notes in Computer Science, vol. 10416. Springer, 2017, pp. 671–686. doi: 10.1007/978-3-319-66158-2_43.

[113] V. Lagoon and A. Metodi, "Deriving Optimal Multiplication-by-Constant Circuits with a SAT-based Constraint Engine," in *Proc. ModRef 19th Workshop Constraint Modelling Reformulation*, 2020, pp. 1–6.

[114] V. S. Dimitrov, L. Imbert, and A. Zakaluzny, "Multiplication by a Constant Is Sublinear," in *18th IEEE Symposium on Computer Arithmetic (ARITH-18 2007), 25-27 June 2007, Montpellier, France*, IEEE Computer Society, 2007, pp. 261–268. doi: 10.1109/ARITH.2007.24.

[115] A. Avizienis, "Signed-Digit Number Representations For Fast Parallel Arithmetic," *IRE Transactions on Electronic Computers*, no. 3, pp. 389–400, 1961, doi: 10.1109/TEC.1961.5219227.

[116] R. Garcia and A. Volkova, "Toward the Multiple Constant Multiplication at Minimal Hardware Cost," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 5, pp. 1976–1988, May 2023, doi: 10.1109/TCSI.2023.3241859.

[117] J. J. Dekker and H. Bierlee, "Pindakaas: CPAIOR-24" (accessed November 2024). [Online]. Available at: https://doi.org/10.5281/zenodo.10851856

[118] A. G. Dempster and M. D. Macleod, "Constant Integer Multiplication Using Minimum Adders," *Iee proceedings-circuits, devices and systems*, vol. 141, no. 5, pp. 407–413, 1994.

[119] O. Gustafsson, "Towards Optimal Multiple Constant Multiplication: A Hypergraph Approach," in *42nd Asilomar Conference on Signals, Systems and Computers, ACSSC 2008, Pacific Grove, CA, USA, October 26-29, 2008*, IEEE, 2008, pp. 1805–1809. doi: 10.1109/ACSSC.2008.5074738.

[120] M. Kumm, "Optimal Constant Multiplication Using Integer Linear Programming," *IEEE Trans. Circuits Syst. II Express Briefs*, no. 5, pp. 567–571, 2018, doi: 10.1109/TCSII.2018.2823780.

[121] S. Ma and P. Ampadu, "Optimal SAT-based Minimum Adder Synthesis of Linear Transformations," in *62nd IEEE International Midwest Symposium on Circuits and Systems, MWSCAS 2019, Dallas, TX, USA, August 4-7, 2019*, H. Lee and R. L. Geiger, Eds., IEEE, 2019, pp. 335–338. doi: 10.1109/MWSCAS.2019.8885033.

[122] L. Aksoy, P. F. Flores, and J. Monteiro, "Exact and Approximate Algorithms For The Filter Design Optimization Problem," *IEEE Transactions on Signal Processing*, vol. 63, no. 1, pp. 142–154, 2015, doi: 10.1109/TSP.2014.2366713.

[123] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT Solvers to Cryptographic Problems," in *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, O. Kullmann, Ed., in Lecture Notes in Computer Science, vol. 5584. Springer, 2009, pp. 244–257. doi: 10.1007/978-3-642-02777-2_24.

[124] B. Han, J. Leblet, and G. Simon, "Hard Multidimensional Multiple choice Knapsack Problems, an Empirical Study," *Computers & Operations Research*, vol. 37, no. 1, pp. 172–181, 2010, doi: 10.1016/j.cor.2009.04.006.

[125] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, T. Balyo, N. Froleyks, M. J. H. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., in Department of Computer Science Report Series B. University of Helsinki, 2020, pp. 51–53.

[126] M. Bofill, J. Coll, J. Suy, and M. Villaret, "An MDD-based SAT Encoding for Pseudo-Boolean Constraints with at-Most-One Relations," *Artificial Intelligence Review*, vol. 53, no. 7, pp. 5157–5188, 2020, doi: 10.1007/s10462-020-09817-6.

[127] W. Harvey and P. J. Stuckey, "Improving Linear Constraint Propagation by Changing Constraint Representation," *Constraints : an international journal*, vol. 8, no. 2, pp. 173–207, 2003.

[128] J. Cocke, "Global common subexpression elimination," in *Proceedings of a Symposium on Compiler Optimization, Urbana-Champaign, Illinois, USA, July 27-28, 1970*, R. S. Northcote, Ed., ACM, 1970, pp. 20–24. doi: 10.1145/800028.808480.

[129] H. Bierlee and J. J. Dekker, "Pindakaas: CPAIOR-25 (submission)." [Online]. Available at: https://doi.org/10.5281/zenodo.14500064