SAT Encodings of Finite CSPs

Dissertation
zur Erlangung des akademischen Grades
Doktoringenieur (Dr.Ing.)

vorgelegt an der
Technische Universität Dresden
Fakultät Informatik

eingereicht von
Msc. Van-Hau NGUYEN

Promotionskommission:

Gutachter: Prof.Dr.habil. Steffen HÖLLODOLBER
(Technische Universität Dresden)
Prof.Dr. Pedro BARAHONA
(Universidade Nova de Lisboa)

Fachreferent: Prof.Dr. Sebastian RUDOLPH
(Technische Universität Dresden)

Vorsitzender: Prof.Dr. Raimund DACHSELT
(Technische Universität Dresden)

Mitglied: Prof.Dr. Michael SCHRÖDER
(Technische Universität Dresden)
Acknowledgements

Special thanks Steffen Hölldobler for his excellent supervision. This thesis would not have been possible without his support. Not only did he provide me the freedom to choose my own directions for the research, but he also guided me in a way that ensured I never lost my overview of the whole picture. He has followed every step in my PhD life; while patiently and constantly trained me in doing research. He supplied my pathway into the European PhD Program in Computational Logic (EPCL) program which gave me many opportunities to enrich my research experience by studying aboard, participating training camps regularly, communicating with many experts in the field. I am continuously learning from him, not only in regular seminars, but even through our everyday interactions. Throughout this journey of my research I have carried his words of wisdom. I am forever grateful for his guidance.

I am also truly thankful to Pedro Barahona for his patience and being able to guide me through my ideas for this research. With him, my mind can fly freely through the process of developing and exchanging research context. He is the one who is able to make the most mundane tasks become very interesting. He is not just an excellent professor, but a very good friend. Thanks to him, I had a good chance to stay in Lisbon and I will never forget the periods I stayed in Lisbon - such a unique city within Europe! These words are not enough to show my appreciation to Pedro Barahona, who has always been an inspiration through his ideas and enthusiasm.

I am indebted to Christoph Wernhard, a wonderful office mate, for his essential support. Not only did he help me in doing research, but he also helped me in daily issues. Christoph was very patient in correcting my writing. I thank him for suggesting the necessary adjustments to be made from the early phrase of my thesis. With him, I have had also fruitful discussion throughout this research.

I am most thankful for the opportunity to work with several experts, who have helped me overcome many difficulties in my research. I thank Dr. Steven Prestwich for the many valuable discussions. I am also very grateful to Dr. Miroslav N. Velev for his co-operation.

This acknowledgement would not be complete without extending my sincere thanks to Prof. Sebastian Rudolph for being my Fachreferent, to
Prof. Raimund Dachselt and Prof. Michael Schröder for being members of the jury.

I have been a part of a very good group thanks to my colleagues. I greatly acknowledge Norbert Manthey and Peter Stein, who helped me significantly from the very beginning, together with Emmanuelle-Anna Dietz and Tobias Philipp. I thank Sylvia Wünsch and Julia Kopenhagen for providing me with all the necessary logistic support and many other things.

My thesis has been a success due to many financial supports. Many thanks to: the Vietnam government for a three-year scholarship; DAAD (Deutscher Akademischer Austauschdienst) for language, culture, and subsidy support; the Leonardo program for six-month abroad (Lisbon); EPCL for my stay at the partner university (Universidade Nova de Lisboa); Förderverein Freunde und Förderer der Informatik der TU Dresden e.V. for one conference expense; and Gesellschaft von Freunden und Förderern der TU Dresden e.V for three-month financial support.

I thank wholeheartedly my friends for being with me. Many thanks to Luu Hoang Duc and Nguyen Hong Duong for their endless encouragement and support - you are admirable people. Special thank goes to Ari Saptawijaya whose warm hospitality sustained me during the period I stayed in Lisbon. Special thanks to my dear friends, Stephanie McGill and Joss Dent, for your careful reading and numerous remarks on my thesis.

Finally, I cannot sufficiently express my gratitude all of my family members. I am indebted to every single member: Dao Thi Hanh, Nguyen Thi Minh Khanh, and Nguyen Thi Khanh Van. If I have to choose people to whom I dedicate my thesis, without any hesitation I choose my parents. I am aware of that I probably will never be able to fully express my thanks and gratitude to all that they have done for me - they have been a source of uplift and strength in my life. My father, who passed away during the most difficult point of this thesis, would have been proud and happy.
Abstract

Boolean satisfiability (SAT) is the problem of determining whether there exists an assignment of the Boolean variables to the truth values such that a given Boolean formula evaluates to true. SAT was the first example of an NP-complete problem [Coo71]. Only two decades ago SAT was mainly considered as of a theoretical interest. Nowadays, the picture is very different. SAT solving becomes mature and is a successful approach for tackling a large number of applications, ranging from artificial intelligence to industrial hardware design and verification.

SAT solving consists of encodings and solvers. In order to benefit from the tremendous advances in the development of solvers, one must first encode the original problems into SAT instances. These encodings should not only be easily generated, but should also be efficiently processed by SAT solvers. Furthermore, an increasing number of practical applications in computer science can be expressed as constraint satisfaction problems (CSPs). However, encoding a CSP to SAT is currently regarded as more of an art than a science, and choosing an appropriate encoding is considered as important as choosing an algorithm. Moreover, it is much easier and more efficient to benefit from highly optimized state-of-the-art SAT solvers than to develop specialized tools from scratch. Hence, finding appropriate SAT encodings of CSPs is one of the most fascinating challenges for solving problems by SAT.

This thesis studies SAT encodings of CSPs and aims at: 1) conducting a comprehensively profound study of SAT encodings of CSPs by separately investigating encodings of CSP domains and constraints; 2) proposing new SAT encodings of CSP domains; 3) proposing new SAT encoding of the at-most-one constraint, which is essential for encoding CSP variables; 4) introducing the redundant encoding and the hybrid encoding that aim to benefit from both two efficient and common SAT encodings (i.e., the sparse and order encodings) by using the channeling constraint (a term used in Constraint Programming) for SAT; and 5) revealing interesting guidelines on how to choose an appropriate SAT encoding in the way that one can exploit the availability of many efficient SAT solvers to solve CSPs efficiently and effectively. Experiments show that the proposed encodings and guidelines improve the state-of-the-art SAT encodings of CSPs.


## Contents

1 Introduction ............................................. 1
   1.1 Motivations ........................................... 1
   1.2 Challenges of SAT Encodings ......................... 3
   1.3 Objectives ............................................ 3
   1.4 Outline ............................................... 5

2 Preliminaries ............................................ 7
   2.1 Constraint Programming ................................ 7
       2.1.1 Constraint Satisfaction Problem .................. 7
       2.1.2 Local Consistency ................................ 8
       2.1.3 Benchmarks ...................................... 11
   2.2 Boolean Satisfiability ................................. 23
       2.2.1 Syntax of Propositional Logic ................. 23
       2.2.2 Semantics of Propositional Logic ............. 24
       2.2.3 Satisfiability .................................. 25
       2.2.4 SAT with CNF .................................. 26
       2.2.5 Conflict-Driven Clause Learning SAT Solvers .. 29
   2.3 CSP and SAT .......................................... 30
       2.3.1 SAT is a CSP .................................... 30
       2.3.2 Encoding a CSP to SAT ......................... 30
   2.4 Discussion ............................................ 31

3 SAT Encodings of Finite CSP Domains .................. 33
   3.1 Sparse Encoding ....................................... 35
   3.2 Order Encoding ........................................ 37
   3.3 Log Encoding .......................................... 38
   3.4 Compact Order Encoding ............................... 39
   3.5 Hierarchical and Hybrid SAT Encodings ............. 40
   3.6 Representative Encodings ............................ 41
       3.6.1 The Representative-Sparse Encoding .......... 41
       3.6.2 The Representative-Order Encoding .......... 43
   3.7 Discussion ............................................ 45
       3.7.1 Overview ....................................... 45
       3.7.2 Are ALO and AMO Clauses Necessary? ........... 46

4 SAT Encodings of Finite CSP Constraints ............... 49
   4.1 Encoding Extensional and Intensional Constraints ... 50
       4.1.1 Conflict Clauses ................................ 52
       4.1.2 Support Clauses ................................ 56
   4.2 Encoding the At-Most-One Constraint ................ 61
4.2.1 The AMO Pairwise Encoding ........................................ 62
4.2.2 The AMO Binary Encoding .......................................... 63
4.2.3 The AMO Commander Encoding .................................... 63
4.2.4 The AMO Product Encoding ......................................... 64
4.2.5 The AMO Sequential Counter Encoding ................................ 65
4.2.6 The AMO Bimander Encoding ....................................... 66
4.2.7 Comparison ............................................................ 69
4.3 Encoding Linear CSP Constraints ........................................ 74
4.3.1 The Sparse and Order Encodings of Linear CSP Constraints .... 74
4.3.2 The \(Sp-Or_{red}\) and \(Sp-Or_{hyb}\) Encodings .................... 76

5 Experimental Results ...................................................... 79
5.1 SAT Encodings of Finite CSPs ......................................... 80
5.2 SAT Encodings of the At-Most-One Constraint ...................... 95
5.3 SAT Encodings of Linear CSP Constraints ......................... 107
5.4 Discussion ............................................................... 115
5.4.1 The AMO Encodings - Related Work ............................ 115
5.4.2 What Feature Makes One Encoding Better Than Another? .... 116

6 The SAT vs. CP Approaches .............................................. 119
6.1 The All-Interval Series Problem ....................................... 120
6.1.1 The CP Modeling .................................................... 122
6.1.2 The SAT Encoding .................................................. 122
6.1.3 Reformulation ....................................................... 125
6.1.4 Experimental Evaluation .......................................... 126
6.2 The Quasigroup With Holes Problem ................................. 129
6.2.1 The CP Modeling .................................................... 130
6.2.2 The SAT Encoding .................................................. 131
6.2.3 Experimental Evaluation .......................................... 132
6.3 The Langford Problem .................................................. 133
6.3.1 The CP Modeling .................................................... 134
6.3.2 The SAT Encoding .................................................. 134
6.3.3 Experimental Evaluation .......................................... 135
6.4 The Golomb Ruler Problem ............................................. 136
6.4.1 Experimental Evaluation .......................................... 137
6.5 Other Problems ......................................................... 138
6.6 Conclusions ............................................................. 139

7 Conclusions .............................................................. 141
7.1 SAT Encodings of Finite CSPs ....................................... 141
7.2 SAT Encodings of Finite CSP Domains ............................... 141
7.3 SAT Encodings of the At-Most-One Constraint .................... 142
7.4 SAT Encodings of Linear CSP Constraints .......................... 143
7.5 SAT Encodings: an Ultimate Progress ............................... 145
Contents

Bibliography
1.1 Motivations

Boolean satisfiability (SAT) was the first problem shown to be NP-complete by Cook in 1971 [Coo71] and SAT is a core of a large number of computationally intractable problems [Kar72, GJ79]. If one NP-complete problem can be solved in polynomial time, all the others can be solved [Kar72, GJ79]. Therefore, when optimizing SAT, one gets a better chance for solving many other NP-complete problems.

SAT is not only of theoretical interest, but also a noteworthy achievement in practice. The SAT approach has been one of the most successful automated reasoning methods in computer science by solving a large number of both industrial and academic problems in the last two decades. A wide range of practical and challenging applications have been encoded to SAT, for example, validating software models [JV00], scheduling basketball games [Zha03], routing field programmable gate arrays [NSR99], and synthesizing consistent network configurations [NLMK08], AI planning [KS92, KMS96, Kau06, HKH06, KS99, Rin09, Rin11], product configuration [SKK00], model checking [CBRZ01, BK02, BCC+03, Vel04, HGSK07], design debugging [SVAV05, CSSV10], challenging applications from algebra [Zha09], and haplotype inference in bioinformatics [LS06]. Furthermore, many extensions of SAT, using SAT solvers as core engines, have increasingly impacted a number of decision and optimization AI problems such as satisfiability modulo theories (SMT), pseudo-Boolean (PB), maximum satisfiability (MaxSAT), model counting (∼SAT), and quantified-Boolean formulas (QBF). The application list has been on the rise in recent years (see [BHvMW09a]).

Although translating a problem into SAT instances might cause a substantially large formula, this is no longer an obstacle for the state-of-the-art SAT solvers. In the early 90s, SAT solvers only solved a SAT instance of about 100 variables and 200 clauses, the current modern SAT solvers can deal tremendously well with millions of variables, and millions of clauses. SAT algorithms have been intensively proposed and evaluated in the last decade. Particularly, the SAT community organizes SAT competitive events (SAT competition or SAT Races) and the winners of those events frequently set new standards in the area [SAT, JBR12]. Both complete SAT solvers, based on conflict-driven clause learning (CDCL) algorithm [SS96, MMZ+01, ES05, SLM09, Man14] and incomplete solvers, based on stochastic local search, have reached significant levels of high performance. Figure 1.1 clearly depicts the annual improvements of SAT solvers. Consequently, SAT solving is an attractive approach to solve many practical prob-
Figure 1.1: Speedup of SAT solvers in recent years (taken from [Ber14]).

problems. It is quite competitive against other general-purpose methods as well as other special-purpose or traditional methods [Hoo99, AGKS00, BB03, AdVD+04, BB04, GARK07, TTKB09, Zha09, LZMS11, PJ11, JP12]. Take for instance the international conference on automated planning and scheduling, and the international competition constraint programming (CP) solvers in recent years; while in the past conventional CP algorithms outperformed the other approaches in these events, the SAT-based solvers have been remarkably competitive and have reached top ranks ([HKH04, HKH06, vDLRb, vDLRa, BP10]).

On the one hand, in order to benefit from the tremendous advances in the development of SAT solvers, one must first encode the original problems into SAT formulas. These encodings should not only be easily generated, but also be efficiently processed by solvers. On the other hand, an increasing number of practical and important problems in computer science [GW99] are stated as constraint satisfaction problems (CSPs) [RBW06]. Hardly any problems are originally given by SAT formulas. Hence, attacking CSPs by encoding them into SAT and then using highly optimized solvers is an appealing approach. Additionally, it seems clear that specifying a particular problem in the form of SAT formulas is significantly easier than implementing a dedicated CP solver in order to reach the high performance as modern SAT solvers ([FPDN05, Zha09]). In other words, throughout this approach, tackling practical applications will directly get benefits from any improvements of SAT solvers with no additional cost (i.e., one will keep the same encodings, but
1.2. Challenges of SAT Encodings

We address here four main challenges in the area of SAT encodings.

Firstly, in order to access the tremendous performance of SAT solvers, many benchmarks might come from intrinsically easy problems carelessly via an inappropriate encoding [ML08, Pre09, HMNS12]. As a consequence, SAT encodings must be a crucial factor beside SAT solvers. It is well-understood that two SAT encodings used on one problem might have substantially different effects on a SAT solver. One encoding is exponential for resolution, whilst the other encoding is polynomial although two encodings are of polynomial size [HHU07, HMNS12].

Secondly, SAT encodings have played an increasingly important role for the success of SAT-encoded problem solving. In ten challenges for SAT stated by Kautz and Selman [KS07], three out of those are for SAT encodings; the eighth is that: “Characterize the computational properties of different encodings of a real-world problem domain, and/or give general principles that hold over a range of domains”. It has been observed that constructing an effective encoding can be a difficult task in many cases, thus requiring a great deal of expertise. Researchers agree that encoding a problem is currently regarded as more of an art than a science [Wal00, Gen02, Pre03b, Pre03a, Zha09, Pre09, BHN14a]. As a result, finding an appropriate encoding of hard problems into SAT instances is one of the most appealing challenges in SAT solving [Wal00, Gen02, Pre03b, Pre03a, Zha09, Pre09, BHN14a].

Thirdly, besides the algorithms, a proficient and suitable encoding is a crucial contribution to solving efficiently difficult and practical problems. It is undoubtedly that choosing an appropriate SAT-encoding scheme is considered as important as choosing a proficient solver [BB03, AG09].

Fourthly, notwithstanding the steadily increasing diffusion and availability of SAT solvers, understanding SAT encodings is still very limited and challenging for solving hard and practical problems. Although it plays an essential role in the efficiency of solving problems [ML08, HMNS12], there is still an appealing lack of guidance for choosing appropriate encodings for solving specific SAT instances.

1.3 Objectives

This thesis aims at proposing new SAT encodings and conducting a profound study of SAT encodings of CSPs understandably, comparably, and comprehensively. Particularly, this thesis aims at four following contributions.

Firstly, this thesis represents almost all SAT encodings in a comprehensive, understandable, and comparable way. A CSP is defined by a set of variables, a set of domains, and a set of constraints. This thesis studies SAT-encoding schemes by separately investigating SAT encodings of CSP variables with their corresponding
domains and SAT encodings of CSP constraints. The independent consideration of these two parts may help the SAT community to easily study their integration, avoid confusion and facilitate progress in SAT encoding.

Secondly, this thesis proposes new SAT encodings of finite CSP domains. The number of ways for translating a problem to SAT is exactly the number of ways one can use to solve that problem through SAT solving. Obviously, the more SAT-encoding methods one has, the better the chance of achieving success with solving practical problems. This is probably one of the most important goals in SAT encodings. Furthermore, this thesis also studies and compares new SAT encodings with existing ones by both theoretical analyses and empirical evaluation.

Thirdly, in addition to a survey of almost all at-most-one (AMO) SAT encodings that are widely used in SAT encodings of finite CSPs, this thesis proposes a new AMO SAT-encoding. For each AMO SAT-encoding, we address interesting observations: (1) the unit propagation strength and the number of clauses generated; and (2) the relationship between the auxiliary variables required by an AMO encoding and the variables used by its corresponding SAT encoding of finite CSP domains.

Fourthly, this thesis introduces new SAT encodings of linear CSP constraints, which consists of disequality and inequality constraints. We point out the strength and weakness of the sparse encoding and the order encoding, two widely used and efficient encodings of CSPs into SAT. We will show that the former encoding is adequate to propagate disequality constraints while the other is better to handle inequality constraint in the context of linear CSP problems. The new encodings obtained by combining the two models through channeling constraints (as common in Constraint Programming) allow one to benefit from both worlds while not incurring significant overhead. Moreover, this thesis provides several guidelines regarding the choice of suitable SAT encodings for CSPs, taking into account several features of these problems.
1.4 Outline

Chapter 1: Introduction
The chapter introduces motivations, challenges, goals and the structure of this thesis.

Chapter 2: Preliminaries
We will briefly provide basic notations, concepts and definitions on CSP, SAT, and the connection between them. Several important transformation techniques concerning conjunctive normal form (CNF) will be addressed. Furthermore, we will show ten CSPs which are used as the benchmarks for this thesis.

Chapter 3: SAT Encodings of Finite CSP Domains
Firstly, we will present an explicit perspective and a comprehensive study of SAT encodings of finite CSPs by providing a careful comparison among them. Secondly, we will propose the two representative encodings: the representative-sparse encoding and the representative-order encoding. Theoretically, the representative encodings get the greatest benefit of balance between the lower number of variables and the powerful propagation. Next, we will end this chapter with some discussions.

Chapter 4: SAT Encodings of Finite CSP Constraints
We will explicitly represent two kinds of CNF clauses for SAT encodings of extensional and intensional constraints in CSPs. Next, we will address the strengths and drawbacks of two widely used encodings, the sparse and order encodings. Combining the two models through channeling constraints allows one to benefit from both worlds while not incurring significant overhead. Consequently, we propose two new encodings, based on redundant and hybrid modeling. We also give some guidelines regarding the choice of suitable SAT encodings for CSP problems, taking into account several features of these problems. Finally, we will not only give a comprehensive survey of the most well-known at-most-one (AMO) SAT encodings with a running example, but also propose a new AMO SAT encoding.

Chapter 5: Experimental Results
The chapter will be concerned with experiments which illustrate for Chapters 3 and 4. Next, we discuss the interesting question: “What feature makes one encoding better than another?” We will close this chapter with some observations, taking into account the similarities among several AMO SAT encodings, which may help some people to avoid confusion among encodings, presented on literature.

Chapter 6: The SAT vs. CP approaches
The goal of this chapter is to discuss the SAT approach and the CP approach on all the benchmarks used in this thesis. We also show in detail how some problems are modelled in SAT and CP. Interestingly, SAT remarkably outperforms CP on several a multiple permutation problem, which require multiple times the constraint alldifferent - one of the most studied and used global constraint in CP. We roughly address the characteristics of a specific problem to determine which approach could produce a better performance.

Chapter 7: Conclusions
We will conclude this thesis by summarizing the contributions achieved and give several directions for future research.
CHAPTER 2

Preliminaries

It is well-acknowledged that whenever one NP-complete problem can be solved in polynomial time, all the others can be solved as well. Through the transformation of one problem to another one in polynomial time, researchers can take advantage of the efficiency of solving a certainly-known problem (e.g., SAT) for other (unsolved) problems (e.g., constraint satisfaction problems - CSPs). Therefore, the possibility to solve many CSPs can be gained via the success of SAT solving.

This chapter provides basic notations, concepts and definitions of constraint programming - CP, SAT, and the connection between CSP and SAT. Furthermore, we present some important knowledge of conjunctive normal form (CNF), and try to address the question: "Why is CNF considered as a standard format for SAT solver?"

2.1 Constraint Programming

Constraint programming (CP) has been playing a crucial role in modelling and reasoning for a wide range of real world problem domains. It provides a powerful paradigm for solving constraint satisfaction problems (CSPs) in artificial intelligence, computer science, and operations research. CP solving consists of a large number of methods, algorithms, approaches and languages. This section presents only basic definitions and important techniques used throughout this thesis. For more details we refer the reader to [RBW06].

2.1.1 Constraint Satisfaction Problem

**Definition 2.1.1** A constraint satisfaction problem (CSP) is a triple $\langle V, D, C \rangle$, where

- $V := \{V_1, \ldots, V_k\}$ is a set of variables,
- $D := \{D(V_1), \ldots, D(V_k)\}$ is a set of domains,
- $C := \{C_1, \ldots, C_m\}$ is a set of constraints.

A constraint $C_i$ is a pair $(R_{C_i}, S_{C_i}), 1 \leq i \leq m$, where $S_{C_i}$ is scope of $C_i$, and $R_{C_i}$ is a relation on the variables in $C_i$ (i.e., $R_{C_i}$ is a subset of the Cartesian product of the domains of the variables in $S_{C_i}$).

A tuple $\langle v_1, \ldots, v_k \rangle \in \langle D(V_1), \ldots, D(V_k) \rangle$ satisfies a constraint $C_i = (R_{C_i}, S_{C_i})$, if the projection of the tuple into the constraint variables is a member of the constraint, i.e., if $\langle v_1, \ldots, v_k \rangle \downarrow S_{C_i} \in R_{C_i}$, where $\downarrow$ denotes a projection operator. A tuple is consistent with a certain constraint if it satisfies the constraint. A tuple is
inconsistent with a certain constraint if it does not satisfy the constraint. A tuple \( \langle v_1, \ldots, v_k \rangle \) is a solution to a CSP if it satisfies all constraints in \( C \). The CSP problem is to find an assignment (or all assignments) for all the variables that satisfies all the constraints.

A CSP consisting of a finite set of variables and a finite domain for each variable is a finite CSP. This thesis mostly restricts the attention to binary finite CSPs, i.e., finite CSPs where all constraints are between at most two variables. Note that any non-binary CSP constraint can be converted into an equivalent binary constraint [BvB98].

**Example 2.1.2** Consider a binary CSP \( P = (V, D, C) \) where:
\[
V := \{x_1, x_2, x_3\}, \\
D := \{\{0, 1, 2\}, \{1, 2, 3\}, \{-1, 0, 1\}\}, \\
C := \{\langle x_1 = x_2, \{x_1, x_2\}\rangle, \langle x_2 > x_3, \{x_2, x_3\}\rangle\}.
\]

The following tuples are solutions of \( P \):
\[
\{(1, 1, -1), (1, 1, 0), (2, 2, -1) (2, 2, 0), (2, 2, 1)\}.
\]

### 2.1.2 Local Consistency

Most state-of-the-art CP solvers apply two main methods: backtracking search and local consistency. Backtracking search algorithms perform a depth-first traversal of a search tree. Particularly, the idea is to incrementally build a partial assignment by instantiating a new node (or variable) with a value from its domain of the search tree, such that the value satisfies all constraints whose scope is a subset of the already assigned variables. Local consistency algorithms perform inferences aiming at eliminating infeasible domain values of variables, and reducing the search space.

Propagation algorithms enforce a local consistency notion and play an important role in the efficiency of CP solvers. For this reason, many consistency criteria have been investigated. We refer the reader to [vHK06] for the details.

Arc consistency is one of the most commonly enforced forms of consistency in CP. The idea is to guarantee that every value in a domain must be consistent with every constraint. Moreover, arc-consistency is a useful technique because of its tradeoff between the cost of the constraint propagation performed at each node in the search tree and pruning efficiency. Consequently, finding efficient algorithms for enforcing arc-consistency is crucial in CP [Bes06, vHK06].

**Definition 2.1.3 (Arc-consistency)** A binary constraint \( C = \langle R_C, \{x_1, x_2\}\rangle \) is arc-consistent if for each value \( d_1 \in D(x_1) \) there exists a value \( d_2 \in D(x_2) \) such that \( (d_1, d_2) \in R_C \), and for each value \( d_2 \in D(x_2) \) there exists a value \( d_1 \in D(x_1) \) such that \( (d_1, d_2) \in R_C \).

**Example 2.1.4** Consider a binary CSP \( P = (V, D, C) \) where:
\[
V := \{x_1, x_2, x_3\}, \\
D := \{\{0, 1, 2\}, \{1, 2, 3\}, \{-1, 0, 1\}\}, \\
C := \{\langle x_1 = x_2, \{x_1, x_2\}\rangle, \langle x_2 > x_3, \{x_2, x_3\}\rangle\}.
\]
2.1. Constraint Programming

Consider the constraint \( C_1 = \{x_1 = x_2, \{x_1, x_2\}\} \), we see that the value 0 from \( D(x_1) \) has no value equal to it in \( D(x_2) \). Similarly, the value 3 from \( D(x_2) \) has no value greater than it in \( D(x_1) \). Therefore, \( P \) is currently not arc-consistent. To achieve arc-consistency, we have to remove all the inconsistent values. We will address this issue later.

An extension of arc-consistency for a constraint with more than two variables in its scope is called generalized arc-consistency. A variable \( x \) is generalized arc-consistency with a constraint, say \( C \), if every value in its domain can be extended to all other variables of \( C \) such that the extended tuples satisfy \( C \).

**Definition 2.1.5 (Generalized arc-consistency)** A non-binary constraint \( C = \langle R_C, \{x_1, \ldots, x_m\}\rangle(m > 2) \) is generalized arc-consistent if for all \( i \in \{1, \ldots, m\} \) and all values \( d_i \in D(x_i) \), there are values \( d_j \in D(x_j) \) for all \( j \in \{1, \ldots, m\} \setminus \{i\} \) such that \( (d_1, \ldots, d_m) \in R_C \).

**Example 2.1.6** Consider a non-binary CSP \( P = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle \) where:

\[
\mathcal{V} := \{x_1, x_2, x_3\}, \\
\mathcal{D} := \\{(0,1,2), \{1,2,3\}, \{-1,0\}\}, \\
\mathcal{C} := \{(x_1 + x_2 = x_3, \{x_1, x_2, x_3\}\}).
\]

\( P \) is not generalized arc-consistent since the sum \( x_1 + x_2 \) is always strictly greater than zero, whereas \( x_3 \) is always less than or equal to zero. In other words, for all values in the domains of variables \( x_1 \) and \( x_2 \), there exists no value for \( x_3 \) that satisfies the constraint \( x_1 + x_2 = x_3 \).

For some specific domains, one can apply some variants of arc-consistency. Particularly, the following consistency is regarded as a relaxation of arc-consistency and generalized arc-consistency. Let \( \min(Dx) \) and \( \max(Dx) \) denote the minimum and maximum values of the domain for variable \( x \), respectively, and the interval notation \([a..b]\) denotes the set of consecutive values \([a, a+1, \ldots, b]\).

**Definition 2.1.7 (Bounds consistency)** A constraint \( C = \langle R_C, \{x_1, \ldots, x_m\}\rangle(m > 1) \) is bounds consistent if for all \( i \in \{1, \ldots, m\} \) and each \( d_i \in \{\min(Dx_i), \max(Dx_i)\} \), there exists a value \( d_j \in [\min(Dx_j)..\max(Dx_j)] \) for all \( j \in \{1, \ldots, m\} \setminus \{i\} \) such that \( (d_1, \ldots, d_i, \ldots, d_m) \in R_C \).

The aim of bounds consistency is to avoid checking all values in the domains like (generalized) arc-consistency, but only the minimum and maximum values.

**Example 2.1.8** Consider a binary CSP \( P = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle \) where:

\[
\mathcal{V} := \{x_1, x_2, x_3\}, \\
\mathcal{D} := \\{[4..8], [0..3], [2..2]\}, \\
\mathcal{C} := \{(x_1 = x_2 + x_3, \{x_1, x_2, x_3\}\}).
\]

Suppose that \( x_1 = 8 \), for which one obtains the equation \( x_2 + x_3 = 8 \). Since there are no values for \( x_2 \in [0..3] \) and \( x_3 \in [2..2] \) that satisfy the equation, the CSP is not bounds consistent. In fact, to achieve bounds consistency for \( P \), one can use the following propagation rules (see [Apt03]):
\[ x_1 \geq \min(Dx_2) + \min(Dx_3) = 2, \quad x_1 \leq \max(Dx_2) + \max(Dx_3) = 5 \]
\[ x_2 \geq \min(Dx_1) - \min(Dx_3) = 2, \quad x_2 \leq \max(Dx_1) - \min(Dx_3) = 6 \]
\[ x_3 \geq \min(Dx_1) - \min(Dx_2) = 1, \quad x_3 \leq \max(Dx_1) - \min(Dx_2) = 8 \]

The possible domains for \( x_1, x_2, \) and \( x_3 \), obtained by the propagation rules are \([2.5],[2.6],\) and \([1.8]\), respectively. Finally, we achieve the bounds consistent CSP when combining the obtained intervals with the initial intervals: \( \mathcal{D} = \{[4..5],[2..3],[2..2]\} \).

In order to prevent the search from encountering conflicts, forward checking (FC) is widely used. FC guarantees arc-consistency for all constraints between the current variables (i.e., already assigned) and the unassigned variables. Particularly, at each step of extending the current partial assignment any value of an unassigned variable that conflicts with this assignment is (temporarily) removed from its domain. As a result of forward checking, all remaining values of the current variable are guaranteed to be consistent with the past variables.

**Example 2.1.9** Consider a binary CSP consisting of a triple \( \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle \) where:
\[ \mathcal{V} := \{x_1, x_2, x_3, x_4\}, \]
\[ \mathcal{D} := \{\{0, 1\}, \{0, 1\}, \{0, 1, 2\}, \{0, 1, 2, 3\}\}, \]
\[ \mathcal{C} := \{\{x_i \neq x_j, \{x_i, x_j\}\}, 1 \leq i < j \leq 4\}. \]
The above constraint is very common in CSP, also known as alldifferent(\(x_1, x_2, x_3, x_4\)).

Suppose that the current instantiation sets \( x_1 = 0 \). This instantiation eliminates the value \( \{0\} \) from \( D(x_2) \), \( D(x_3) \), and \( D(x_4) \) due to constraints containing the variable \( x_1 \). The domains are then reduced to \( \mathcal{D} = \{\{0\}, \{1\}, \{1, 2\}, \{1, 2, 3\}\} \).

It is worth noticing that enforcing FC occurs only between the current variable and exactly one uninstantiated variable.

Compared to FC, maintaining arc-consistency (MAC) can further detect inconsistencies by maintaining full arc-consistency. While FC detects only the conflicts between the current variable and unassigned variables, MAC detects the conflicts among unassigned variables. Since MAC enforces arc-consistency between every pair of the unassigned variables, MAC can further detect inconsistencies between a pair of unassigned variables without assigning them any value. Consequently, MAC has a smaller search space although it requires a larger amount of computation than FC.

To enforce FC on a binary CSP, the algorithm requires \( O(ed) \) worst case time complexity, where \( e \) is the number of constraints and \( d \) is the size of the domains of the variables. On the contrary, many different algorithms to enforce arc-consistency have been proposed (for the details, see [Bes06]). An optimal algorithm achieves arc-consistency in \( O(ed^2) \) worst case time complexity [Bes06].
Example 2.1.10 Consider a binary CSP \( P = (\mathcal{V}, \mathcal{D}, \mathcal{C}) \) where:
\[
\mathcal{V} := \{x_1, x_2, x_3, x_4\},
\mathcal{D} := \{\{0, 1\}, \{0, 1, 2\}, \{0, 1, 2, 3\}\},
\mathcal{C} := \{\{x_i \neq x_j\}, \{x_i, x_j\}\}, 1 \leq i < j \leq 4\}.
\]

Consider the current instantiation \( x_1 = 0 \). Thanks to forward checking, the following steps are performed to achieve arc-consistency:

- The instantiation eliminates the value 0 from \( D(x_2) \), \( D(x_3) \), and \( D(x_4) \) due to constraints containing the variable \( x_1 \). The reduced domains are to \( \mathcal{D} = \{\{0\}, \{1\}, \{1, 2\}, \{1, 2, 3\}\}. Forward checking stops after this step. In contrast, MAC keeps going.

- Since the domain \( D(x_2) \) is updated, the pairs \( (x_2, x_3) \), \( (x_2, x_4) \), and \( (x_3, x_4) \) are considered. Due to the constraints between \( x_2 \) and \( x_3 \), and \( x_2 \) and \( x_4 \) the value 1 is eliminated from \( D(x_3) \) and \( D(x_4) \). The reduced domains are \( \mathcal{D} = \{\{0\}, \{1\}, \{2\}, \{2, 3\}\}. \)

- The algorithm continues checking whether any constraint is inconsistent or not. Now only the value in \( D(x_3) \) and the constraint \( x_3 \neq x_4 \) lead to the elimination of the value 2 from \( D(x_4) \). The reduced domains are \( \mathcal{D} = \{\{0\}, \{1\}, \{2\}, \{3\}\}. \)

- Finally, no domain is changed, so the algorithm stops.

Fortunately, in the previous example, the final values of the variables are the unique solution for \( P \): \( x_1 = 0, x_2 = 1, x_3 = 2, x_4 = 3. \)

In general, enforcing arc-consistency cannot prune all possible conflicts. In other words, not all values that remain after performing MAC are necessarily a part of a CSP solution. Nevertheless, one always guarantees that any value that is eliminated from the domain of a variable will not occur in any solution to the CSP.

### 2.1.3 Benchmarks

The CSPs used as the benchmarks in this thesis are widely used in both CSP and SAT communities, particularly in CP-solver competitions [vDRLb, vDRLa] and in regular SAT competitions [SAT].

To present a constraint of a CSP, the definition about the constraint is rather general. In practice, many constraints are defined by intention by means of a combination of simpler constraints \( C_i \) such as:

- Simple \( : C_i \)
- Conditional \( : C_i \rightarrow C_j \)
- Conjunction \( : C_i \land C_j \)
- Disjunction \( : C_i \lor C_j \)
- Cardinality \( : lo \leq \#\{C_1, \ldots, C_n\} \leq up \)

The benchmarks in this thesis will be specified with these intentional linear constraints in a finite domain modelling language similar to the MiniZinc modelling
language [MS]. The command line include "alldifferent.mzn" indicates that the global constraint alldifferent is used by the model.

2.1.3.1 The Pigeon-Hole Problem

For each of \( p \) pigeons, assign one of \( h \) holes, so that each pigeon has its own hole. The problem can be modelled as follows:

```mini
%% in MiniZinc:
% include "alldifferent.mzn";

int : p; % number of pigeons
int : h; % number of holes
array [1..h] of var 1..p: x;

forall(i in 1..p, j in 1..p: i < j)
    constraint (x[i] ≠ x[j]);
%% in MiniZinc:
% constraint alldifferent(x);

solve satisfy;
```

In constraint programming, to speed up solvers the global constraint alldifferent is strongly recommended to use. The problem is unsatisfiable when \( h < p \), this thesis only consider when \( h = p - 1 \). Note that the only constraints used are disequality constraints \( x[i] ≠ x[j] \). Interestingly, the set of disequality constraints has a certain structure in the sense that they represent a global constraint alldifferent. This is a trivially unsatisfiable problem, composed exclusively of constraints of difference on the CSP variables. Figure 2.1 shows an example for the pigeon-hole problem.
2.1. Constraint Programming

This structure is adequately exploited in finite domain solvers by means of specialised global constraints. For instance, graph-based algorithms (see [Régo94]) can propagate very efficiently the constraint \textit{alldifferent}. However, this remains a difficult problem for SAT solvers.

2.1.3.2 The Hamiltonian Cycle Problem

A Hamiltonian cycle is a cycle (i.e., closed loop) in an undirected or directed graph that visits each vertex exactly once, excluding the vertex that is both first and last which is visited twice. Given a directed or undirected graph with \( n \) vertices, specified by an adjacency matrix \( a \). The Hamiltonian cycle problem can be modelled as follows.

\%
% in MiniZinc:
% include "alldifferent.mzn";

int: n; % number of vertices
array[1..n,1..n] of var 0..1: a;
array[1..n] of var 1..n: v;
forall(i in 1..n, j in 1..n: i < j)
  constraint (v[i] ≠ v[j]);
%
% in MiniZinc:
% constraint alldifferent(x);
constraint forall(i in 1..n, j in 1..n & a[i,j]=0)
  (v[i] ≠ v[j]+1);
constraint forall(i in 1..n)(
  if (v[i]==n)
    forall(j in 1..n & a[i,j]=1)
      (#(v[j]=1)=1);
);

solve satisfy;

Figure 2.2 shows an example for the Hamiltonian problem.

The problem of finding a Hamiltonian cycle is \textit{NP}-complete (see prob.GT37 on p.199 of [GJ79]). The problem has many real-life applications, which relate to visiting all locations, such as mail delivery, traveling salesman, garbage pickup, and bus service.

2.1.3.3 The Hidoku Problem

A Hidoku of size \( n \) consists of a grid with \( n \times n \) cells. In each cell there has to be exactly one number between 1 and \( n^2 \). Naturally, corner cells have only three neighbors, and cells on edges have five neighbors, whereas cells in the middle of the grid have eight neighbors. Furthermore, if a cell has value \( z \) (\( 1 \leq z < n^2 \)), then
exact one of its neighboring cells must have value $z + 1$, except for the cell with value $z = \sqrt{n}$. A Hidoku that meets these conditions is said to be valid.

From the rules for a valid Hidoku it also follows that each number appears exactly once on the whole board, because there are $n^2$ cells and due to the neighborhood relationship per next cell the value has to be increased by one. In some cases, some cells may have pre-assigned values. The problem can be modelled as follows (for empty Hidoku):

```plaintext
%%in MiniZinc
% include "alldifferent.mzn";

int: n; % order of grid
array[1..n*n] of var 1..n*n: v;
array[1..n,1..n] of var 0..1: a;

forall(i in 1..n*n, j in 1..n*n: i < j)
  constraint(v[i] ≠ v[j]);
%% in MiniZinc
% constraint alldifferent(v);
constraint forall(i in 1..n*n, j in 1..n*n & a[i,j]==0)
  (v[i] ≠ v[j]+1);

solve satisfy;
```

A valid Hidoku of size 6 is given in Figure 2.3 (taken from [hid]). One can prove that the Hidoku problem is a special case of finding a Hamiltonian path. A SAT encoding for Hidoku was introduced by Hölldobler et al.[HMNS12]. Fig. 2.3 shows an example of a Hamiltonian cycle, taken from [ham].
2.1. Constraint Programming

Figure 2.3: A valid Hidoku of size 6. The bold numbers indicate that they are pre-assigned.

2.1.3.4 The All-Interval Series Problem

The goal of the problem is to arrange a permutation of \( n \) integers ranging from 1 to \( n \) in such a way that the differences between adjacent numbers are also a permutation of the numbers from 1 to \( n-1 \). As a result, the performance of this benchmarks is heavily influenced by the performance of the encoding of the all\( \text{different} \) constraint. The all-interval series problem is one of the classical CSPs and usually regarded as a difficult benchmark to find all solutions (CSPLIB prob007 in [GW99]). Let us consider one solution for \( n = 11 \); differences between adjacent numbers are written underneath the numbers:

\[
\begin{matrix}
1 & 11 & 2 & 10 & 3 & 9 & 4 & 8 & 5 & 7 & 6 \\
10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\
\end{matrix}
\]

The problem is modelled in the MiniZinc modelling language as follows with a note that breaking-symmetry constraints will be addressed in Chapter 6:

```miniZinc
%in MiniZinc
\%
% include "all\(different\).mzn";
%

int: n;
array[0..n-1] of var 0..n-1: s; % vector of numbers
array[1..n-1] of var 1..n-1: v;
% v: a vector of differences between adjacent numbers

constraint all\(different\)(v);
% constraint between v and s
constraint forall(i in 1..n-1)
  (v[i]=abs(s[i]-s[i-1]));
constraint forall(i in 1..n-1, j in 1..n-1: i < j)
  (v[i] ≠ v[j]);
```
in MiniZinc
% constraint alldifferent(v);

\%
breaking-symmetry constraints, fixing the order of the
\%
sequence 0,n–1,1
\%
constraint forall (i in 0..n–2) (s[i]=0<-->s[i+1]=n–1);
\%
constraint forall (i in 0..n–2) (s[i]=n–1<-->s[i+1]=1);
\%
constraint forall (i in 0..n–3) (s[i]=0<-->s[i+2]=1);
\%
solve :: int_search(s, first_fail, indomain_min, complete)
satisfy;

Note that this problem can be modelled by using only disequality constraints
(see Section 6.1.1). In Chapter 6 we study the all-interval series problem in both
the CP approach and the SAT approach.

2.1.3.5 The Quasigroup With Holes Problem

A quasigroup is a square of values \( q_{ij} \), where \( 1 \leq i \leq n \) and \( 1 \leq j \leq n \). Each number
\([1..n]\) occurs exactly once in each row and column. Achlioptas et al. [AGKS00] introduced a method for generating satisfiable quasigroup with holes (QWH) instances in
which some of the \( q_{ij} \) are given. QWH is a NP-complete problem [Col84, AGKS00].

For an \( n \times n \) board, assign one of \( n \) numbers to each of its cells, such that
the same number is not repeated over the columns and the rows. The problem can be
modelled as follows:

\%
in MiniZinc:
% include "alldifferent.mzn";

int: n; % order of the quasigroup.
array[1..n,1..n] of var 1..n: q;

forall(i in 1..n)
  for all(j1 in 1..n, j2 in 1..n)
    constraint(q[i,j1] != q[i,j2]);

forall(j in 1..n)
  for all(i1 in 1..n, i2 in 1..n)
    constraint(q[i1,j] != q[i1,j1]);
\%
in MiniZinc, for two above constraints:
% constraint forall(i in 1..n)(
  % alldifferent(j in 1..n)(q[i,j]) &
  % alldifferent(j in 1..n)(q[j,i])
%)
solve satisfy;
2.1. Constraint Programming

\[
\begin{array}{ccc}
1 & 3 \\
4 & 1 \\
3 & 2 \\
3 & 1
\end{array}
\quad
\begin{array}{ccc}
1 & 2 & 3 & 4 \\
2 & 4 & 1 & 3 \\
3 & 1 & 4 & 2 \\
4 & 3 & 2 & 1
\end{array}
\]

Figure 2.4: An example of Quasigroup Completion Problem of order 4, with 8 holes.

In the problem, some cells have pre-assigned by equality constraints such as \(\text{constraint}(q[i,j] = p)\) for some \(i, j \in 1..n\), and \(p \in 1..n\). Figure 2.4 shows an example for the quasigroup with holes problem.

QWH instances can be considered as a multiple permutation problem in which the variables may occur in more than one permutation [HSW11]. In a permutation the dominating constrains are disequalities of type \(X \neq Y\), where \(X\) and \(Y\) are variables. More specifically, the CSP constraints have some structures, as they represent \textit{alldifferent} constraints on the rows and columns of the quasigroup.

In Chapter 6 we study the quasigroup with holes problem in both the CP approach and the SAT approach.

2.1.3.6 The Graph Colouring Problem

For an undirected graph with \(n\) vertices, specified by an adjacent matrix \(a\), assign one of \(k\) colours to each of the nodes. Graph colouring is the problem of finding the minimum value for \(k\) such that no two adjacent nodes have the same colour. The following model is to check whether the problem is satisfy with \(k\) colours. With a given value \(k\), there are two cases:

1. If the program returns \textit{no}, \(k\) is decreased and the program is called until it returns \textit{yes}. The minimum colour is the current value \(k\).

2. Otherwise, if the program return \textit{yes}, \(k\) is increased and the program is called until it returns \textit{no}. The minimum colour is the current value \(k-1\).

\[
\text{int: vertices; \% number of vertices;}
\text{int: colours; \% number of colours.}
\]

\[
\text{array[1..vertices] of var 1..colours: v;}
\text{array[1..vertices,1..vertices] of var 0..1: a;}
\text{constraint for all(i in 1..vertices, j in 1..vertices:}
\quad \text{i < j & a[i,j]==1})
\quad \text{(v[i] \neq v[j]);}
\text{solve satisfy;}
\]

Fig. 2.5 shows a proper vertex coloring of the Petersen graph with 3 colors. The \textit{graph colouring problem} is a well known NP-complete problem [GJ79]. It has many applications, such as scheduling [Mar04] and register allocation [Cha04].
2.1.3.7 The Round Robin Problem

The goal of the problem is to schedule:

- a given set of \( m \) teams (\( m \) is even) over \( m - 1 \) weeks, with
- each week divided into \( m/2 \) periods, and
- each period divided into two slots, in which the first team in each slot plays at home, whilst the second plays away.

The following constraints must be satisfied:

1. every team plays once a week,
2. every team plays at most twice in the same period, and
3. every team plays with every other team.

The problem can be modelled as follows:

```plaintext
int m; % number of teams
array[1..m-1,1..m] of var (1..m): p;
constraint forall (w in 1..m-1, i in 1..m, j in i..m: i < j)
    & (p[w,i] != p[w,j]);
constraint forall (w1 in 1..m-1, w2 in 1..m-1: w1 != w2)
    forall(i in 1..m/2, j in 1..m/2)
    (var c (0..2) &
     constraint (c = #(|p[w1,i*2-1] = p[w2,j*2-1] & p[w1,i*2-1] = p[w2,j*2],
      & p[w1,i*2] = p[w2,j*2-1],
      & p[w1,i*2] = p[w2,j*2]|)));
solve satisfy;
```
2.1. Constraint Programming

Table 2.1: A schedule for the round robin problem with 8 teams (taken from prob026 in [GW99]).

<table>
<thead>
<tr>
<th></th>
<th>week 1</th>
<th>week 2</th>
<th>week 3</th>
<th>week 4</th>
<th>week 5</th>
<th>week 6</th>
<th>week 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>period 1</td>
<td>1-2</td>
<td>1-3</td>
<td>5-8</td>
<td>4-7</td>
<td>4-8</td>
<td>2-6</td>
<td>3-5</td>
</tr>
<tr>
<td>period 2</td>
<td>3-4</td>
<td>2-8</td>
<td>1-4</td>
<td>6-8</td>
<td>2-5</td>
<td>1-7</td>
<td>6-7</td>
</tr>
<tr>
<td>period 3</td>
<td>5-6</td>
<td>4-6</td>
<td>2-7</td>
<td>1-5</td>
<td>3-7</td>
<td>3-8</td>
<td>1-8</td>
</tr>
<tr>
<td>period 4</td>
<td>7-8</td>
<td>5-7</td>
<td>3-6</td>
<td>2-3</td>
<td>1-6</td>
<td>4-5</td>
<td>2-4</td>
</tr>
</tbody>
</table>

An example of a schedule for the round robin problem is shown in Table 2.1 with 8 teams. The round robin problem has a number of variants, widely used for scheduling tournaments (CSPLIB prob026 in [GW99]). It has been studied for a long time in operations research and computer science communities [RT08]. The problem requires the consideration of both disequality and inequality CSP constraints. Note that its constraints requires a cardinality over disequality constraints, such that no two teams play together twice.

2.1.3.8 The Golomb Ruler Problem

A Golomb ruler (CSPLIB prob006 in [GW99]) *golomb*(n,d) aims at finding a vector \( g \), with \( n \) elements in strictly increasing order with domain \([0..d]\), such that all differences between any two elements are different. In fact, there are \( n(n - 1)/2 \) such differences, that is, \( g_j - g_i \ (1 \leq i < j \leq n) \). Such a ruler is said of order \( n \) and length \( d \). We executed the satisfiable version of the problem to check whether there is a ruler with \( g_n \leq d \) (page 70 at [MS]):

```plaintext
%% in MiniZinc:
% include "alldifferent.mzn"

int: n; % number of marks on ruler
int: d; % max length of ruler
array[1..n] of var 0..d: g;
array[1..n,1..n] of var 0..d: diffs;

constraint g[1] = 0;
constraint forall ( i in 1..n-1 ) (g[i] < g[i+1]);
constraint forall (i,j in 1..n where i > j)
     (diffs[i,j] = g[i] - g[j]);
forall(i in 1..n, j in 1..n: i > j)
     constraint (diffs[i] ≠ diffs[j]);
%% in MiniZinc
% constraint alldifferent(diffs)
%% breaking symmetry:
%% the first difference is less than the last
constraint diffs[2,1] < diffs[n,n-1];
```
solve satisfy;

A Golomb ruler which is able to measure all distances up to its length is called a perfect Golomb ruler. If no shorter Golomb ruler of the same order exists, a Golomb ruler is optimal. An optimal and perfect solution of the Golomb ruler problem of order 4 and length 6 is shown in Figure 2.6.

The Golomb ruler problem requires the consideration of both disequality and inequality CSP. This problem appears in numerous practical applications, for example information theory and error correction [RB67], radio frequency selection [FS77], radio antenna placement [TMGWS08].

2.1.3.9 The Open Shop Scheduling Problem

Given a set of $n$ jobs, each composed of $n$ tasks with duration $d$ can be performed in $n$ machines, such that the $n$ tasks of each job execute in different machines. Assume that each machine can handle one task at a time. The goal of the problem is to find starting times for each of the tasks, such that they are all finished by a certain deadline $t$ (the makespan). It is worth noting that there is no ordering constraints on operations. The problem can be modelled as follows:

```plaintext
int t, n; t1=t+1  % makespan, number of jobs.
array[1..n,1..n] of var 0..t: d; % matrix of durations
array[1..n,1..n] of var 1..n: m; % matrix of machines
array[1..n,1..n] of var 1..t: s; % matrix of start times
array[1..n,1..n] of var 1..t: e; % matrix of finish times
constraint forall(i in 1..n, j in 1..n)
    (e[i,j] == s[i,j] + d[i,j]);
constraint forall(i in 1..n, j in 1..n) (e[i,j] <= t1);
constraint%% ensure no overlap of tasks
    forall(i in 1..n,j in 1..n, p in 1..n where p != i,
        q in 1..n where m[i,j] == m[p,q])
        (e[i,j] <= s[p,q] \/ e[p,q] <= s[i,j]);
```
Figure 2.7: An illustration of the open shop scheduling problem with 3 machines.

A model for the open shop scheduling problem with 3 machines is shown in Figure 2.7. Note that this problem is mostly composed of disjunctions of linear inequalities.

2.1.3.10 The Langford Problem

Given a value of $n$, a Langford sequence is a permutation of the sequence of $2 \times n$ numbers $1, 1, 2, 2, ..., n, n$, in which the two $1$s are one unit apart, the two $2$s are two units apart, and more generally the two $k$s are $k$ units apart ($1 \leq k \leq n$).

The Langord problem (CSPLIB prob024 in [GW99]) is the task of constructing a Langord sequence. The problem can be modelled as follows:

```plaintext
include "alldifferent.mzn";

int: n;
set of int: positionDomain = 1..2*n;
array[positionDomain] of var positionDomain: position;
array[positionDomain] of var 1..n: solution;

constraint forall(i in 1..n) (  
  position[i+n] = position[i] + i+1  
  /
  solution[position[i]] = i  
  /
  solution[position[n+i]] = i  
);
constraint all_different(position);

solve satisfy;
output [ show(solution), "\n" ];
```
Figure 2.8 shows a solution for the Langford problem with $n=4$.

![Diagram of a solution for the Langford problem with n=4.]

The problem has an exponential number of solutions increasing roughly as $(\frac{44}{e})^n$ [Mil06]. Unsurprisingly, there are many instances consuming exponential time, for example, an instance $n = 19$ needs 2.5 years and the latest solution for $n = 24$ consuming three months of computation with 12 to 15 processors, giving approximately $4.7 \times 10^{16}$ solutions by Jaillet and Krajecki (see [JK04] and [Mil06]). Finding all Langford sequences has been a real challenge for combinatorial search.

In Chapter 6 we study the Langford problem in both the CP approach and the SAT approach.
2.2 Boolean Satisfiability

We follow and use the notations from [HÖ9].

2.2.1 Syntax of Propositional Logic

A syntax comprises of rules (or grammar) used for constructing words of a language or formulas without regard to any meaning given to them. We start by defining an alphabet of propositional logic.

**Definition 2.2.1** An alphabet of propositional logic consists of

- a (countably) infinite set \( \mathcal{R} \) of propositional variables,
- the unary symbol \( \neg \); four binary symbols of connectives: \( \land \) (conjunction, and), \( \lor \) (disjunction, or), \( \rightarrow \) (implication), \( \leftrightarrow \) (equivalence), and
- the special characters "(" and ")".

Different alphabets of propositional logic differ in \( \mathcal{R} \) and, hence, alphabets are usually specified by specifying \( \mathcal{R} \). From now on, the symbol \( n \) at \( o/n \) is the arity of the connective \( o \). In the context of SAT, \( \mathcal{R} \) is \( \mathbb{N}^+ \).

**Definition 2.2.2** (Atom) An atomic formula, briefly called atom, is a propositional variable.

**Definition 2.2.3** (Propositional formula) The set of propositional formulas is the smallest set \( \mathcal{L}(\mathcal{R}) \) of strings over an alphabet \( \mathcal{R} \) of propositional logic with the following properties:

- If \( F \) is an atomic formula, then \( F \in \mathcal{L}(\mathcal{R}) \)
- If \( F \in \mathcal{L}(\mathcal{R}) \), then \( \neg F \in \mathcal{L}(\mathcal{R}) \)
- If \( o/2 \) is a binary connective, \( F, G \in \mathcal{L}(\mathcal{R}) \), then \( (F \circ G) \in \mathcal{L}(\mathcal{R}) \)

**Definition 2.2.4** (Literal) A literal is an atom, or a negated atom.

Sometimes we need the complement of a literal \( L \) (denoted \( \bar{L} \)), which is defined as follows:

- If \( L \) is an atom \( A \), then \( \bar{L} = \neg A \),
- if \( L \) is a negated atom \( \neg A \), then \( \bar{L} = A \).

A pair \( L \) and \( \bar{L} \) of literals is called complementary.

**Definition 2.2.5** (Clause) A clause \( C \) is a generalized disjunction \( [L_1, \ldots, L_n], n \geq 0 \), where every \( L_i, 1 \leq i \leq n \), is a literal.

**Definition 2.2.6** (Formula in CNF) A formula is in conjunctive normal form (CNF) if and only if it is of the form \( (C_1, \ldots, C_m), m \geq 0 \), where every \( C_j, 1 \leq j \leq m \), is a clause.

This thesis restricts propositional formulas in CNF.
2.2.2 Semantics of Propositional Logic

In contrast to syntax, the semantics studies how to assign meaning to a language (or formulas). The set of truth values is the set \{1, 0\}. The meaning for the functions \neg, \land, \lor, \to, \leftrightarrow are given by the following truth table (Table 2.2):

<table>
<thead>
<tr>
<th>(p)</th>
<th>(q)</th>
<th>(\neg p)</th>
<th>(p \land q)</th>
<th>(p \lor q)</th>
<th>(p \to q)</th>
<th>(p \leftrightarrow q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Now a valuation for general formulas is defined through an assignment or an interpretation.

**Definition 2.2.7 (Interpretation)** Let \(F, G, G_1,\) and \(G_2\) be propositional formulas (i.e., \(\in \mathcal{L}(\mathcal{R})\)). An interpretation \(I = (\{0, 1\}, \cdot, : \mathcal{L}(\mathcal{R}) \to \{1, 0\})\) consists of the set \(\{0, 1\}\) and a mapping \(\cdot, : \mathcal{L}(\mathcal{R}) \to \{1, 0\}\) with:

\[
[F]_I = \begin{cases} 
  w \in \{0, 1\} & \text{if } F \text{ is a propositional variable,} \\
  \neg[G]_I & \text{if } F \text{ is of the form } \neg G, \\
  ([F]_I^1 \circ [G]_I^2) & \text{if } F \text{ is of the form } (G_1 \circ G_2). 
\end{cases}
\]

**Definition 2.2.8 (Model)** An interpretation \(I\) for a propositional formula \(F\) is called a model for \(F\), in symbols \(I \models F\), if \([F]_I = 1\).

**Definition 2.2.9** Let \(F\) be a propositional formula.

- \(F\) is **satisfiable** iff there is a model for \(F\)
- \(F\) is **unsatisfiable** iff there is no model for \(F\)
- \(F\) is **valid** iff all interpretations for \(F\) are models for \(F\)
- \(F\) is **falsifiable** iff some interpretations for \(F\) are not models for \(F\)

**Definition 2.2.10 (Equisatisfiable)** Given \(F, G \in \mathcal{L}(\mathcal{R})\), \(F\) and \(G\) are equisatisfiable if either both formulas are satisfiable or both are not.

**Definition 2.2.11** An interpretation \(I\) is called model for a set \(\mathcal{G}\) of formulas (\(I \models \mathcal{G}\)) iff \(I\) is a model for all \(F \in \mathcal{G}\).

**Definition 2.2.12 (Semantically equivalent)** Two propositional formulas \(F\) and \(G\) are semantically equivalent, in symbols \(F \equiv G\), iff for all interpretations \(I\) we have: \(I \models F\) iff \(I \models G\).
2.2. Boolean Satisfiability

**Theorem 2.2.13** There are some equivalence laws:

\[ \lnot \lnot F \equiv F \]  \hspace{1cm} \text{Double negation}

\[ F \land F \equiv F \]  \hspace{1cm} \text{Idempotence}

\[ F \lor F \equiv F \]

\[ F \land G \equiv G \land F \]  \hspace{1cm} \text{Commutativity}

\[ \lnot (F \lor G) \equiv \lnot F \land \lnot G \]  \hspace{1cm} \text{de Morgan I}

\[ \lnot (F \land G) \equiv \lnot F \lor \lnot G \]  \hspace{1cm} \text{de Morgan II}

\[ F \land (G \lor H) \equiv (F \land G) \lor (F \land H) \]  \hspace{1cm} \text{Distributivity}

\[ F \lor (G \land H) \equiv (F \lor G) \land (F \lor H) \]

\[ F \land (G \land H) \equiv (F \land G) \land H \]  \hspace{1cm} \text{Associativity}

\[ F \lor (G \lor H) \equiv (F \lor G) \lor H \]

\[ (F \equiv G) \equiv (F \land G) \lor (\lnot F \land \lnot G) \]  \hspace{1cm} \text{Equivalence}

\[ F \land (F \lor G) \equiv F \]  \hspace{1cm} \text{Absorption}

\[ F \lor (F \land G) \equiv F \]

\[ (F \rightarrow G) \equiv (\lnot F \lor G) \]  \hspace{1cm} \text{Implication}

\[ (F \lor G) \equiv F, \text{ if } F \text{ is valid} \]

\[ (F \land G) \equiv G, \text{ if } F \text{ is valid} \]  \hspace{1cm} \text{Tautology}

\[ (F \lor G) \equiv G, \text{ if } F \text{ is unsatisfiable} \]

\[ (F \land G) \equiv F, \text{ if } F \text{ is unsatisfiable} \]  \hspace{1cm} \text{unsatisfiability}

2.2.3 Satisfiability

**Definition 2.2.14 (SAT)** A propositional satisfiability problem (abbreviated as SAT) consists of a formula F (in CNF), and is the problem to decide whether F is satisfiable.

**Theorem 2.2.15 (Cook’s Theorem [Coo71])** SAT is NP-complete.

SAT is the first problem shown to be NP-complete by Stephen Cook [Coo71]. Cook proved that every decision problem in the complexity NP-complete class can be polynomially reduced to the SAT problem. As a result, if there is a polynomial time algorithm that decides SAT, then there is a polynomial time algorithm for all other NP-complete problems, and vice versa. Intuitively, SAT is the “hardest” problem
in the NP-complete class. In computer science, SAT features a central role in the theories of complexity and computation.

2.2.4 SAT with CNF

In order to fulfill the purpose of this thesis, encoding finite CSPs into CNF, we provide here some essential knowledge of CNF in the context of SAT solving.

2.2.4.1 Why CNF?

Generally, Boolean satisfiability (abbreviated as SAT) is used to refer to the problem of finding a satisfying assignment for any Boolean formula. One can divide two types of format for SAT solving: CNF and non-CNF. During the CNF encoding, information of the structure of the problem is lost since CNF is flat and homogeneous. In order not to lose a great deal of internal structure by converting a Boolean formula into CNF, several non-CNF formats have been reported in literature [Sta02, TBW04, MS06, PTS07]. Nevertheless, CNF has been the most intensively studied format compared to any others and widely accepted as a standard format for SAT solvers [SAT].

In fact, testing a given formula in CNF for being falsifiable is trivial, however, for being satisfiable is NP-complete. The interesting question is “Why is CNF considered as a standard format for SAT solver?” The main reasons might be as follows.

- Firstly and most importantly, any Boolean formula can be transformed into a equisatisfiable CNF formula in linear time and space by introducing extra variables [Tse83].

- Secondly, in contrast to diverse input formats in CSP solving, most of the effort of the SAT community focuses on a single representation (i.e., CNF). Consequently, the format of CNF has encouraged a great deal of research into highly optimized data structures (e.g., two-watched-literal scheme), efficient algorithms and simple implementations.

- Thirdly, CNF allows SAT solvers to directly represent learned clauses which may let the solvers determine a certain level for backjumping. This reason may lead to the fact that representation of learned clauses is now standard in SAT solvers, whereas presenting learned constraint (typically called no-good in CSPs) is much less obvious in CP solvers.

2.2.4.2 Transforming a Propositional Formula into CNF

The following theorem plays a prominent role for SAT encodings since it is one of the most important reasons why almost all SAT solvers use CNF as the input format.

**Theorem 2.2.16** There is an algorithm which transforms any propositional formula into a semantically equivalent formula in CNF.
2.2. Boolean Satisfiability

Here are important steps for transforming an arbitrary Boolean formula $F$ into a corresponding CNF.

**Input:** A given propositional formula $F$.

**Output:** An equivalent formula in CNF.

1. Eliminate the equivalences and implications in F:
   
   $F \equiv G$ by $(F \rightarrow G) \land (G \rightarrow F)$ (Equivalence)

   Note that this substitution may lead to exponential blow-up formula.

   $F \rightarrow G$ by $\neg F \lor G$ (Implication)

2. Substitute the resulting formula, in step 1, every subformula of the form in F:
   
   $\neg \neg F$ by $F$ (double negation)

   $\neg (F \lor G)$ by $\neg F \land \neg G$ (de Morgan I)

   $\neg (F \land G)$ by $\neg F \lor \neg G$ (de Morgan II)

   The goal of step 2 is to transform $F$ into *negation-normal form* where all negation symbols occur only in front of variables.

3. Substitute the resulting formula, in step 2, every subformula of the form in F by using the distributivity rule:
   
   $F \land (G \lor H)$ by $(F \land G) \lor (F \land H)$

   $F \lor (G \land H)$ by $(F \lor G) \land (F \lor H)$

**Example 2.2.17**

\[
F = (r \land (p \rightarrow q)) \rightarrow t \\
\equiv \neg(r \land (\neg p \lor q)) \lor t \quad \text{step 1} \\
\equiv (\neg r \lor (p \land \neg q)) \lor t \quad \text{step 2} \\
\equiv ((\neg r \lor p) \land (\neg r \lor \neg q)) \lor t \quad \text{step 3} \\
\equiv (\neg r \lor p \lor t) \land (\neg r \lor \neg q \lor t) \quad \text{step 3}
\]

As a result, only connectives $\neg$, $\land$, and $\lor$ occur in the obtained CNF formula.

### 2.2.4.3 Tseitins Polytime Transformation to CNF

As mentioned before, the substitution in step 1 of the previous conversion algorithm may lead to a combinatorial explosion.

**Example 2.2.18** Let us consider the DNF formula

\[
F = (y_1 \land z_1) \lor (y_2 \land z_2) \lor \cdots \lor (y_n \land z_n),
\]

where $y_i$ and $z_i$ are Boolean variables. Then, every logically equivalent formula in CNF includes $2^n$ clauses.

To avoid the explosion, Tseitin proposed a method [Tse83] that introduces new variables to replace subformulas along with clauses to guarantee the relationships between these new variables and the subformulas. The resulting CNF maintains satisfiability, that is, the new formula is satisfiable if and only if the original formula is satisfiable. Moreover the new formula has a linear number of clauses. Let us see how Tseitin’s transformation works through Example 2.2.18:
1. Introduce \( n \) new variables \( x_i, 1 \leq i \leq n \), such that \( x_i \leftrightarrow (y_i \land z_i) \)

2. Add \( 3n \) clauses \( (\neg x_i \lor y_i), (\neg x_i \lor z_i) \) and \( (\neg y_i \lor \neg z_i \lor x_i) \) for all \( 1 \leq i \leq n \)

3. The formula \( F \) now becomes \( (x_1 \lor x_2 \lor \cdots \lor x_n) \)

As a consequence, the resulting CNF formula consists of \( 3n \) variables and \( 3n + 1 \) clauses, instead of \( 2n \) variables and \( 2^n \) clauses, respectively.

### 2.2.4.4 Representation of CNF for SAT

The DIMACS format, followed by the center for Discrete Mathematics and Computer Science since 1993 [JT96], has been widely accepted as the standard format for SAT, in order to write a Boolean formulas in conjunctive normal form (CNF). The format file in an ASCII file, which has been used as SAT benchmark problems for the regular SAT solver competitions, consists of a two major sections: the preamble and the clauses.

1. **The preamble** contains information about the instance, shown in lines. Each line starts with a single character, followed by a space. There are two types of lines:
   - Comments - these lines are optional and they must begin with a lower case character \( c \):
     
     \[ c \text{ This is an example of a comment line.} \]
   - Problem line - unlikely comment lines, there is only one problem line per input file. The problem line must appear before any other content and has the following format:

     \[ p \text{ format variables clauses} \]

     where the lower-case \( p \) indicates that this is the problem line; the **format** specifies that format, which is expected, for SAT it is the word “cnf”; the **variables** are an integer value specifying the number of variables in the instance; the **clauses** are also an integer value specifying the number of clauses in the instance;

2. The **clauses** are separated into lines. If the variables are assumed to be numbered from 1 up to \( n \), then each clause is a sequence of distinct numbers between \(-n \) and \( n \), each separated by either a space or a tab, ending with 0 on the same line. Positive numbers denote the corresponding variables. Negative numbers denote the negations of the corresponding variables.

Let us consider an example of CNF formula:

\[ (x_1 \lor x_2 \lor \neg x_4) \land (\neg x_2 \lor x_3 \lor x_5) \land (\neg x_1 \lor \neg x_3). \]

Replacing ground atoms by nature numbers, the formula can be represented by the DIMACS format as follows.
2.2. Boolean Satisfiability

c comment lines begin with a character c
c start with
c comments

p cnf 5 3
 1  2 -4  0
-2  3  5  0
-1 -3  0

2.2.5 Conflict-Driven Clause Learning SAT Solvers

Many techniques for improving the efficiency of SAT solvers have been investigated and evaluated for the last two decades. State-of-the-art SAT solvers are conflict-driven clause learning (CDCL) SAT solvers, which are themselves based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure [DP60, DLL62]. Since many important recent applications of SAT (e.g., hardware verification or product configuration) significantly rely on these solvers, this section briefly provides the key techniques of the success of CDCL SAT solvers. We refer the reader to [SS96, MMZ’01, ES05, SLM09, Man14] for the best reference here.

A Brief Overview of CDCL SAT Solvers The pioneering techniques of SAT solvers introduced in the early 1960s were the DPLL procedure [DP60, DLL62], consisting of of three major techniques: branching, unit propagation, and chronological backtracking. The SAT solvers had no remarkable improvements for almost 40 years until the appearance of clause learning and non-chronological backtracking techniques, first proposed by Marques-Silva and Sakallah in GRASP [SS96], and independently by Bayardo and Schrag in RelSAT [JS97]. Furthermore, the CDCL SAT solvers have not fully reached the high level of efficiency until the outcome of the well-designed solver Chaff by Moskewicz et al. [MMZ’01]. The authors proposed three major contributions: a lazy data structure - the watched literals, the variable state independent decaying sum (VSIDS) branching heuristic, and the first unique implication point (1UIP) backtracking scheme. These features are the heart of all search-based modern SAT solvers. It is worth noticing that Minisat by Sörensson and Eén is a minimalistic implementation of a Chaff-like SAT solver [ES03, ES05], which is not only a small well-structured, but also a high performance solver. It performs learning by following the exact same steps as proposed in GRASP and additionally employs the conflict clause minimization technique. Minisat has been considered as a tutorial and as an excellent starting point for researchers wishing to modify it for their purposes. Minisat has been one of the driving forces in the development of efficient solvers in the regular international SAT competition [min]. Most of the currently modern SAT solvers have been based on Minisat. This thesis uses three solvers: Riss3G [Man13] (SAT competition 2013 version), Lingeling [Bie13] (SAT competition 2013 version) and Clasp [GKS09] (clasp2.1.3x86_64linux). These three state-of-the-art CDCL SAT solvers were ranked best on application and craft benchmarks in different categories at recent SAT competitions [SAT].
Unit Propagation  Unit propagation (UP), also called Boolean constraint propagation, is a procedure of automated theorem provers that simplifies formulas. The procedure is based on unit clauses, i.e., clauses contain only a single literal. Then, simplifying a formula consisting unit clauses is repeatedly performed by two following steps:

1. If a unit clause, say \([L]\) appears in the CNF, then literal \(L\) is set to \(I\) to satisfy the clause, consequently, any clause containing literal \(L\) is removed from the formula.

2. Literal \(\neg L\) appearing in any clause are deleted (i.e., \(\neg L \equiv 0\)).

The formula is simplified by UP and checked for either success (the empty formula is derived), meaning that all clauses are satisfied, or failure (an empty clause is derived), meaning a dead end.

UP is one of the three major techniques which were introduced with the debut of the DPLL procedure [DP60, DLL62]. Moskewicz et al. [MMZ+01] observed that 80% of execution time of search-based SAT solvers is spent on performing unit propagation [MMZ+01]. Hence, an efficient implementation of unit propagation plays a critical role in the success of the SAT solver.

2.3 CSP and SAT

2.3.1 SAT is a CSP

SAT can be formally defined in terms of CSPs as follows.

**Definition 2.3.1 (SAT)** The satisfiability problem (SAT) is a triple \((\mathcal{V}, \mathcal{D}, \mathcal{C})\)

\[
\mathcal{V} := \{x_1, x_2, \ldots, x_n\} \text{ is a set of } n \text{ Boolean variables,}
\]

where: \(\mathcal{D} := \{0, 1\}\) is called the domain of SAT used by \(n\) variables,

\(\mathcal{C} := \{C_1, C_2, \ldots, C_m\}\) is a set of \(m\) clauses.

The goal of SAT is to determine whether there exists an assignment of a value to each Boolean variable that satisfies all the clauses.

2.3.2 Encoding a CSP to SAT

It has been observed that if the problem \(P'\) is more efficient to solve than problem \(P\) and there exists a polynomial translation of \(P\) to \(P'\), then one can solve \(P\) through \(P'\). For example, in order to benefit from the steadily increasing diffusion and availability of SAT solvers, a wide rage of practical problems (problem \(P\)) have been encoded to SAT instances (problem \(P'\)) and solved efficiently.

To translate a CSP to a SAT problem, one needs an encoding \(\mathcal{E}\) that maps sets of variables, domains, and constraints of the CSP to corresponding SAT clauses:

\[
\mathcal{E} : P \longrightarrow SAT
\]
2.4. Discussion

This encoding must guarantee that any model $\phi$ of the CSP can be extended to a model $E(\phi)$ of SAT, and any model $E(\phi)$ of SAT can be decoded to a model $\phi$ of the original CSP.

2.4 Discussion

The Boolean satisfiability problem (SAT) and constraint satisfaction problems (CSPs) have been studied as two relatively independent topics, although, they share a lot of features in common with respect to the approach used for solving a given problem and the algorithm used by the solvers. As a result, the relationship between SAT and CSPs has been studied by many authors. Walsh performed a comprehensive study of the connection between them [Wal00]. Bordeaux et al.[BHZ06] made an informative survey providing a broad overview of two areas in a comparative way. The similarities and differences between CSP and SAT are compared and contrasted. Furthermore, the authors addressed a wide range of topics, ranging from philosophy, applications, modeling to algorithms and specific techniques. With the insights given in this survey, it is observed that the integration of the two areas can probably profit from the improvements of each area. This survey contributes to bridging the gap between CSPs and SAT.

It has been observed that few researchers have been interested in the translation of SAT into a CSP. In 1996, Bennaceur studied the translation of SAT problem as CSP [Ben96]. Furthermore, he established a comparison between SAT and CSP techniques in [Ben04]. Particularly, through the translation of SAT instances into binary CSPs, Bennaceur pointed out that the arc and path consistency concepts can be expressed in terms of logical inference rules [Ben04]. He also showed the relationships between the behavior of the DPLL procedure in SAT and maintaining arc-consistency (MAC) in CSPs for solving a SAT instance. Walsh compared the impact of achieving some level of consistency during the search (e.g., forward checking and arc-consistency) on the corresponding CSP with the performance of unit propagation in the DPLL procedure on the respective SAT instance [Wal00].

On the one hand, an increasing number of practical applications in computer science can be expressed as CSPs. On the other hand, in order to benefit from the tremendous advances in the highly optimized and remarkably efficient SAT solvers, one must first encode the original problem to an equivalent SAT formula. Consequently, numerous studies of SAT encodings of finite CSPs have been recently reported [GJ96, Hoo99, Wal00, BB03, Pre03b, Pre03a, AdVD+04, BB04, FPDN05, Gav07, Vel07a, GARK07, TTKB09, Pre09, Zha09, TTB11, LZMS11, PJ11, JP12, BHN14a, BHN14b].

The next chapters of this thesis will investigate how to encode a finite CSP to SAT and provide interesting guidelines, regarding the choice of suitable SAT encodings for CSPs, so that solvers can efficiently deal with the resulting instances.
 CHAPTER 3

SAT Encodings of Finite CSP Domains

It is intensively observed that modelling is far better understood for CSPs than for SAT. This chapter provides a novel approach on how to encode a finite CSP into a SAT instance in a comprehensive and informative way. This chapter, encoding CSP domains, and the next chapter, encoding CSP constraints, could be regarded as a foundation study of SAT encodings of finite CSPs. In the context of this thesis, for convenience, the terms translating, encoding, formulating, and modelling are used interchangeably.

Following the formal definition 2.1.3 in Chapter 2, CSP consists of a triple:

- $\mathcal{V}$, the finite set of variables with
- $\mathcal{D}$, the set of corresponding domains, and
- $\mathcal{C}$, the set of constraints.

SAT encoding of a CSP consists of encoding of CSP variables with their corresponding domains and encoding of CSP constraints. Throughout this thesis, for the sake of simplicity, the term encoding of finite CSP domains is used to mean encoding of CSP variables with their corresponding domains.

Although translating a CSP to a SAT instance has been investigated by numerous researchers [Wal00, Gen02, Pre03a, Pre04b, FPDM05, Gav07, Vel07a, Gel08, TTKB09, ACLM12], to the best of our knowledge we are unaware of any literature that clearly classifies SAT-encoding schemes based on encoding CSP domains and CSP constraints separately. Nevertheless, several reasons inspire us to do so:

1. It has been widely observed that the translation of finite CSP domains has a more significant influence on the performance of SAT solvers than the encoding of CSP constraints. For example, there is a slight difference in performance between the direct encoding [dIK89, Wal00] and the support encoding [Kas90, Gen02], which use the same encoding of finite CSP domains but different encodings of a CSP constraint (i.e., one uses conflicts clause whereas the other uses support clauses, see Section 4.1). However, there is a considerable difference in performance among the direct encoding (or the support encoding), the order encoding, and the log encoding, which use different encoding of CSP domains but the same encoding of a CSP constraint.
2. There are different SAT encodings that use the same method of encoding CSP domains, but they use different methods of encoding CSP constraints. This may cause confusion. For example, it is ambiguous whether the order encoding and the log encoding use either the support clause or the conflict clause (in Chapter 4).

3. Each SAT encoding of CSP domains can combine with any SAT encoding of CSP constraints. Hence, considering them separately allows the SAT community to easily study their integration.

4. This classification can make SAT encodings much more understandable and comprehensive.

Although this chapter focuses on SAT encodings of a CSP domain, when a SAT encoding is mentioned to encode a CSP one should understand that this encoding uses the conflict clause to encode a CSP constraint, unless otherwise noted.¹

Figure 3.1 shows the process of encoding a CSP into SAT, including the corresponding chapters for each part. As we can see, this chapter aims at comprehensively presenting all well-known SAT encodings of a finite CSP domain to SAT clauses. When encoding a finite domain, SAT encodings have different performance profiles and achieve different degrees of consistency level through unit propagation performed by SAT solvers.

Chapter 4 provides SAT encodings of CSP constraints, whereas Chapter 5 empirically provides the result among SAT encodings, which are studied in Chapters 3 and 4.

From now on, the terms constraint and variable are used to refer to a CSP constraint and CSP variable, respectively. The terms propositional variables, Boolean variables, and SAT variables are used interchangeably. A SAT formula is always in CNF. Without loss of generality, let a finite domain of each CSP variable \( V \in \mathcal{V} \) be the set of constants \( \{1, 2, \ldots, n\} \). Our goal is to encode each variable \( V \) with its domain into CNF.

¹To encode CSP constraint, conflict clauses or support clauses can be used, see Section 4.1.
3.1 Sparse Encoding

The term “sparse” used by Hoos [Hoo99] refers to the SAT encoding that was introduced by Kleer [dK89] and was commonly called the direct encoding by Walsh [Wal00]. This encoding is the most straightforward way to transform a CSP into a SAT instance. Hereby, we use the sparse encoding as denoted in [JP12], and adopted among others by: 1) the direct encoding [dK89, Wal00]; and 2) the support encoding, first investigated by Kasif [Kas90] and further studied by Gent [Gen02]. Note that the direct encoding and the support encoding use the same way of encoding a finite CSP domain, but different ways of encoding a constraint to SAT, which will be presented in Chapter 4.

A Boolean variable $d_i^V$ is assigned to 1 if and only if the variable $V$ takes the value $i, 1 \leq i \leq n$, from its domain $\{1, \ldots, n\}$, therefore, $d_i^V = 0$ when $V$ is not assigned to $i$. To guarantee one CSP variable is assigned exactly one value, in the corresponding SAT formula, the sparse encoding requires two sets of clauses, the at-least-one (ALO) and the at-most-one (AMO) clauses. We call ALO and AMO clauses the domain constraint for the sparse encoding.
• To guarantee that each CSP variable must be assigned to at least one value in its domain (i.e., the ALO constraint), the \textit{at-least-one} (ALO) clauses are included in SAT instances:

\[ d^V_1 \lor d^V_2 \lor \cdots \lor d^V_n \]

• To guarantee that each CSP variable can not take more than one domain value (i.e., the AMO constraint), the \textit{at-most-one} (AMO) clauses also need to be added. Several AMO encodings have been studied, here we use the most simple and widely-used, the \textit{AMO pairwise encoding}\(^2\):

\[
-\neg d^V_1 \lor -\neg d^V_2 \lor -\neg d^V_3 \ldots -\neg d^V_1 \lor -\neg d^V_n \\
-\neg d^V_2 \lor -\neg d^V_3 \ldots -\neg d^V_2 \lor -\neg d^V_n \\
\vdots \\
-\neg d^V_{n-1} \lor -\neg d^V_n 
\]

The above encoding generates \( \frac{n(n-1)}{2} \) binary clauses to encode the AMO constraint. In fact, there are several other ways which need only \( O(n) \) clauses (see [FG10, HN13b]). Section 4.2 will study these encodings intensively.

Whereas many SAT encodings share the same method of translating each finite domain into SAT, these encodings differ in their method of translating CSP constraints. For example, the \textit{support encoding} [Gen02] and the \textit{direct encoding} [dK89, Wal00].

It is worth mentioning that many variants of the \textit{sparse encoding} may omit the AMO clauses [SLM92, Pre03a, Pre04a, Pre04b, Vel07a]. However, then there is no equivalence between SAT and CSP solutions, which will be further discussed later in Section 3.7. Interestingly, Jeavons and Petke [JP12] recently showed that the modern SAT solvers (i.e., using unit propagation, clause learning with highly-tuned learning schemes, branching strategies and restart policies) solve particular families of CSPs far more efficiently than conventional CP solvers by using the \textit{sparse encoding} (i.e., the \textit{direct} and the \textit{support encodings}) to translate these CSPs into SAT instances. Additionally, the authors pointed out that state-of-the-art SAT solvers are able to decide the \textit{satisfiability} of the \textit{sparse encoding} of any CSP instance with bounded width in expected polynomial-time [JP12].

The number of SAT variables in the \textit{sparse encoding} is linear with respect to the corresponding CSP domain size. One major drawback of these variables is that they are very penalizing for large domains, consequently they consumes much runtime of SAT solvers.

\(^2\)To differ the way of encoding a finite domain to SAT and the way of encoding a AMO constraint, we use term “SAT encoding” for the former and term “AMO SAT encoding” for the latter.
3.2 Order Encoding

The order encoding has been well-studied in [CB94, AM04, BB03]. For example, it was used to encode cardinality constraints [BB03, Bai10] into SAT before Tamura et al. [TTKB09] named the order encoding and used it for an efficient SAT-based constraint solver, Sugar. The similarity among the order encoding, sequential counter encoding [Sin05], regular encoding [ACLM09], and several others will be presented at the end of this chapter (also see [HN13b, BHN13]).

The order encoding represents a CSP variable V with domain \{1, \ldots, n\} by a vector of n − 1 Boolean variables \([o_1^V, \ldots, o_{n-1}^V]\). To specify the CSP assignment \(V = i\) the first \(i - 1\) Boolean variables are assigned to 1 and the remaining to 0, i.e., as non-increasing vector \([o_1^V, \ldots, o_{n-1}^V]\). This specification is called the domain constraint for the order encoding that consists of a set of binary clauses.

\[
\bigwedge_{i=1}^{n-2} [o_{i+1}^V \rightarrow o_i^V]
\]

which is semantically equivalent to

\[
\bigwedge_{i=1}^{n-2} [-o_{i+1}^V \vee o_i^V].
\]

The above clauses guarantee the desired properties [SL07]:

- if \(o_i^V = 1\), then \(o_j^V = 1\) for all \(1 \leq j \leq i \leq n - 1\),

- if \(o_i^V = 0\), then \(o_j^V = 0\) for all \(1 \leq i \leq j \leq n - 1\).

A CSP assignment \(V = i\) is modelled by imposing \(o_{i-1}^V = 1\) and \(o_i^V = 0\), whereas its negation \(V \neq i\) is represented by \(o_{i-1}^V = o_i^V\) [MC12] (to cope with \(V = 1\), we assume an extra bounding variable \(o_0^V = 1\)).

According to [BB03] the main advantage of this encoding is in the representation of interval domains and the propagation of their bounds. Indeed, the value of \(V\) may be restricted to a range \([i..j]\), by setting \(o_{i-1}^V = 1\) and \(o_j^V = 0\).

As an example of encoding a finite domain, Table 3.1 shows the Boolean variables used by both SAT encodings. Note that the Boolean variable satisfy the ALO and AMO clauses in the sparse encoding, and the domain constraint for the order encoding.

Interestingly, Petke and Jeavons [PJ11] showed that one can use the order encoding to translate various tractable CSP into tractable instances of SAT. In other words, by using the order encoding one can solve several SAT-encoded tractable CSPs in polynomial time.

The number of SAT variables in the order encoding requires linear number with respect to its domain size. A similar drawback to the sparse encoding, these variables are very penalizing for large domains, as a result they remarkably consume the runtime of SAT solvers.
Table 3.1: An illustration of the sparse (Sp) and order (Or) encodings with a variable $V$ with domain $\{1, ..., 6\}$ (i.e., $n=6$), where a “$\cdot$” indicates an undefined value (either 0 or 1). The Boolean variables assigned to $Y$ mean they have the same value, even though they are undefined yet (either 0 or 1). The sets of variables $\{d_1^V, \ldots, d_6^V\}$ and $\{o_1^V, \ldots, o_6^V\}$ satisfy the domain constraints for the sparse and the order encodings, respectively.

<table>
<thead>
<tr>
<th>Variable</th>
<th>$d_1^V$</th>
<th>$d_2^V$</th>
<th>$d_3^V$</th>
<th>$d_4^V$</th>
<th>$d_5^V$</th>
<th>$d_6^V$</th>
<th>$o_1^V$</th>
<th>$o_2^V$</th>
<th>$o_3^V$</th>
<th>$o_4^V$</th>
<th>$o_5^V$</th>
<th>$o_6^V$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V = 1$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V = 3$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V \neq 3$</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$V = 6$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$3 \leq V \leq 5$</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

3.3 Log Encoding

Although the performance of SAT solvers depends on many factors, it is most influenced by the number of variables, the number of clauses, and the length of clauses (e.g., unit clauses or binary clauses). Therefore, it is a good idea to study an encoding that not only easily translates a CSP to a SAT instance, but also needs as few SAT variables as possible. Hopefully the potential search space for SAT solvers can be reduced. The log encoding is such an encoding.

The log encoding, first proposed by Iwana and Miyazaki [IM94] to encode Hamiltonian circuit, k-Clique, and colouring problems to SAT, has many different names. It is called the compact encoding by Hoos [Hoo99]; the bitwise representation by Millstein and Weld [EMW97]; the circuit-based encoding by Gelder [Ge08]. However, the most common name is the log encoding by Walsh [Wal00].

Instead of introducing $n$ propositional variables to encode a finite CSP variable $V$ with $n$ domain values, as the sparse and order encodings do, the log encoding requires only $m = \lfloor \log_2 n \rfloor$ propositional variables. As a result, the number of SAT variables is exponentially reduced compared with the sparse or order encodings.

Let $l_b^i$ define a propositional variable and $l_b^i = 1$ if and only if bit $b$ of value $i$ (represented as a binary string) assigned to $V$ is 1 (see an example at Table 3.2 on page 40). The log encoding does not need the ALO and AMO clauses, but if the domain size of $n$ is not a power of two then we must exclude the non-domain values (at the top of each domain) by adding prohibited-value clauses. Frisch and Peugniez [FPDN05] introduced a variant of the log encoding, so-called binary encoding, which allows all combinations of values by adding extra combinations for extra values.

The main drawback of the log encoding is that it generates long clauses to encode constraints, especially for large domains (i.e., $n$ is large). Hence, SAT instances obtained by the log encoding may perform poorly compared to the sparse encoding and the order encoding. Theoretically, Walsh proved that the performance of unit
3.4. Compact Order Encoding

propagation, performed by DPLL SAT solvers, in the log encoding is less powerful than in the direct encoding (see Theorem 15 in [Wal00]). In practice, it has been observed that the log encoding usually has a poorer performance than the direct encoding, with the notable exception of the graph colouring problem [Vel07a] in which the author applied the static variable-ordering scheme. However, even in the graph colouring problem, the multi-valued direct encoding (i.e., the direct encoding without at-most-one clauses) usually runs faster than the log encoding [Gel08]. More often, the direct encoding outperforms the log encoding, for example, in the Hamiltonian path problem [Hoo99] or planning problems [EMW97]. Furthermore, Frisch and Peugniez pointed out [FPDN05] that the binary encoding, a variant of the log encoding, performs much worse than the direct encoding, and sometimes results in impractically large CNF formulas.

3.4 Compact Order Encoding

Tanjo et al. [TTB11] proposed the compact order encoding, aiming to take advantage of fast propagation of the order encoding and the compactness of the log encoding. Based on a numeral system of base $B \geq 2$, the compact order encoding encodes a variable $V$ with a domain \( \{1, \ldots, n\} \) by some digits that are then translated into SAT by the order encoding. The encoding can be described as follows: $V = \sum_{i=0}^{m-1} B^i v_i$, where $m = \lceil \log_B n \rceil^3$, $1 \leq x < n$, and $0 \leq v_i < B$ for all $v_i$, and each $v_i$ is encoded by the order encoding.

It is worth pointing out that the log and order encodings are two special cases of the compact order encoding with base $B = 2$ and $B \geq n$, respectively. As an example of encoding a finite domain, Table 3.2 on page 40 shows the Boolean variables used by SAT encodings.

The compact order encoding has been only evaluated on the open shop scheduling problem, in which it outperformed the log and order encodings [TTB11]. At the first glance, nevertheless, this encoding contains two main drawbacks. First, the compact order encoding loses the benefit of the order encoding in the representation of interval domains consequently, loosing the powerful propagation of bounds consistency. Second, like the log encoding, the compact order encoding generates long clauses compared with the sparse and order encodings (although shorter than the log encoding) to translate disequality constraints the form $X \pm c \neq Y$, where $X$ and $Y$ are integer variables, $c$ is a constant, into SAT. This drawback remains for CSP inequality constraints.\(^4\) Tanjo et al. [TTB11] pointed out that the compact order encoding performs poorly in general since it requires many inference steps to carry.

---

\(^3\) $\lceil m \rceil$ is the smallest integer not less than $m$.

\(^4\) $X \pm c \triangleright Y$, where $\triangleright \in \{\triangleright, \geq, <, \leq\}$. 
Table 3.2: An illustration of the log (L), order compact (O C), and order (O) encodings with a variable $V$ with domain $\{1, \ldots, 8\}$ (i.e. $n = 8$). In each assignment, the first line of the order compact encoding presents a value domain using a numeric system of base $B = 3$ (i.e $m = 2$), while the second line presents numeric system by the order encoding.

<table>
<thead>
<tr>
<th>$V$</th>
<th>$L$</th>
<th>$O$</th>
<th>$C$</th>
<th>$O$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$l_1^V$</td>
<td>$l_2^V$</td>
<td>$l_3^V$</td>
<td>$c_1^V$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

3.5 Hierarchical and Hybrid SAT Encodings

Velev first introduced the hierarchical and hybrid SAT encodings for CSP [Vel07a]. In these encodings, a domain is recursively subdivided into smaller subdomains until, at the lowest level, a domain value in each subdomain is selected. At each level of the hierarchy, one can choose a simple encoding (12 simple encodings presented in [Vel07a])$^5$ and the number of subdomains on the next levels. Each hierarchy of SAT encodings corresponds to a tree of ITE ("if-then-else") by selecting a value from a finite domain of CSP variables. An operator, named ITE($i$, $t$, $e$), checks the first Boolean parameter $i$; if $i=1$ then ITE selects the second argument $t$ and otherwise the third argument $e$. Due to the structure of the ITE tree, these SAT encodings require only conflict clauses and they do not require ALO and AMO clauses, as in the sparse or order encodings. By using these encodings, Velev could deal with the Hamiltonian cycle, graph colouring and formal verification of superscalar microprocessors problems [Vel07a, VG, VG09].

The hierarchical and hybrid SAT encodings can generate a large number of

---

$^5$ A simple encoding consists of the log encoding, the sparse and variants of sparse encodings.
translations (hundreds or thousands of different SAT encodings) of a CSP domain due to a variety of structures combining with 12 simple encodings (pointed out in [Vel07a]). Consequently, this method has to cope with too many different SAT encodings. For this reason, it is impractical to determine how these variants behave for a given problem. From a theoretical point of view, Velev did not show any consistency properties, an important feature for SAT encodings.

3.6 Representative Encodings

This section introduces two specific hierarchical hybrid encodings, the representative-sparse encoding and the representative-order encoding, for modeling a CSP domain to SAT. The representative encoding aims at taking advantage of the sparse and order encodings, but require a considerably smaller number of SAT variables. From a theoretical point of view, the new encodings that use only two levels in the hierarchy (all experiments obtained with three or more levels were clearly less efficient) can be parameterised by different sizes of the first level of the hierarchy. In general, these representative encodings are incomparable with respect to their flat counterparts (the sparse and order encodings which are special cases with one single partition).

From a practical point of view, the experimental results, presented in Section 5.1, in a set of representative benchmarks show that, regardless of the variability of run times in different SAT solvers, the representative encodings are quite competitive and usually outperform (sometimes very significantly) the sparse and order encodings. Therefore, these representative encodings are quite promising, highlighting their potential for handling hard and practical problems.

3.6.1 The Representative-Sparse Encoding

The representative-sparse encoding is a hierarchical hybrid encoding. To represent a CSP variable $V$ with domain $\{v_1, \ldots, v_n\}$, the representative-sparse encoding uses a set of Boolean representative variables $\{g_1, \ldots, g_m\}$ ($1 \leq m \leq n/2$) at level one to divide the domain into $m$ subdomains represented at level two with $r$ Boolean sparse variables $x_1, \ldots, x_r$ ($r = \lceil n/m \rceil$). The variables of both levels require ALO and AMO constraints. An assignment in this encoding is as follow:

$$V = v_i \iff \left\{ \begin{array}{ll} g_{\lfloor i/r \rfloor} & \land x_r & \text{if } i \mod r = 0; \\ g_{\lfloor i/r \rfloor + 1} & \land x_{i \mod r} & \text{otherwise}. \end{array} \right. \quad \text{(3.1)}$$

Formula 3.1 translates a finite CSP domain $\{v_1, \ldots, v_n\}$ of variable $V$ into SAT clauses by using Boolean representative variables $\{g_1, \ldots, g_m\}$, and a set of Boolean sparse variables, $\{x_1, \ldots, x_r\}$, where $r = \lceil n/m \rceil$.

Fig 3.2 shows how the Boolean representative variables, $g_i$ ($1 \leq i \leq m$), assign a value of $V$ to the subdomains. Note that the representative-sparse encoding can

\[\lfloor x \rfloor ] \] is the biggest (smallest) integer number not bigger (less) than $x$, and $\mod$ is the remainder operator.
Figure 3.2: An illustration of the representative-sparse encoding of the domain of a CSP variable using a group of representative variables at level one, \( \{g_1, g_2, g_3\} \), and a set of Boolean sparse variables at level two, \( \{x_1, x_2, x_3, x_4, x_5\} \), where exactly one variable is selected at each group.

represent a CSP domain of \( m \times r \) values. Therefore, when \( n < m \times r \) the prohibited values, which lie between \( \{n + 1, ..., m \times r\} \), must be excluded.

**Proposition 3.6.1** When indexing the domain values of the CSP variables into SAT variables, the representative-sparse encoding is sound and complete.

**Proof** The representative variables \( g_i, 1 \leq i \leq m \leq n/2 \), at level one partition the domain of variable \( V \) with \( n \) values into \( m \) following subdomains: \( \{v_1, \ldots, v_{[n/m]}\}, \ldots, \{v_{(m-1)[n/m]+1}, \ldots, v_n\} \). Moreover, the ALO and AMO clauses for both sets of Boolean variables \( \{g_1, \ldots, g_m\} \) and \( \{x_1, \ldots, x_r\} \) lead to the selection of exactly one partition and exactly one value from the represented partition for the value of \( V \).

Note that the sparse encoding requires \( n \) Boolean variables to encode a CSP variable \( V \) with domain \( \{v_1, \ldots, v_n\} \), whereas the representative-sparse encoding requires only \( m + \lfloor n/m \rfloor \) Boolean variables, \( 1 \leq m \leq n/2 \).

The sparse encoding is used by both the direct encoding [dK89] and the support encoding [Kas90]. Unit propagation on the direct encoding maintains a form of consistency called forward checking [Wal00], while unit propagation on the support encoding preserves arc-consistency [Gen02]. Propagation for representative-sparse encoding is not stronger or weaker with respect to the sparse encoding as stated in the following proposition.

**Proposition 3.6.2** Unit propagation applied to the representative-sparse encoding (when \( m \geq 2 \)) is not comparable to the sparse encoding.

**Proof** We prove for the case \( m = 3 \) (the others are similar). Let CSP variables \( V \) and \( W \) have domain \( \{1, \ldots, 15\} \), as in Fig 3.2, and be constrained by \( V \neq 3 \lor W \neq 5 \). In the representative-sparse encoding the constraint is represented by a clause \( \neg(g_1^V \land x_3^V) \lor \neg(g_1^W \land x_5^W) \), which is semantically equivalent to the clause \( \neg g_1^V \lor \neg x_3^V \lor \neg g_1^W \lor \neg x_5^W \) by applying de Morgan’s law. In the sparse encoding the
constraint is represented by a simpler clause $-d_3^V \lor -d_W^V$. For a subsequent assignment $W = 5$ (obtained during search or propagation), unit propagation (UP) in the *sparse encoding* results in the unit clause $-d_3^V$, that may be further propagated, whereas in the *representative-sparse encoding* it leads to the non-unit binary clause $-g_1^V \lor -x_3^V$. Hence, in this case, UP in the *sparse encoding* is stronger than in the *representative-sparse encoding*.

On the other hand, suppose variable $V$ with domain $\{2, 7, 12, 13\}$ be a constraint to be encoded into SAT. In the *representative-sparse encoding* the constraint is represented by a clause $(g_1^V \land x_2^V) \lor (g_2^V \land x_2^V) \lor (g_3^V \land x_2^V) \lor (g_4^V \land x_2^V)$, which is semantically equivalent to the clause $(x_2^V \land (g_1^V \lor g_2^V \lor g_3^V \lor g_4^V)) \lor (g_3^V \land x_2^V)$ by applying de Morgan’s law; consequently, the clause is simplified to a clause $x_2^V \lor (g_3^V \land x_2^V)$ due to $(g_1^V \lor g_2^V \lor g_3^V) = 1$, and finally we obtain the CNF clause $(x_2^V \lor g_3^V) \land (x_2^V \lor x_3^V)$ by applying de Morgan’s law. In the *sparse encoding* the constraint is represented by clause $d_1^V \lor d_2^V \lor d_3^V \lor d_4^V \lor d_5^V$. For a subsequent assignment $V < 11$ (i.e., $-g_3^V$), then UP in the *representative-sparse encoding* results in the unit clause $x_2^V$, whereas in the *sparse encoding* leads to $d_3^V \lor d_5^V$, that is not further propagated. Hence, in this case UP in the *representative-sparse encoding* is stronger than in the *sparse encoding*. □

Finally, it is worth noting that one of the key strengths of the *representative-sparse encoding* is the ability to represent interval variables significantly better than the *sparse encoding* (in terms of the length clauses) when the interval does not cross the partitions. For example, to represent $V \geq 11$, the *sparse encoding* requires the long clause $d_{11}^V \lor d_{12}^V \lor d_{13}^V \lor d_{14}^V \lor d_{15}^V$, whereas the *representative-sparse encoding* simply requires $g_3^V$ to be set to true. To represent $V \geq 12$, the *sparse encoding* requires the clause $d_{12}^V \lor d_{13}^V \lor d_{14}^V \lor d_{15}^V$, whereas the *representative-sparse encoding* requires the clause $g_3^V \lor -(g_3^V \land x_1^V)$, which is semantically equivalent to a clause $g_3^V \lor -x_2^V$ by applying de Morgan’s law.

There are several interesting remarks for *representative-sparse encoding*.

1. In general, one may tailor this encoding for specific problems. For example, when the elimination of all odd or even values from a domain are frequent, one could set $m = n/2$, with only two variables at the second level, one for the odd and the other for the even values.

2. The *sparse encoding* is a specialization of *representative-sparse encoding* by setting $m = 1$.

3. One can see the special case of *representative-sparse encoding* by setting $m = 2$, namely the *log-direct* encoding published in [NVB13].

### 3.6.2 The Representative-Order Encoding

The *representative-order encoding* uses Boolean *representative* variables $g_1, \ldots, g_m$ $(1 \leq m \leq n/2)$ at level one to divide the domain into $m$ partitions represented at level two with $r$ Boolean *order* variables $x_1, \ldots, x_r$ $(r = m + \lceil n/m \rceil - 1)$. The variables of level one require ALO and AMO constraints, while the variables of level
Figure 3.3: An illustration of the representative-order encoding of the domain of a CSP variable using a group of representative variables at level one, \( \{g_1, g_2, g_3\} \), where exactly one variable is selected, and a set of Boolean order variables at level two, \( \{x_1, x_2, x_3, x_4\} \), where these variables are set in the constraint of the order encoding.

two satisfies the domain constraint in the order encoding. An assignment in this encoding is as follows:

\[
V = v_i \iff \begin{cases} 
    g_{i/(r+1)} & \land x_r & \text{if } i \mod (r - 1) = 0; \\
    g_{i/(r+1)} + 1 & \land x_1 & \text{if } i \mod (r + 1) = 1; \\
    g_{i/(r+1)} + 1 & \land x_i \mod (r + 1) & \text{otherwise.}
\end{cases}
\]

(3.2)

Formula 3.2 translates a finite CSP domain of variable \( V = \{v_1, \ldots, v_n\} \), into SAT clauses by using Boolean representative variables, \( \{g_1, \ldots, g_m\} \), and a set of Boolean order variables, \( \{x_1, \ldots, x_r\} \), where \( r = \lfloor n/m \rfloor - 1 \).

Note that the order encoding requires \( n - 1 \) Boolean variables to encode a CSP variable \( V \) with domain \( \{v_1, \ldots, v_n\} \), whereas the representative-order encoding requires only \( m + \lfloor n/m \rfloor - 1 \) Boolean variables, \( 1 \leq m \leq n/2 \). Fig 3.3 shows how the representative variables, \( g_i, 1 \leq i \leq m \), assign a value \( V \) to the subdomains. As with the representative-sparse encoding when \( n < m/r \) the prohibited values must be excluded. This assignment is correct as stated in the following proposition.

**Proposition 3.6.3** When indexing the domain values of the CSP variables into SAT variables, the representative-order encoding is sound and complete.

**Proof** The representative variables at level one, \( g_i, 1 \leq i \leq m \leq n/2 \), divide the domain of variable \( V \) into \( m \) subdomains \( \{v_1, \ldots, v_{n/m}\}, \ldots, \{v_{(m-1)(n/m)+1}, \ldots, v_n\} \).

Given the at-least-one and at-most-one clauses for the representative Boolean variables \( \{g_1, \ldots, g_m\} \) exactly one partition can be selected. Moreover, the constraints on the Boolean variables \( \{x_1, \ldots, x_r\} \) imposed by the order encoding, lead to the selection for \( V \) of exactly one value in the selected partition. \( \Box \)

The comparison of the strength of unit propagation with the order and representative-order encoding is not straightforward. In the case of interval domains the order encoding is usually simpler, as it sets one single positive and one
single negative literal for the interval limits, whereas the \textit{representative-order} encoding requires the setting of the representative literals as well. In the above example of CSP variables $V$ having domain \{1, ..., 15\}, an interval $V \in \{3..10\}$ is imposed by setting $o_3^V \land \neg o_{10}^V$ in the order encoding whereas it requires setting $(g_3^V \land x_2^V) \lor g_2^V$ in the representative-order encoding.

On the other hand, for non-convex domains, the \textit{representative-order} encoding may be more compact. For example, suppose a variable $V$ with domain \{[2..4]; [7..9]; [12..14]\} be a constraint to be encoded into SAT. In the \textit{representative-order} encoding the constraint is represented as $(g_1^V \land x_1^V) \lor (g_2^V \land \neg x_1^V) \lor (g_3^V \land x_1^V) \lor (g_4^V \land \neg x_1^V) \lor (g_3^V \land x_1^V) \lor (g_4^V \land \neg x_1^V)$, which is semantically equivalent to a clause $(g_1^V \lor g_2^V \lor g_3^V)(x_1^V \lor \neg x_1^V)$, and finally to $x_1^V \land \neg x_1^V$ due to $(g_1^V \lor g_2^V \lor g_3^V) = 1$. Meanwhile, the order encoding requires the setting of $(o_3^V \land \neg o_{10}^V) \lor (o_6^V \land \neg o_{10}^V) \lor (o_{10}^V \land \neg o_{14}^V)$, which results in eight 3-ary clauses in CNF. Admittedly, this is an extreme example. Nevertheless, if the order encoding is usually more compact, many cases exist where the \textit{representative-order} is superior, especially with non-convex domains.

It is worth mentioning that the order encoding is a specialization of the \textit{representative-order} encoding by setting $m = 1$. Furthermore, one can see the special case of \textit{representative-order} encoding by setting $m = 2$, namely the log-order encoding published in [NVB13].

### 3.7 Discussion

This section presents an overview of different encodings and some discussion. We will provide an empirical study of SAT encodings in Chapter 5.

#### 3.7.1 Overview

To illustrate for SAT encodings of a finite domain, Table 3.3 on page 46 shows an example for different encodings. The log, sparse, order, and representative-sparse encoding are abbreviated as \textit{log}, \textit{sparse}, \textit{order}, and \textit{rep.-sparse}, respectively. \textit{Domain constraints} at the second column means the domain constraint for Boolean variables in each corresponding encoding.

Table 3.4 on page 47 indicates the comparison among different SAT encodings. Column \#Vars indicates the number of required variables in the corresponding encoding. Column \textit{Length} shows the length of clauses, whereas \textit{Requirement} means the domain constraint for the variables in the corresponding encoding. The log, sparse, order, compact order, representative-sparse encoding and representative-order encoding are abbreviated as \textit{log}, \textit{sparse}, \textit{order}, \textit{com.-order}, \textit{rep.-sparse}, and \textit{rep.-order}, respectively. As can be seen, the representative encodings seem to be a good tradeoff between the number of variables required and the length of generated clauses.
Table 3.3: The different encodings for translating CSP to SAT, illustrated on constraints: \( X = c \) and \( X \neq c \), where \( X \) and \( Y \) have a domain of eight values \( \{1, 2, \ldots, 8\} \).

<table>
<thead>
<tr>
<th>Encodings</th>
<th>Domain constraints</th>
<th>( X = i )</th>
<th>( X \neq i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>log</td>
<td>( X = 1 ) (-l_1^X \land \neg l_2^X \land \neg l_3^X) ( X \neq 1 ) ( l_2^X \lor l_3^X \lor l_3^X)</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td></td>
<td>( X = 2 ) (-l_1^X \land \neg l_2^X \land l_3^X) ( X \neq 2 ) ( l_1^X \lor l_2^X \lor \neg l_3^X)</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td></td>
<td>( X = 8 ) [\ldots]</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td>sparse</td>
<td>( d_1^X \lor \ldots \lor d_8^X ) ( X = 1 ) ( d_1^X ) ( X \neq 1 ) (-d_1^X)</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td></td>
<td>( \bigwedge_{1 \leq i \neq j \leq 8} (\neg d_i^X \lor \neg d_j^X) ) ( X = 2 ) ( d_2^X ) ( X \neq 2 ) (-d_2^X)</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td></td>
<td>[\ldots]</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td>order</td>
<td>( \bigwedge_{i=1}^6 (\neg o_i^X \lor \neg o_{i+1}^X) ) ( X = 1 ) (-o_1^X) ( X \neq 1 ) ( o_1^X)</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td></td>
<td>( X = 2 ) ( o_2^X \land \neg o_3^X) ( X \neq 2 ) (-o_1^X \lor o_2^X)</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td></td>
<td>( X = 3 ) ( o_2^X \land \neg o_3^X) ( X \neq 3 ) (-o_2^X \lor o_3^X)</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td></td>
<td>( X = 4 ) ( o_3^X \land \neg o_5^X) ( X \neq 4 ) (-o_3^X \lor o_5^X)</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td></td>
<td>( X = 5 ) ( o_4^X \land \neg o_5^X) ( X \neq 5 ) (-o_3^X \lor o_5^X)</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td></td>
<td>( X = 6 ) ( o_5^X \land \neg o_6^X) ( X \neq 6 ) (-o_3^X \lor o_6^X)</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td></td>
<td>( X = 7 ) ( o_6^X \land \neg o_7^X) ( X \neq 7 ) (-o_3^X \lor o_7^X)</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td></td>
<td>( X = 8 ) ( o_7^X) ( X \neq 8 ) (-o_7^X)</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td>rep.-sparse (m=2)</td>
<td>( \bigwedge_{1 \leq i \neq j \leq 4} (\neg x_i^X \lor \neg x_j^X) ) ( X = 1 ) ( g_1^X \land x_1^X) ( X \neq 1 ) (-g_1^X \lor \neg x_1^X)</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td></td>
<td>( X = 3 ) ( g_1^X \land x_3^X) ( X \neq 3 ) (-g_1^X \lor \neg x_3^X)</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td></td>
<td>( X = 4 ) ( g_2^X \land x_1^X) ( X \neq 4 ) (-g_2^X \lor \neg x_1^X)</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td></td>
<td>( X = 5 ) ( g_3^X \land x_3^X) ( X \neq 5 ) (-g_3^X \lor \neg x_3^X)</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td></td>
<td>( X = 6 ) ( g_2^X \land x_3^X) ( X \neq 6 ) (-g_2^X \lor \neg x_3^X)</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td></td>
<td>( X = 7 ) ( g_1^X \land x_4^X) ( X \neq 7 ) (-g_1^X \lor \neg x_4^X)</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
<tr>
<td></td>
<td>( X = 8 ) ( g_2^X \land x_4^X) ( X \neq 8 ) (-g_2^X \lor \neg x_4^X)</td>
<td>[\ldots]</td>
<td>[\ldots]</td>
</tr>
</tbody>
</table>

3.7.2 Are ALO and AMO Clauses Necessary?

To guarantee that each CSP variable is assigned exactly one value, a SAT instance must include the ALO and AMO clauses. However, in some cases SAT encodings do not require either ALO clauses or AMO clauses or both.

Selman et al. [SLM92] first introduced a SAT encoding, a variant of the direct encoding (see Section 4.1.1) that omits AMO clauses. There is no longer the 1-to-1 correspondence between SAT and CSP solutions, in which each CSP variable might be assigned more than one value simultaneously and a solution to the original CSP can be decoded from a SAT solution by extracting any one of the assigned values for each CSP variable.

Prestwich [Pre03a] conjectured and experimentally showed that omitting AMO clauses (on the graph colouring problem) may increase the solution density of the
Table 3.4: An illustration of SAT encodings of a variable \( V \) with domain \( \{1, 2, \ldots, n\} \). The second last column shows the length of SAT clauses when encoding the disequality constraint \( V \neq c \), while the last shows the requirement.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>#Vars</th>
<th>Note</th>
<th>Length</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>sparse</td>
<td>( n )</td>
<td>unit</td>
<td>AMO</td>
<td></td>
</tr>
<tr>
<td>order</td>
<td>( n - 1 )</td>
<td>binary</td>
<td>domain constraint</td>
<td></td>
</tr>
<tr>
<td>log</td>
<td>( m = \lfloor \log_2 n \rfloor )</td>
<td>( m )-ary</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>com.-order</td>
<td>( \lfloor \log_2 n \rfloor (B - 1) )</td>
<td>( B \geq 2 )</td>
<td>( (2B - 1) )-ary</td>
<td>domain constraint</td>
</tr>
<tr>
<td>rep.-sparse</td>
<td>( m + \lfloor n/m \rfloor )</td>
<td>1 ( \leq m \leq n/2 )</td>
<td>binary</td>
<td>AMO</td>
</tr>
<tr>
<td>rep.-order</td>
<td>( m + \lfloor n/m \rfloor - 1 )</td>
<td>1 ( \leq m \leq n/2 )</td>
<td>3-ary</td>
<td>domain constraint</td>
</tr>
</tbody>
</table>

search space, and as a result one might solve the problem more quickly.

A naturally interesting problem leads to the question: when can ALO and AMO clauses be omitted? Fortunately, Frisch et al. [FPDN05] identified the condition for omitting these clauses from SAT instances, obtained by encoding CSPs.

The following definition applies to both Boolean and non-Boolean formulas. However, we will only consider Boolean formulas.

**Definition 3.7.1** (Positive and Negative Formulas - Definition 1 in [FPDN05]) A formula occurs positively with itself. If \( \alpha \) occurs positively (negatively) within \( \gamma \), then \( \alpha \) occurs positively (negatively) within \( \gamma \land \beta, \beta \land \gamma, \gamma \lor \beta, \beta \lor \gamma, \beta \rightarrow \gamma, \gamma \leftrightarrow \beta \) and \( \beta \leftrightarrow \gamma \). If \( \alpha \) occurs positively (negatively) within \( \gamma \), then \( \alpha \) occurs negatively (positively) within \( \neg \gamma, \gamma \rightarrow \beta, \gamma \leftrightarrow \beta \) and \( \beta \leftrightarrow \gamma \). A formula is said to be negative (positive) with respect to an atom if that atom does not occur positively (negatively) in the formula. A formula is said to be positive (negative) if it is positive (negative) with respect to all atoms.

The following theorem specifies the sufficient conditions for a formula which can omit ALO clauses or AMO clauses. Note that the unary/unary transform used in [FPDN05] is the sparse encoding.

**Theorem 3.7.2** (Satisfiability without ALO or AMO - Theorem 1 in [FPDN05]) Let \( K \) be an arbitrary Boolean formula, \( L \) be a conjunction of ALO formulas, and \( M \) be a conjunction of AMO formulas. Let AMO(\( V \)) and ALO(\( V \)) be AMO and ALO formulas, respectively, for \( d_1^V, \ldots, d_n^V \). (1) If \( K \) is negative with respect to each of \( d_1^V, \ldots, d_n^V \) and \( K \land L \land M \) is satisfiable, then so is \( K \land L \land M \land \text{AMO}(\mathcal{V}) \). (2) If \( K \) is positive with respect to each of \( d_1^V, \ldots, d_n^V \) and \( K \land L \land M \) is satisfiable, then so is \( K \land L \land M \land \text{ALO}(\mathcal{V}) \).

**Example 3.7.3** Consider the graph colouring as an example by using two adjacent vertices, \( V \) and \( W \), with the same domain (of colours) \( \{1, \ldots, n\} \). One must guarantee that no two adjacent vertices share the same color. The constraint generates the following clauses:

\[-(d_i^V \land d_j^W) \equiv -(d_i^V \land -d_j^W), 1 \leq i \neq j \leq n.\]
According to Theorem 3.7.2, the set of above clauses is negative with respect to each of variable. Hence, AMO clauses are not necessary.

The main results of this chapter are also published in [NVB13, BHN14b].
Chapter 4

SAT Encodings of Finite CSP Constraints

Chapter 3 introduced SAT encodings of finite CSP domains. This chapter shows how to encode finite CSP constraints into SAT. In CSP, constraints can be classified into three types: the *extensional*, *intensional*, and *global* constraints. In fact, global constraints are a special case of intensional constraints, due to their important role, which have been intensively studied in CSPs.

This chapter is organized as follows. Section 4.1 presents two SAT encodings of *extensional* and *intensional* constraints.

Section 4.2 shows SAT encodings of the *at-most-one* (AMO) constraint. The AMO constraint is to account for one value for a CP variable when modeling. As we showed in Chapter 3, the sparse encoding is one of the most widely used SAT encoding of CSPs. To use this encoding, one has to cope with the AMO constraint. As a result, encoding the AMO constraint into SAT is essential. Furthermore, global constraints provide an excellent example to reveal the difference between SAT and CP in terms of modeling. In CP, hundreds of global constraints are studied (see [vHK06] and [glo]), whereas in SAT, only few global constraints are investigated. One of the most important these is the *alldifferent* constraint, which occurs in a huge number of applications. In SAT, the traditional way to deal with *alldifferent* is to use SAT encoding of the AMO constraint.

This thesis proposes a new encoding, addresses the similarity among several encodings, and conducts an empirical study of various SAT encodings of the AMO constraint. Moreover, to distinguish the term used by a SAT encoding of CSP domains, the term AMO SAT-encoding is used to refer to a SAT encoding of the AMO constraint.

Section 4.3 provides useful new insights regarding the choice of suitable SAT encodings by taking into account several features of CSP constraints.
4.1 Encoding Extensional and Intensional Constraints

An extensional constraint is specified as an explicit list (table) of allowed assignments or disallowed assignments to the variables it constrains. In other words, the extensional constraint is static and not procedural. In contrast, an intensional constraint expresses the relationships that must hold among the assignments to the variables it constrains in an abstract way. Hence, the representation of an intensional constraints is procedural.

In CP, the advantage of intensional constraints is that they are more practical than extensional constraints since they require a smaller formula; whereas in SAT, intensional constraints do not preserve such advantage. For SAT encoding, the following two cases are considered:

- if a CSP constraint is expressed as an extensional constraint, then one can directly generate SAT clauses (see Sections 4.1.1 and 4.1.2), or

- if a CSP constraint is expressed as an intensional constraint, then one must first translate it into an extensional constraint before generating SAT clauses. In particular, the intensional constraint is grounded by generating all the possible input values and checked if it: 1) satisfies the constraint, then one obtains an allowed assignment, or 2) does not satisfy the constraint, then one obtains disallowed assignments.

The above methodology is shown in Figure 4.1 as step 1 results in extensionally allowed/disallowed assignments. From now on, this thesis only considers how to encode/translate these assignments into SAT clauses (in CNF). Step 2 will be addressed in Sections 4.1.1 and 4.1.2.

It is worth pointing out that the step of enumerating all possible assignments leads to \( d^2 \) clauses in binary CSP constraints, where \( d \) is the size of the domain of variables. In general, for a constraint having \( n \) variables, the number of clauses may be up to \( d^n \), an exponential number. Fortunately, SAT solving can often deal with such large formulas thanks to modern SAT solvers.

The relationship between local consistency in CSP and unit propagation is important for SAT encodings. In CSP, some local consistency techniques (e.g., arc consistency and forward checking, see Section 2.1.2) are quite effective for their tradeoff between the cost of the constraint propagation performed at each node in the search tree and their pruning efficiency. SAT solvers [SS96, MMZ+01, ES05, SLM09, Man14] use unit propagation, also called Boolean constraint propagation, as a constraint propagation mechanism. For that reason, when translating a CSP to a SAT instance one should be concerned on whether unit propagation on the resulting SAT instance enforces local consistency on the original CSP. For example, if unit propagation, which is working on a SAT instance obtained by the sparse encoding, infers that a Boolean variable \( x_i^V \) is 0, then the value \( i \) of the corresponding CSP variable \( V \) has been eliminated.
Figure 4.1: A flowchart representing a process of translating CSP constraints into SAT clauses.
There are two types of clauses in CNF to represent the resulting formula by translating a CSP constraint into SAT. The first uses conflict clauses to specify the disallowed variable assignments in the original constraint, whereas the second uses support clauses to specify the allowed variable assignments.

### 4.1.1 Conflict Clauses

In CP, there are tuples or assignments that do not satisfy a constraint. These assignments are called disallowed variable assignments. To represent such assignments in SAT, one can use conflict clauses.

**Definition 4.1.1 (Conflict Clauses)** A conflict clause is a SAT clause that expresses an assignment that is inconsistent with a constraint in the corresponding CSP.

The purpose of a conflict clause is to eliminate any conflict among variables assigned by their values in the corresponding CSP.

To understand the definition more easily, let us consider a binary CSP which consists of a set of variables (e.g., $W$ and $V$), a set of values (e.g., $i$ and $j$), and a set of constraints. Let $K_{Vi}$ be a set of pairs $(W, j)$ such that there exists a disallowed assignment $(V = i, W = j)$ due to a constraint. The conflict clauses are then expressed by:

$$
\bigwedge_{(W, j) \in K_{Vi}} \neg(d_i^V \land d_j^W) \equiv \bigwedge_{(W, j) \in K_{Vi}} (-d_i^V \lor -d_j^W).
$$

Let us show how to translate binary CSP constraints into conflict clauses. For extensional constraints, two cases are considered:

1. When an extensional constraint is specified by an explicit list of disallowed assignments, one can directly translate these assignments into conflict clauses (see Table 4.1).

2. When an extensional constraint is specified by an explicit list of allowed assignments, one first enumerates all possible assignments and then translates the ones which are not in the set of allowed assignments into conflict clauses (see Table 4.2).

The set of disallowed assignments extracted from the list of allowed assignments is exactly the complement of the set of allowed assignments. The extraction possibly generates a large number of assignments. To avoid this drawback, therefore, one can use the support clause (see Section 4.1.2).

Throughout the examples shown in the following tables, one can see that it is easier to translate disallowed assignments to conflict clauses than allowed assignments. To illustrate this, suppose that two CSP variables, $X$ and $Y$ share the same domain \{1, 2, 3\}, and let $(x, y)$ denote an ordered pair of values of $X$ and $Y$.

Table 4.3 presents an example of how to translate an extensional constraint $X < Y$ into conflict clauses in SAT. Although Table 4.2 and Table 4.3 express the
4.1. Encoding Extensional and Intensional Constraints

Table 4.1: An example of a translation of an extensional constraint into conflict clauses. The constraint, represented by $X \geq Y$ in intensional, is defined over two variables $X$ and $Y$ with the same domain $\{1, 2, 3\}$, and expressed by disallowed assignments.

<table>
<thead>
<tr>
<th>extensional constraints</th>
<th>conflict clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>(disallowed assignment)</td>
<td></td>
</tr>
<tr>
<td>(1, 2)</td>
<td>$-d_1^X \lor -d_2^X$</td>
</tr>
<tr>
<td>(1, 3)</td>
<td>$-d_1^X \lor -d_3^X$</td>
</tr>
<tr>
<td>(2, 3)</td>
<td>$-d_2^X \lor -d_3^X$</td>
</tr>
</tbody>
</table>

Table 4.2: An example of a translation of an extensional constraint into conflict clauses. The constraint, represented by $X < Y$ in intensional, is defined over two variables $X$ and $Y$ with the same domain $\{1, 2, 3\}$, and expressed by allowed assignments.

<table>
<thead>
<tr>
<th>extensional constraints</th>
<th>disallowed assignments</th>
<th>conflict clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>(allowed assignment)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1, 2)</td>
<td>${(1, 1), (2, 1), }$</td>
<td>$-d_1^X \lor -d_2^X$</td>
</tr>
<tr>
<td>(1, 3)</td>
<td>$(2, 2), (3, 1), }$</td>
<td>$-d_1^X \lor -d_3^X$</td>
</tr>
<tr>
<td>(2, 3)</td>
<td>$(3, 2), (3, 3)}$</td>
<td>$-d_2^X \lor -d_3^X$</td>
</tr>
</tbody>
</table>

same constraint $X < Y$ over two variables $X$ and $Y$ with the same domain $\{1, 2, 3\}$, they are different with respect to the way of representing the constraint. Table 4.2 shows an extensional constraint with an explicit list of allowed assignments, whereas Table 4.3 shows an intensional constraint, which represents the relationship between $X$ and $Y$.

Table 4.3: An example of a translation of an intensional constraint into conflict clauses. The constraint $X < Y$ is defined over two variables $X$ and $Y$ with the same domain $\{1, 2, 3\}$.

<table>
<thead>
<tr>
<th>intensional constraint</th>
<th>disallowed assignments</th>
<th>conflict clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$(1, 1), (2, 1), }$</td>
<td>$-d_1^X \lor -d_2^X$</td>
</tr>
<tr>
<td>$X &lt; Y$</td>
<td>$(2, 2), (3, 1), }$</td>
<td>$-d_1^Y \lor -d_3^X$</td>
</tr>
<tr>
<td></td>
<td>$(3, 2), (3, 3)}$</td>
<td>$-d_2^Y \lor -d_3^X$</td>
</tr>
</tbody>
</table>

The SAT encoding using the sparse encoding for encoding a finite CSP domain and conflict clauses for encoding a finite CSP constraint is called the direct encoding [dIK89, Wa00]. It is formally defined as follows.

**Definition 4.1.2 (Direct Encoding)** The direct encoding of a binary CSP consists of the appropriate at-least-one, at-most-one, and conflict clauses.
The at-least-one and at-most-one clauses are generated for each CSP variables, whereas the conflict clauses are generated from each CSP constraint.

**Example 4.1.3** We illustrate the direct encoding with a CSP: the graph colouring problem (see Section 2.1.3). The problem consists of two vertices, V and W, sharing the same domain of three colours \{1, 2, 3\}. We use the notation for a Boolean variable as in Section 3.1.

\[
\text{AMO clauses} \quad d_1^V \lor d_2^V \lor d_3^V \\
\lor d_1^W \lor d_2^W \lor d_3^W
\]

\[
\text{ALO clauses} \quad -d_1^V \lor -d_2^V \quad -d_3^Y \lor -d_3^Y \\
\lor -d_1^W \lor -d_2^W \lor -d_3^W
\]

\[
\text{conflict clauses} \quad -d_1^V \lor -d_1^W \quad -d_2^V \lor -d_2^W \quad -d_3^V \lor -d_3^W
\]

The following proposition indicates that there is a 1-to-1 correspondence between the SAT solution obtained by the direct encoding and the CSP solution. In other words, any SAT solution can be decoded to get a corresponding CSP solution and vice versa.

**Proposition 4.1.4** (Soundness and Completeness) The direct encoding is sound and complete.

**Proof** Firstly, ALO and AMO clauses in a SAT instance ensure that every CSP variable has exactly one value from its domain. Furthermore, conflict clauses guarantee that no combination of variable assignments in CSP violates any constraint. Therefore, every complete interpretation of the SAT instance corresponds to a solution to the original CSP. Secondly, every solution to the CSP, which satisfies every constrain, corresponds to a complete interpretation of the SAT instance, which also satisfies all the clauses (i.e., ALO, AMO, and conflicts clauses).

Theoretically, there are several relationships between unit propagation working in DPLL SAT solvers and constraint propagation (e.g., forward checking and arc consistency) in CSP solving. However, one should be aware that the performance is different between theory and practice. For example, it has been observed that one SAT encoding is more efficient in theory with respect to the performance of UP, but it performs worse than another SAT encoding in practice with respect to running time. Some important theorems demonstrating this are represented here, and the proofs can be found in [Wal00]. Note that the DP procedure [DP60, DLL62] here only consists of three rules: (1) the backtracking rule, applied when an empty clause is generated; (2) the unit propagation rule, applied at each node in the search tree; and (3) the branching rule, applied for selecting and assigning unassigned variables.
4.1. Encoding Extensional and Intensional Constraints

**Theorem 4.1.5** (Theorem 12 in [Wal00])

- If unit propagation commits to particular truth assignments on the direct encoding, then enforcing arc-consistency on the original problem eliminates all contradictory values.

- If unit propagation generates the empty clause in the direct encoding then enforcing arc-consistency on the original problem causes a domain wipeout (but the reverse does not necessarily hold).

Theorem 4.1.5 indicates that enforcing arc-consistency on a CSP is stronger than unit propagation on the SAT-encoded CSP by the direct encoding.

**Theorem 4.1.6** (Theorem 13 in [Wal00]) Given equivalent branching heuristics, DP applied to the direct encoding explores the same number of branches as forward checking (FC) applied to the original problem.

Theorem 4.1.6 reveals that FC applied to a CSP explores the same size search tree as the DP applied to on the SAT-encoded CSP by the direct encoding.

**Theorem 4.1.7** (Theorem 14 in [Wal00]) Given equivalent branching heuristics, maintain arc consistency (MAC) applied to the original problem strictly dominates DP applied to the direct encoding.

Theorem 4.1.7 shows that MAC is stronger than DP when applied on the SAT-encoded CSP by the direct encoding.

Like the direct encoding, the log encoding, described in Section 3.3, uses conflict clauses, not support clauses. Walsh [Wal00] proved that unit propagation on the direct encoding is more effective than unit propagation on the log encoding.

**Theorem 4.1.8** (Theorem 15 in [Wal00])

- If unit propagation commits to particular truth assignments on the log encoding, then unit propagation commits to the same truth assignments on the direct encoding.

- If unit propagation generates the empty clause in the log encoding then unit propagation generates the empty clause in the direct encoding then (but the reverse does not necessarily hold).

**Theorem 4.1.9** (Theorem 16 in [Wal00]) Given equivalent branching heuristics, forward checking (FC) applied to the original problem strictly dominates DPLL procedure applied to the log encoding.

The above results considered binary CSPs. However, we also use the direct encoding for non-binary CSPs. Any non-binary CSP constraint can be converted into an equivalent binary constraint [BvB98]. For the constraints of arity $n$, one has two choices: 1) convert these constraints into binary constraints, and then translate
these binary constraints into SAT clauses; 2) translate directly disallowed assignments to corresponding clauses. Obviously, the arity of the constraint determines the length of the corresponding clauses.

Interestingly, Bessiére et al. [BHW04] generalized the direct encoding to non-binary CSPs and other forms of consistency.

4.1.2 Support Clauses

In CP, there are tuples or assignments that satisfy any constraint. These assignments are called allowed variable assignments. To represent such assignments in SAT, one can use support clauses.

**Definition 4.1.10 (Conflict Clauses)** A support clause is a SAT clause that expresses an assignment that is consistent with a constraint in the corresponding CSP.

The purpose of a support clause is to represent a support of a variable with its values for another variable in the corresponding CSP.

To understand the definition, let us consider a binary CSP, which consists of a set of variables (e.g., $W$ and $V$), a set of values (e.g., $i$ and $j$), and a set of constraints. The assignment $V = i$ supports the assignment $W = j$ if they do not violate any constraints. Let $S_{V,W,j}$ be the supporting values in domain of variable $V$ for value $W = j$. The support clauses are expressed by the following formula:

$$d_j^W \rightarrow (\bigvee_{i \in S_{V,W,j}} d_i^V) \equiv \neg d_j^W \lor (\bigvee_{i \in S_{V,W,j}} d_i^V).$$

The support clauses imply that whenever $d_j^W$ holds then at least one of its supports must hold. Moreover, if all the supports of $W = j$ are falsified (i.e., $d_i^V = 0$ for all $i$), so is $d_j^W$.

As an alternative to the direct encoding, the support encoding was introduced by Kasif [Kas90] and studied further by Gent [Gen02]. The support encoding uses the sparse encoding for encoding a finite CSP domain and support clauses for encoding a finite CSP constraint. Let us first consider how to translate a binary constraint into support clauses. For extensional constraints, two cases are considered:

1. When an extensional constraints is specified by an explicit list of allowed assignments, one can directly translate these assignments into support clauses (see Table 4.4).

2. When an extensional constraints is specified by an explicit list of disallowed assignments, one first enumerates all possible assignments, and then translates the ones which are not in the set of disallowed assignments into support clauses (see Table 4.5).

The set of allowed assignments extracted from the list of disallowed assignments is exactly the complement of the set of disallowed assignments. The extraction possibly generates a large number of assignments. To avoid this drawback, therefore, one can use conflict clauses (see Section 4.1.1).
4.1. Encoding Extensional and Intensional Constraints

**Definition 4.1.11 (Support Encoding)** (Definition 1 in [Gen02]) The support encoding of a binary CSP consists of the appropriate at-least-one, at-most-one, and support clauses.

**Example 4.1.12** We illustrate the support encoding with a CSP: the graph colouring problem (see Section 2.1.3). The problem consists of two vertices, \( V \) and \( W \), sharing the same domain of three colours \( \{1, 2, 3\} \). We use the notation for a Boolean variable as in Section 3.1.

\[
\begin{align*}
\text{AMO clauses} & \quad d_1^V \lor d_2^V \lor d_3^V \\
& \quad d_1^W \lor d_2^W \lor d_3^W \\
\text{ALO clauses} & \quad -d_1^V \lor -d_3^V \\
& \quad -d_2^V \lor -d_3^V \\
\text{support clauses} & \quad -d_1^V \lor d_2^V \lor d_3^V \\
& \quad -d_1^W \lor d_2^W \lor d_3^W \\
& \quad -d_1^W \lor d_2^W \lor d_3^W
\end{align*}
\]

**Proposition 4.1.13** (Corollary 5 and Theorem 6 in [Gen02]) The support encoding is sound and complete.

Proposition 4.1.13 means that there is a 1-to-1 correspondence between SAT and CSP solutions. Furthermore, one can decode any SAT solution to get a corresponding CSP solution and vice versa.

**Table 4.4:** An example of a translation of an extensional constraint into support clauses. The constraint, represented by \( X < Y \) in an intensional, is defined over two variables \( X \) and \( Y \) with the same domain \( \{1, 2, 3\} \), and expressed by allowed assignments.

<table>
<thead>
<tr>
<th>extensional constraints (allowed assignment)</th>
<th>support clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 2),</td>
<td>(-d_1^X \lor d_2^Y \lor d_3^X )</td>
</tr>
<tr>
<td>(1, 3),</td>
<td>(-d_1^X \lor d_3^Y \lor d_2^X )</td>
</tr>
<tr>
<td>(2, 3),</td>
<td>(-d_2^X \lor d_3^Y \lor d_2^X )</td>
</tr>
</tbody>
</table>

Table 4.6 presents an example of how to translate an intensional constraint \( X < Y \) into support clauses in SAT. Although Table 4.5 and Table 4.6 express the same constraint \( X < Y \) over two variables \( X \) and \( Y \) with the same domain \( \{1, 2, 3\} \), they are different with respect to the way of representing the constraint. Table 4.5 shows an extensional constraint with an explicit list of allowed assignments, whereas Table 4.2 shows an intensional constraint, which represents the relationship between \( X \) and \( Y \).

Intuitively, when constraints are described as allowed assignments, one should use the support encoding. In contrast, when constraints are described as disallowed assignments, the direct encoding should be used instead.
Table 4.5: An example of a translation of an extensional constraint into support clauses. The constraint, represented by \( X \geq Y \) in intensional, is defined over two variables \( X \) and \( Y \) with the same domain \( \{1, 2, 3\} \), and expressed by disallowed assignments.

<table>
<thead>
<tr>
<th>extensional constraints</th>
<th>allowed assignments</th>
<th>support clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>((1, 2))</td>
<td>(1, 1)</td>
<td>(-d_1^X \vee d_1^Y)</td>
</tr>
<tr>
<td></td>
<td>(2, 1)</td>
<td>(-d_2^X \vee d_2^Y)</td>
</tr>
<tr>
<td>((1, 3))</td>
<td>(2, 2)</td>
<td>(-d_2^X \vee d_3^Y)</td>
</tr>
<tr>
<td></td>
<td>(3, 1)</td>
<td>(-d_3^X \vee d_2^Y)</td>
</tr>
<tr>
<td>((2, 3))</td>
<td>(3, 2)</td>
<td>(-d_3^X \vee d_3^Y)</td>
</tr>
<tr>
<td></td>
<td>(3, 3)</td>
<td>(-d_3^X \vee d_3^Y)</td>
</tr>
</tbody>
</table>

Table 4.6: An example of a translation of an intensional constraint into support clauses. The constraint \( X < Y \) is defined over two variables \( X \) and \( Y \) with the same domain \( \{1, 2, 3\} \).

<table>
<thead>
<tr>
<th>intensional constraints</th>
<th>allowed assignments</th>
<th>support clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X &lt; Y )</td>
<td>(1, 2)</td>
<td>(-d_1^X \vee d_2^Y)</td>
</tr>
<tr>
<td></td>
<td>(1, 3)</td>
<td>(-d_1^X \vee d_3^Y)</td>
</tr>
<tr>
<td></td>
<td>(2, 3)</td>
<td>(-d_2^X \vee d_3^Y)</td>
</tr>
</tbody>
</table>

Theoretically, in analogy to the direct encoding there are several relationships between unit propagation on the support encoding and constraint propagation. The following theorem was proven by Gent [Gen02].

**Theorem 4.1.14** (Corollary 7 in [Gen02]) DP (without pure literal deletion) on the support encoding of CSP’s and MAC on the original instance perform equivalent search, given equivalent branching decisions.

Compared with Theorem 4.1.5, one can claim that DP, applied to the support encoding, produces a smaller search space than when applied to the direct encoding since MAC requires a smaller search space than FC (see Section 2.1.2). Gent pointed out that the support clause can be derived from the conflict clause by binary resolution. In addition, Drake et al.[DFGW02] described a generalized binary resolution, namely HypBinRes (a hyper-binary resolution step is a reference that involves more than two binary clauses [Bac02]) in which one can use a single reasoning step for automatically inferring support clauses from the direct encoding. The authors showed that these inferred support clauses improve SAT solver performance.

The above results considered binary CSPs. However, we also use the support encoding for non-binary CSPs. For constraints of arity \( n \), one can convert any non-binary CSP constraint into an equivalent binary constraint [BvB98]. The following example is drawn from [Rou05]. Suppose that \( X \), \( Y \), and \( Z \)
4.1. Encoding Extensional and Intensional Constraints

59

have domains \(\{(x_1, x_2, x_3), (y_1, y_2, y_3), (z_1, z_2, z_3)\}\), respectively. Assume that three variables are defined by a constraint with a set of support assignments: \(\{(x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3)\}\). In order to convert these support assignments into binary constraints, we introduce a new variable \(V\) with domain \(\{1, 2, 3\}\), where 1, 2 and 3 represent the tuples \(\{(x_1, x_2, x_3), (y_1, y_2, y_3), (z_1, z_2, z_3)\}\), respectively. Then, we have the following binary constraints between \(V\) and \(X\), between \(V\) and \(Y\) and \(V\) and \(Z\) with support tuples \(\{(1, x_1), (2, y_1), (3, z_1)\}\), \(\{(1, x_2), (2, y_2), (3, z_2)\}\), and \(\{(1, x_3), (2, y_3), (3, z_3)\}\), respectively.

Interestingly enough, Bessière et al. [BHW04] generalized the support encoding to non-binary CSPs and other form of consistency.

From now on, to encode a CSP, we can integrate any encoding of finite domains presented in Chapter 3 (e.g., sparse, order, log, representative-sparse, and representative-order encodings) with one of two types of clause: conflict clauses (introduced in Section 4.1.1) or support clauses (introduced in Section 4.1.2).

Table 4.7 on page 59 demonstrates how a constraint \(X \neq Y\) is translated into SAT by using the sparse encoding and the order encoding with different clauses: conflict and support clauses. Due to space, we use a small domain with 4 values \(\{1, 2, 3, 4\}\) for \(X\) and \(Y\). Nevertheless, Table 4.7 reveals that to translate constraint \(X \neq Y\), conflict clauses always are binary clauses, whereas the larger the domain of CSP variables is, the longer support clauses are. Due to space we do not convert support clauses (currently in non-CNF) produced by the order encoding into CNF (on Table 4.7). However, it is worth noting that the order encoding potentially produces an exponential number of long clauses when using support clauses.

Table 4.7: The different encodings for translating CSP to SAT, illustrated on constraints \(X \neq Y\) using conflict clauses and support clauses, where \(X\) and \(Y\) have a domain of 4 values \(\{1, 2, 3, 4\}\).

<table>
<thead>
<tr>
<th>enc</th>
<th>conflict</th>
<th>CSP</th>
<th>support</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>sparse (\neg(X = 1 \land Y = 1))</td>
<td>(-d_1^x \lor -d_1^y)</td>
<td>(X = 1 \lor (Y = 2) \lor (Y = 3) \lor (Y = 4))</td>
<td>(-d_2^x \lor -d_2^y \lor -d_2^z \lor -d_2^z)</td>
<td>(\neg 0^x \lor \neg 0^y \lor \neg 0^z \lor \neg 0^z)</td>
</tr>
<tr>
<td>(\neg(X = 2 \land Y = 2))</td>
<td>(-d_2^x \lor -d_2^y)</td>
<td>(X = 2 \lor (Y = 1) \lor (Y = 3) \lor (Y = 4))</td>
<td>(-d_2^x \lor -d_2^y \lor -d_2^z \lor -d_2^z)</td>
<td>(\neg 0^x \lor \neg 0^y \lor \neg 0^z \lor \neg 0^z)</td>
</tr>
<tr>
<td>(\neg(X = 3 \land Y = 3))</td>
<td>(-d_3^x \lor -d_3^y)</td>
<td>(X = 3 \lor (Y = 1) \lor (Y = 2) \lor (Y = 4))</td>
<td>(-d_3^x \lor -d_3^y \lor -d_3^z \lor -d_3^z)</td>
<td>(\neg 0^x \lor \neg 0^y \lor \neg 0^z \lor \neg 0^z)</td>
</tr>
<tr>
<td>(\neg(X = 4 \land Y = 4))</td>
<td>(-d_4^x \lor -d_4^y)</td>
<td>(X = 4 \lor (Y = 1) \lor (Y = 2) \lor (Y = 3))</td>
<td>(-d_4^x \lor -d_4^y \lor -d_4^z \lor -d_4^z)</td>
<td>(\neg 0^x \lor \neg 0^y \lor \neg 0^z \lor \neg 0^z)</td>
</tr>
<tr>
<td>order (\neg(X = 1 \land Y = 1))</td>
<td>(a_1^x \lor a_1^y)</td>
<td>(X = 1 \lor (Y = 2) \lor (Y = 3) \lor (Y = 4))</td>
<td>(a_2^x \lor (a_1^x \land a_2^y) \lor a_2^y)</td>
<td>(a_2^x \lor (a_1^x \land a_2^y) \lor a_2^y)</td>
</tr>
<tr>
<td>(\neg(X = 2 \land Y = 2))</td>
<td>(-a_2^x \lor a_2^x \lor -a_2^y \lor a_2^y)</td>
<td>(X = 2 \lor (Y = 1) \lor (Y = 3) \lor (Y = 4))</td>
<td>((-a_2^x \lor a_2^x) \lor (a_2^x \lor -a_2^y) \lor a_2^y)</td>
<td>((-a_2^x \lor a_2^x) \lor (a_2^x \lor -a_2^y) \lor a_2^y)</td>
</tr>
<tr>
<td>(\neg(X = 3 \land Y = 3))</td>
<td>(-a_3^x \lor a_3^x \lor -a_3^y \lor a_3^y)</td>
<td>(X = 3 \lor (Y = 1) \lor (Y = 2) \lor (Y = 4))</td>
<td>((-a_3^x \lor a_3^x) \lor (a_3^x \lor -a_3^y) \lor a_3^y)</td>
<td>((-a_3^x \lor a_3^x) \lor (a_3^x \lor -a_3^y) \lor a_3^y)</td>
</tr>
<tr>
<td>(\neg(X = 4 \land Y = 4))</td>
<td>(-a_4^x \lor a_4^x \lor -a_4^y \lor a_4^y)</td>
<td>(X = 4 \lor (Y = 1) \lor (Y = 2) \lor (Y = 3))</td>
<td>((-a_4^x \lor a_4^x) \lor (a_4^x \lor -a_4^y) \lor a_4^y)</td>
<td>((-a_4^x \lor a_4^x) \lor (a_4^x \lor -a_4^y) \lor a_4^y)</td>
</tr>
</tbody>
</table>

Table 4.8 gives an example of conflict clauses on the constraint \(X \neq Y\) and of support clauses on the constraint \(X \geq Y\). The example uses two common SAT encodings (i.e., the sparse encoding and the order encoding) and the representative-sparse encoding (rep.-sparse with two variables at the first level, see Section 3.6.1). Note that using conflict clauses for the constraint \(X \geq Y\) and support clauses on the constraint \(X \neq Y\) can generate very long clauses. In particular, the larger the domains of variables are, the longer the generated clauses of SAT instances are.
Table 4.8: The different encodings for translating CSP to SAT, illustrated on constraints $X \neq Y$ using conflict clauses and $X \geq Y$ using support clauses, where $X$ and $Y$ have a domain of 8 values \{1, 2, ..., 8\}.

<table>
<thead>
<tr>
<th>enc</th>
<th>$X \neq Y$</th>
<th>$X &gt; Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CSP</td>
<td>SAT</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 1 \land Y = 1 \right)$</td>
<td>$\neg d_1^X \lor \neg d_1^Y$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 2 \land Y = 2 \right)$</td>
<td>$\neg d_2^X \lor \neg d_2^Y$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 3 \land Y = 3 \right)$</td>
<td>$\neg d_3^X \lor \neg d_3^Y$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 4 \land Y = 4 \right)$</td>
<td>$\neg d_4^X \lor \neg d_4^Y$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 5 \land Y = 5 \right)$</td>
<td>$\neg d_5^X \lor \neg d_5^Y$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 6 \land Y = 6 \right)$</td>
<td>$\neg d_6^X \lor \neg d_6^Y$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 7 \land Y = 7 \right)$</td>
<td>$\neg d_7^X \lor \neg d_7^Y$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 8 \land Y = 8 \right)$</td>
<td>$\neg d_8^X \lor \neg d_8^Y$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 1 \land Y = 1 \right)$</td>
<td>$\alpha_1^X \lor \alpha_1^Y$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 2 \land Y = 2 \right)$</td>
<td>$\neg \alpha_2^X \lor \alpha_2^X \lor \neg \alpha_2^Y \lor \alpha_2^Y$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 3 \land Y = 3 \right)$</td>
<td>$\neg \alpha_3^X \lor \alpha_3^X \lor \neg \alpha_3^Y \lor \alpha_3^Y$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 4 \land Y = 4 \right)$</td>
<td>$\neg \alpha_4^X \lor \alpha_4^X \lor \neg \alpha_4^Y \lor \alpha_4^Y$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 5 \land Y = 5 \right)$</td>
<td>$\neg \alpha_5^X \lor \alpha_5^X \lor \neg \alpha_5^Y \lor \alpha_5^Y$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 6 \land Y = 6 \right)$</td>
<td>$\neg \alpha_6^X \lor \alpha_6^X \lor \neg \alpha_6^Y \lor \alpha_6^Y$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 7 \land Y = 7 \right)$</td>
<td>$\neg \alpha_7^X \lor \alpha_7^X \lor \neg \alpha_7^Y \lor \alpha_7^Y$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 8 \land Y = 8 \right)$</td>
<td>$\neg \alpha_8^X \lor \alpha_8^X \lor \neg \alpha_8^Y \lor \alpha_8^Y$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 1 \land Y = 1 \right)$</td>
<td>$\neg g_1^X \lor \neg x_1^X \lor \neg g_1^Y \lor \neg x_1^Y$ $X = 1 \rightarrow Y \leq 1$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 2 \land Y = 2 \right)$</td>
<td>$\neg g_2^X \lor \neg x_2^X \lor \neg g_2^Y \lor \neg x_2^Y$ $X = 2 \rightarrow Y \leq 2$</td>
</tr>
<tr>
<td>$\left(m=2\right)$</td>
<td>$\neg \left( X = 3 \land Y = 3 \right)$</td>
<td>$\neg g_3^X \lor \neg x_3^X \lor \neg g_3^Y \lor \neg x_3^Y$ $X = 3 \rightarrow Y \leq 3$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 4 \land Y = 4 \right)$</td>
<td>$\neg g_4^X \lor \neg x_4^X \lor \neg g_4^Y \lor \neg x_4^Y$ $X = 4 \rightarrow Y \leq 4$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 5 \land Y = 5 \right)$</td>
<td>$\neg g_5^X \lor \neg x_5^X \lor \neg g_5^Y \lor \neg x_5^Y$ $X = 5 \rightarrow Y \leq 5$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 6 \land Y = 6 \right)$</td>
<td>$\neg g_6^X \lor \neg x_6^X \lor \neg g_6^Y \lor \neg x_6^Y$ $X = 6 \rightarrow Y \leq 6$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 7 \land Y = 7 \right)$</td>
<td>$\neg g_7^X \lor \neg x_7^X \lor \neg g_7^Y \lor \neg x_7^Y$ $X = 7 \rightarrow Y \leq 7$</td>
</tr>
<tr>
<td></td>
<td>$\neg \left( X = 8 \land Y = 8 \right)$</td>
<td>$\neg g_8^X \lor \neg x_8^X \lor \neg g_8^Y \lor \neg x_8^Y$ $X = 8 \rightarrow Y \leq 8$</td>
</tr>
</tbody>
</table>
4.2 Encoding the At-Most-One Constraint

The sparse encoding (see Section 3.1) is the most straightforward to translate CSP domains into SAT. The sparse encoding requires the ALO and AMO constraints to enforce that a CSP variable is assigned to exactly one value within its domain. Whereas ALO can be easily represented by a single SAT clause, SAT encodings of the AMO constraint is more complicated and has been intensively studied [Pre07b, Pre07a, KK07, FG10, Che10, BHN13, HN13b]. Other motivations also come from the many applications such as computer tomographs [BB03], partial Max-SAT [ACLM09, ACM10], or cardinality constraints [FG10]. To avoid the confusion between a SAT encoding of a finite CSP domain and a SAT encoding of the AMO constraint, this thesis will use the term AMO SAT-encoding for a SAT encoding of the AMO constraint.

Inspired by many interesting and recent results [FPDN01, Pre07b, KK07, Pre07a, FG10, Che10], the purpose of this section is to: (1) survey the widely used AMO SAT-encodings; (2) propose a new AMO SAT-encoding; and 3) presents several important and interesting observations. For example, several AMO SAT-encodings are exactly the SAT encodings of a finite CSP domain.

In the sparse encoding (see Section 3.1), if a propositional variable is used to represent the binding of a CSP variable $V$ with the domain of $n$ values to a particular value, then the AMO constraint requires that at most one of the $n$ propositional variables is bound to 1. Herein, this will be denoted by $AMO(x_1, \ldots, x_n)$, where $x_i$, $1 \leq i \leq n$, is propositional variable.

Before giving a brief survey of AMO SAT-encodings, this section first defines several important notions and notations, mainly following Frisch and Giannoros [FG10].

**Definition 4.2.1 (Correctness)** Let $X = \{x_i \mid 1 \leq i \leq n, n \in \mathbb{N}\}$ be a finite set of propositional variables, let $A$ be a finite, possibly empty set of auxiliary propositional variables, and let $\phi(X, A)$ be a propositional formula in conjunctive normal form (CNF) encoding the constraint $AMO(x_1, \ldots, x_n)$. The encoding $\phi(X, A)$ is correct if and only if:

- any partial interpretation $\hat{x}$ that satisfies $AMO(x_1, \ldots, x_n)$ can be extended to a complete interpretation that satisfies $\phi(X, A)$, and

- for any partial interpretation $\hat{x}$ for $X$ which assigns more than one variable of $X$ to 1, unit propagation (UP) detects a conflict, i.e., repeated applications of UP yield the empty clause.

It is well-known that one usually considers whether UP in SAT solvers achieve pruning in a similar way to CP solvers applying local consistency to the original CSP (e.g., arc consistency or forward checking).
Definition 4.2.2 Unit propagation (UP) of a SAT encoding of the constraint $AMO(x_1, \ldots, x_n)$ achieves the same pruning as arc consistency on the original CSP, which is referred to as the UPaAC property from here on, if two following conditions hold [FG10]:

- at-most-one propositional variable in $X$ is assigned to 1, and if
- any variable $x_i \in X$ is assigned to 1, then all the other variables occurring in $X$ must be assigned to 0 by using UP.

In the following sections, generally $AMO(X)$ and $ALO(X)$ denote the at-most-one and at-least-one clauses for the set of propositional variables $X = \{x_1, \ldots, x_n\}$, respectively, and we define $EO(X) := AMO(X) \land ALO(X)$, namely exactly-one clauses, for the set of propositional variables $X$. Our goal is to encode the constraint $AMO(X)$ into CNF. For the sake of convenience, a running example illustrates these encodings through the set consisting of eight Boolean variables, $X = \{x_1, \ldots, x_8\}$.

4.2.1 The AMO Pairwise Encoding

This encoding has several different names: the naive encoding [Sin05, KK07], the pairwise encoding [SL07, Pre07a], or the binomial encoding [FG10]. This thesis refers to it as the AMO pairwise encoding. The idea of this encoding is to express that all possible combinations of two variables are not simultaneously assigned to 1. Therefore as soon as one literal is assigned to 1, the all others must be assigned to 0:

$$\bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} \neg(x_i \land x_j) \equiv \bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} \neg x_i \lor \neg x_j.$$  

Example 4.2.3 In the running example, the AMO pairwise encoding produces the following clauses:

$$
\neg x_1 \lor \neg x_2 \land \neg x_1 \lor \neg x_3 \land \neg x_1 \lor \neg x_4 \land \ldots \land \neg x_1 \lor \neg x_8 \land \\
\neg x_2 \lor \neg x_3 \land \neg x_2 \lor \neg x_4 \land \ldots \land \neg x_2 \lor \neg x_8 \land \\
\neg x_3 \lor \neg x_4 \land \ldots \land \neg x_3 \lor \neg x_8 \land \\
\vdots \\
\neg x_7 \lor \neg x_8 
$$

The AMO pairwise encoding is a traditional way of encoding the AMO constraint into SAT. Although this encoding does not need any auxiliary variables, it requires a quadratic number of clauses. Consequently, this method may result in large formulas on problems with large domains. Nevertheless, the AMO pairwise encoding is not only widely used in practice, but also able to combine with other encodings [KK07, Vel07a, Che10, BHN13, HN13b]. It is important to stress that the AMO pairwise encoding has the UPaAC property (see Table 4.9 on page 72).
4.2. Encoding the At-Most-One Constraint

4.2.2 The AMO Binary Encoding

Frisch et al. [FPDN01, FPDN05] proposed the binary encoding. Independently, Prestwich introduced it as the bitwise encoding [Pre07a, Pre09]) and used it to successfully solve a number of large instances of CSPs with a standard SAT solver [Pre07b, Pre07a]. This thesis refers to it as the AMO binary encoding.

The AMO binary encoding requires a set of auxiliary Boolean variables \( \{b_1, \ldots, b_{\lceil \log_2 n \rceil}\} \) with a set of clauses:

\[
\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{\lceil \log_2 n \rceil} x_i \rightarrow \phi(i, j) \equiv \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{\lceil \log_2 n \rceil} \neg x_i \lor \phi(i, j),
\]

where \( \phi(i, j) \) denotes \( b_j \) (or \( \neg b_j \)) if the bit \( j \) of \( i \) is represented by a binary string is 1 (or 0).

The idea is to create the different sequences of \( \lceil \log_2 n \rceil \)-tuples \( b_j, 1 \leq j \leq \lceil \log_2 n \rceil \), such that whenever any \( x_i \) is assigned to 1 for all \( i \), then one immediately infers that the other variables \( x_{i'} \) must be assigned to 0, for any \( 1 \leq i' \neq i \leq n \).

**Example 4.2.4** The running example is represented by the AMO binary encoding as follows:

\[
\begin{align*}
    x_1 & \rightarrow \neg b_1 \land x_2 \rightarrow b_1 \land x_3 \rightarrow \neg b_1 \land \ldots \land x_8 \rightarrow b_1 \\
    x_1 & \rightarrow \neg b_2 \land x_2 \rightarrow \neg b_2 \land x_3 \rightarrow b_2 \land \ldots \land x_8 \rightarrow b_2 \\
    x_1 & \rightarrow \neg b_3 \land x_2 \rightarrow \neg b_3 \land x_3 \rightarrow \neg b_3 \land \ldots \land x_8 \rightarrow b_3
\end{align*}
\]

which is semantically equivalent to

\[
\begin{align*}
    \neg x_1 \lor \neg b_1 \land \neg x_2 \lor b_1 \land \neg x_3 \lor \neg b_1 \land \ldots \land \neg x_8 \lor b_1 \\
    \neg x_1 \lor \neg b_2 \land \neg x_2 \lor \neg b_2 \land \neg x_3 \lor b_2 \land \ldots \land \neg x_8 \lor b_2 \\
    \neg x_1 \lor \neg b_3 \land \neg x_2 \lor \neg b_3 \land \neg x_3 \lor \neg b_3 \land \ldots \land \neg x_8 \lor b_3
\end{align*}
\]

There are two important remarks about the AMO binary encoding:

1. The AMO binary encoding has the UPaAC property (see [FG10]).

2. The auxiliary variables used by the AMO binary encoding (i.e., \( b_j \)) exactly correspond to the variables used by the log encoding of a finite CSP domain to SAT (see the log encoding in Section 3.3).

4.2.3 The AMO Commander Encoding

Klieber and Kwon [KK07] described the AMO commander encoding by dividing the set of propositional variables \( X = \{x_1, \ldots, x_n\} \) into \( m \) (between 1 and \( n \)) disjoint subsets denoted by \( \{G_1, \ldots, G_m\} \), and introducing a commander variable \( c_i \) for each subset \( G_i, 1 \leq i \leq m \). The AMO commander encoding is defined as follows.
1. Exactly one variable in each set \( G_i \cup \{ \neg c_i \} \) is assigned to 1:

\[
\bigwedge_{i=1}^{m} EO(\{ \neg c_i \} \cup G_i) = \bigwedge_{i=1}^{m} AMO(\{ \neg c_i \} \cup G_i) \land \bigwedge_{i=1}^{m} ALO(\{ \neg c_i \} \cup G_i),
\]

whereas the ALO constraint is easily translated into a single clause, AMO can be encoded either by the AMO pairwise or commander encoding.

2. At most one commander variable is assigned to 1. This constraint can be encoded either by the AMO pairwise encoding or by another encoding (even by a recursive application of the AMO commander encoding):

\[
\bigwedge_{i=1}^{m} AMO(c_i).
\]

Example 4.2.5 In the running example, by selecting \( m = 4 \), dividing the set \( X = \{ x_1, \ldots, x_8 \} \) into the disjoint subsets \( G_1 = \{ x_1, x_2 \} \), \( G_2 = \{ x_3, x_4 \} \), \( G_3 = \{ x_5, x_6 \} \), and \( G_4 = \{ x_7, x_8 \} \), and adding four commander variables \( c_1, c_2, c_3, \) and \( c_4 \) we obtain:

\[
AMO(\neg c_1, x_1, x_2) \land (\neg c_1 \lor x_1 \lor x_2) \land \\
AMO(\neg c_2, x_3, x_4) \land (\neg c_2 \lor x_3 \lor x_4) \land \\
AMO(\neg c_3, x_5, x_6) \land (\neg c_3 \lor x_5 \lor x_6) \land \\
AMO(\neg c_4, x_7, x_8) \land (\neg c_4 \lor x_7 \lor x_8).
\]

By using the AMO pairwise encoding, the above formula is further encoded:

\[
c_1 \lor \neg x_1 \land c_1 \lor \neg x_2 \land \neg x_1 \lor \neg x_2 \land \neg c_1 \lor x_1 \lor x_2 \land \\
c_2 \lor \neg x_3 \land c_2 \lor \neg x_4 \land \neg x_3 \lor \neg x_4 \land \neg c_2 \lor x_3 \lor x_4 \land \\
c_3 \lor \neg x_5 \land c_3 \lor \neg x_6 \land \neg x_5 \lor \neg x_6 \land \neg c_3 \lor x_5 \lor x_6 \land \\
c_4 \lor \neg x_7 \land c_4 \lor \neg x_8 \land \neg x_7 \lor \neg x_8 \land \neg c_4 \lor x_7 \lor x_8.
\]

At most one among the commander variables is assigned to 1:

\[
AMO(c_1, c_2, c_3, c_4) \equiv (\neg c_1 \lor \neg c_2) \land (\neg c_1 \lor \neg c_3) \land \\
(\neg c_1 \lor \neg c_4) \land (\neg c_2 \lor \neg c_3) \land \\
(\neg c_2 \lor \neg c_4) \land (\neg c_3 \lor \neg c_4).
\]

Compared with the AMO pairwise encoding, the AMO commander encoding requires a fewer number of clauses but introduces auxiliary variables. The commander method also has the UPaAC property (see Table 4.9 on page 72).

4.2.4 The AMO Product Encoding

Chen [Che10] proposed an AMO encoding, named the AMO product encoding. Instead of encoding the AMO constraint \( AMO(x_1, \ldots, x_n) \), the author encoded a constraint consisting of \( n \) corresponding points, denoted by

\[
\{ (u_i, v_j) \mid 1 \leq i \leq p, 1 \leq j \leq q, p \times q \geq n \}.
\]

The idea can be explained as follows:
4.2. Encoding the At-Most-One Constraint

1. Each variable $x_k, 1 \leq k \leq n$ is mapped onto a corresponding point $(u_i, v_j)$, where $u_i \in U = \{u_1, \ldots, u_p\}$, and $v_i \in V = \{v_1, \ldots, v_q\}$.

2. Then, the AMO product encoding is obtained as:

$$AMO(X) = AMO(U) \land AMO(V) \land \bigwedge_{1 \leq i \leq p, 1 \leq j \leq q} \left((\neg x_k \lor u_i) \land (\neg x_k \lor v_j)\right),$$

where $AMO(U)$ and $AMO(V)$ can be encoded by either another encoding or a recursive application of the AMO product encoding.

**Example 4.2.6** With regard to the running example, by choosing $p = 3$, $q = 3$, and using the AMO pairwise encoding for $AMO(U)$ and $AMO(V)$, the derived clauses are:

- $AMO(U) = (\neg u_1 \lor \neg u_2) \land (\neg u_1 \lor \neg u_3) \land (\neg u_2 \lor \neg u_3)$
- $AMO(V) = (\neg v_1 \lor \neg v_2) \land (\neg v_1 \lor \neg v_3) \land (\neg v_2 \lor \neg v_3)$
- $AMO(X) = AMO(U) \land AMO(V) \land$
  - $(\neg x_1 \lor u_1) \land (\neg x_2 \lor u_2) \land (\neg x_2 \lor v_1) \land (\neg x_3 \lor u_3) \land (\neg x_3 \lor v_1) \land (\neg x_4 \lor u_1) \land (\neg x_4 \lor v_2) \land (\neg x_5 \lor u_2) \land (\neg x_5 \lor v_2) \land (\neg x_6 \lor u_3) \land (\neg x_6 \lor v_2) \land (\neg x_7 \lor u_1) \land (\neg x_7 \lor v_3) \land (\neg x_8 \lor u_2) \land (\neg x_8 \lor v_3)$

There are two important remarks about the AMO product encoding:

1. It has the UPaAC property [Che10].

2. The auxiliary variables used by this AMO product encoding (i.e., $(u_i, v_j)$) exactly correspond to the variables used by the representative-sparse encoding, which encodes a finite CSP domain into SAT proposed by Barahona et al. [BHN14b] (see Table 4.9 on page 72).

4.2.5 The AMO Sequential Counter Encoding

By building a count-and-compare hardware circuit and translating this circuit to an equivalent CNF formula, Sinz [Sin05] introduced cardinality constraints $\leq_k (x_1, \ldots, x_n)$. Here, we only consider the case $k = 1$ and obtain the AMO sequential counter encoding [Sin05, SL07]:

$$\left((\neg x_1 \lor s_1) \land (\neg x_n \lor s_{n-1})\right) \bigwedge_{1 \leq i \leq n} \left((\neg x_i \lor s_i) \land (\neg s_{i-1} \lor s_i) \land (\neg x_i \lor \neg s_{i-1})\right),$$

where $s_i, 1 \leq i \leq n - 1$, are auxiliary variables. The above formula is a constraint which guarantees that whenever any $x_i, 1 \leq i \leq n$, is assigned to 1, then the other variables $x_{i'}$ must be assigned to 0, for any $1 \leq i' \neq i \leq n$. 
Example 4.2.7 By introducing seven auxiliary variables for the AMO sequential counter encoding, we obtain the following formula:

\[
\neg x_1 \lor s_1 \land \\
\neg x_2 \lor s_2 \land \neg s_1 \lor s_2 \land \neg x_2 \lor \neg s_1 \land \\
\neg x_3 \lor s_3 \land \neg s_2 \lor s_3 \land \neg x_3 \lor \neg s_2 \land \\
\neg x_4 \lor s_4 \land \neg s_3 \lor s_4 \land \neg x_4 \lor \neg s_3 \land \\
\neg x_5 \lor s_5 \land \neg s_4 \lor s_5 \land \neg x_5 \lor \neg s_4 \land \\
\neg x_6 \lor s_6 \land \neg s_5 \lor s_6 \land \neg x_6 \lor \neg s_5 \land \\
\neg x_7 \lor s_7 \land \neg s_6 \lor s_7 \land \neg x_7 \lor \neg s_6 \land \\
\neg x_8 \lor \neg s_7.
\]

Two notes are worth mentioning for this encoding:

1. The AMO sequential counter encoding has the UPaAC property (see [SL07, FG10] and Table 4.9 on page 72).

2. The auxiliary variables used by this AMO sequential counter encoding (i.e., \(s_i\)) exactly correspond to the variables used by one of the following encodings: unary representation [BB03], order encoding [CB94, TTKB09], regular encoding [AM04], ladder encoding [GPS02], and relaxed ladder encoding [Pre07a] (see [HN13b, BHN13] and Section 4.2.7.1).

4.2.6 The AMO Bimander Encoding

This thesis proposes a new AMO SAT-encoding, the so-called AMO bimander encoding. The general idea of the new encoding is based on both the ideas of the AMO binary encoding and the AMO commander encoding.

We partition a set of propositional variables \(X = \{x_1, \ldots, x_n\}\) into \(m\) (between 1 and \(n\)) disjoint subsets \(\{G_1, \ldots, G_m\}\) such that each subset \(G_i\) consists of \(g = \lceil \frac{n}{m} \rceil\) variables. However, instead of introducing commander variables like in the AMO commander encoding, the AMO bimander encoding introduce a set of auxiliary propositional variables \(b_1, \ldots, b_{\log_2 m}\) as in the AMO binary encoding. The variables \(b_1, \ldots, b_{\log_2 m}\) play the role of the commander variables in the AMO commander encoding.

The AMO bimander encoding is the conjunction of the following clauses:

1. At most one variable in each subset can be 1. One must encode this constraint for each subset \(G_i, 1 \leq i \leq m\), by using the AMO pairwise encoding:

\[
\bigwedge_{i=1}^{m} AMO(G_i). \tag{4.1}
\]

2. The following clauses are generated by the constraints between each variable and commander variables in a subset:

\[
\bigwedge_{i=1}^{m} \bigwedge_{h=1}^{g \lfloor \log_2 m \rfloor} \bigwedge_{j=1}^{g \lfloor \log_2 m \rfloor} x_{i,h} \to \phi(i,j) \equiv \bigwedge_{i=1}^{m} \bigwedge_{h=1}^{g \lfloor \log_2 m \rfloor} \bigwedge_{j=1}^{g \lfloor \log_2 m \rfloor} \neg x_{i,h} \lor \phi(i,j), \tag{4.2}
\]
4.2. Encoding the At-Most-One Constraint

where \( \phi(i,j) \) denotes \( b_j \) (or \(-b_j\)) if the bit \( j \) of \( i - 1 \) represented by a unique binary string is 1 (or 0).

**Example 4.2.8** In the running example by choosing \( m = \lfloor \sqrt{n} \rfloor = 3 \) we obtain \( G_1 = \{x_1, x_2, x_3\}, G_2 = \{x_4, x_5, x_6\}, \text{ and } G_3 = \{x_7, x_8\} \). Consequently, Formula 4.1 generates the following set of clauses:

\[
AMO(x_1, x_2, x_3) \land AMO(x_4, x_5, x_6) \land AMO(x_7, x_8).
\]

In the second step, we introduce a set of auxiliary variables \( \{b_1, \ldots, b_{\log_2 m}\} = \{b_1, b_2\} \). Then, the following set of clauses is generated:

\[
-x_1 \lor -b_1 \land -x_4 \lor b_1 \land -x_7 \lor -b_1 \land \\
x_1 \lor -b_2 \land -x_4 \lor -b_2 \land -x_7 \lor b_2 \land \\
x_2 \lor -b_1 \land -x_5 \lor b_1 \land -x_8 \lor -b_1 \land \\
x_2 \lor -b_2 \land -x_5 \lor -b_2 \land -x_8 \lor b_2 \\
x_3 \lor -b_1 \land -x_6 \lor b_1 \land \\
x_3 \lor -b_2 \land -x_6 \lor -b_2 \land
\]

Compared with the AMO commander encoding, the AMO bimander encoding does not require any constraint among the sequences of auxiliary variables because any combination of such variables \( b_1, \ldots, b_{\log_2 m} \) of a corresponding subset is different from any combinations of all the other groups. Let us prove some important properties of the AMO bimander encoding.

**Theorem 4.2.9 (Correctness)** The AMO bimander encoding is correct.

**Proof** Assume that we have a partial interpretation \( \hat{x} = (x_1, \ldots, x_l), 1 \leq l \leq n \), with at most one variable assigned to \( I \). In case all variables are assigned to 0), then condition (4.1) is trivially satisfied. The same holds for condition (4.2). In case that only one variable, say \( x_i, 1 \leq i \leq n \), is assigned to \( I \), then there is a corresponding sequence of \( l \) values assigned to the corresponding sequence of \( \{b_1, \ldots, b_{\log_2 m}\} \). Hence, condition (4.2) is satisfied as well. Therefore, the partial interpretation \( \hat{x} \) can possibly be extended to a complete interpretation that satisfies two conditions.

Now suppose that we have a partial interpretation \( \hat{x} = (x_1, \ldots, x_l), 1 \leq l \leq n \), with more than one variable assigned to \( I \). Assume that \( x_i = 1 \) and \( x_j = 1 \), for \( 1 \leq i \neq j \leq l \). In order to satisfy the condition (4.1), the variables \( x_i \) and \( x_j \) must belong to different subsets. That leads to two differently corresponding patterns of the sequence \( \{b_1, \ldots, b_{\log_2 m}\} \) which are assigned to \( I \). As a result, the sequence contains one propositional variable \( b_k, 1 \leq k \leq \lfloor \log_2 m \rfloor \) that is assigned to both \( I \) and 0 at the same time. In other words, there exists a clause, which is of the form \( b_k \land -b_k \). Hence, if any partial interpretation has more than one variable assigned to \( I \), then UP produces an empty clause. It means that this partial interpretation can not be extended to a complete interpretation.

Follow Definition 4.2.1 we conclude that the AMO bimander encoding correctly encodes the AMO constraint into SAT.
Theorem 4.2.10 (Strength) The AMO bimander has the UPaAC property.

Proof Suppose that we have a partial interpretation \( \hat{x} = (x_1, \ldots, x_l) \), \( 1 \leq l \leq n \), where \( I \) is assigned to exactly one variable. Now we will show that UP will assign all other variables to 0. Assume that variable \( x_{i,j} = 1 \), which is the \( j \)th variable in the subset \( G_i, 1 \leq i \leq m \), then this interpretation forces a corresponding pattern of the sequence \( \{b_1, \ldots, b_{\log_2 m}\} \) to \( I \). Because \( x_{i,j} = 1 \), all other variables in the subset \( G_i \) are set to 0, followed by condition (4.1). Due to condition (4.2), all the other variables in the subsets \( G_{i'} \), \( 1 \leq i' \neq i \leq m \) are set to 0 because they have different patterns of the sequence \( \{b_1, \ldots, b_{\log_2 m}\} \) corresponding to \( x_{i,j} = 1 \). Follow Definition 4.2.2 we conclude that UP on the AMO bimander encoding achieves arc consistency.

Complexity We partition a set of propositional variables \( X = \{x_1, \ldots, x_n\} \) into \( m \) \((1 \leq m \leq n)\) disjoint subsets \( \{G_1, \ldots, G_m\} \) of size \( g = \left\lceil \frac{n}{m} \right\rceil \) variables. As we supposed, we need a set of \( \lceil \log_2 m \rceil \) auxiliary variables. Condition (4.1) uses the AMO pairwise encoding for \( m \) groups, and each group consists of \( g \) variables. Consequently, we have \( m \ast g(\frac{g-1}{2}) = n \left\lceil \frac{n}{2} \right\rceil \) new clauses. Condition (4.2) requires \( m \ast [g \ast \log_2 m] = n \ast [\log_2 m] \) clauses. Hence, the encoding uses \( \frac{n(\frac{n}{2} - 1)}{2} + n \left\lfloor \log_2 m \right\rfloor = \frac{n^2}{4} + n \left\lfloor \log_2 m \right\rfloor - \frac{n}{2} \) clauses.

Generalization It is worth pointing out that the AMO bimander encoding can be easily generalized to encode the at-most-\( k \) constraint. Again, the set of variables is partitioned into several subsets.

1. For each subset, the at-most-\( k \) constraint is encoded by a modified pairwise (or another) encoding.

2. The constraints between each variable and the commander variables in a subset are encoded by the following clauses:

\[
\bigwedge_{i=1}^{m} \bigwedge_{h=1}^{g} \bigvee_{l=1}^{k} \bigwedge_{j=1}^{\lfloor \log_2 m \rfloor} \neg x_{i,h} \lor \phi(i, h, l, j),
\]

where \( \phi(i, h, l, j) \) denotes \( b_{i,j} \) (or \( \neg b_{i,j} \)) if the bit \( j \) of \( i - 1 \) represented by a binary string is 1 (or 0).

Special Cases One should observe that the AMO bimander encoding is a general case of several encodings. For example,

- The AMO pairwise encoding is a special case of the AMO bimander encoding by setting \( m = 1 \).

- The AMO commander encoding is a special case of the AMO bimander encoding by setting \( m = 2 \) (when both encodings divide into 2 subsets).
4.2. Encoding the At-Most-One Constraint

- The AMO binary encoding is a special case of the AMO bimander encoding by setting \( m = n \).

4.2.7 Comparison

4.2.7.1 Similarities of Some AMO Encodings

Gent and Nightingale [GN04] used the ladder structure, originally proposed by Gent et al. [GPS02], to describe a new encoding of the constraint \( \text{alldifferent}(x_1, \ldots, x_n) \) into SAT. The ladder structure consisting of \( n - 1 \) auxiliary variables \( y_1, \ldots, y_{n-1} \) has no adjacent pair of variables \( y_r \) and \( y_{r+1}, 1 \leq r \leq n - 2 \), where \( y_r = 0 \land y_{r+1} = 1 \). Another way of stating the ladder structure is that it consists of a sequence of 0 or more 1, and all following variables are assigned to 0. For convenience, we assume an extra bounding variable \( y_0 = 1 \) and \( y_n = 0 \). To forbid all invalid structures, the authors add the ladder validity clauses:

\[
\bigwedge_{r=1}^{n} (y_r \rightarrow y_{r-1}) \equiv \bigwedge_{r=1}^{n} (\neg y_r \lor y_{r-1})
\] (4.3)

Formula 4.3 guarantees that whenever a variable, say \( y_r (1 \leq r \leq n - 1) \), is set to 0, all variables following, \( y_{r+1} (1 \leq r \leq n - 1) \), are also set to 0, by unit propagation. To connect auxiliary variables \( \{y_1, \ldots, y_{n-1}\} \) with \( \{x_1, \ldots, x_n\} \), the channelling constraints are used between two representations:

\[
\bigwedge_{r=1}^{n} [(y_{r-1} \land \neg y_r) \leftrightarrow x_r]
\] (4.4)

The ladder encoding consists of set of clauses in Formula 4.3 and Formula 4.4:

\[
\bigwedge_{r=1}^{n} [(\neg y_{r-1} \lor y_r \lor x_r) \land (\neg x_r \lor y_r) \land (\neg x_r \lor \neg y_{r-1}) \land (\neg y_r \lor y_{r-1})]
\] (4.5)

Instead of using Formula 4.4, Prestwich introduces a relaxed challenging constraint

\[
\bigwedge_{r=1}^{n} [(y_{r-1} \land \neg y_r) \leftarrow x_r]
\] (4.6)

Prestwich supposed the relaxed ladder encoding, which consists of set of clauses in Formula 4.3 and Formula 4.6 [Pre07a].

This section addresses the relationships among several SAT encodings. To do so, we point out the similarity among the AMO sequential counter encoding [Sin05], the AMO relaxed ladder encoding [Pre07a], and the AMO ladder encoding [GN04]. These encodings are used to encode the AMO encoding.

**Proposition 4.2.11** The AMO sequential counter encoding is exactly the AMO relaxed ladder encoding, and these two encodings are the AMO ladder encoding without a set of redundant clauses.
Proof First, we will point out that the AMO sequential counter encoding is the ladder encoding without a set of redundant clauses. The sequence of auxiliary variables used by the ladder structure is a non-increasing vector \([GN04]\). On the contrary, the AMO sequential counter encoding \([Sin05]\) needs a sequence of auxiliary variables \(\langle s_1, \ldots, s_{n-1} \rangle\), under the condition that these variables create a non-decreasing vector. Without losing the correctness property, we consider the sequence of auxiliary variables \(\langle s_1, \ldots, s_{n-1} \rangle\) in \([Sin05]\) to play the role of the sequence of auxiliary variables \(\langle y_1, \ldots, y_{n-1} \rangle\) in \([GN04]\) with the condition among \(\langle s_1, \ldots, s_{n-1} \rangle\) reversed (i.e., the sequence is a non-increasing vector). Now the ladder validity clauses are represented as follows:

\[
\bigwedge_{i=1}^{n} (s_{i-1} \rightarrow s_i) \equiv \bigwedge_{i=1}^{n} (\neg s_{i-1} \lor s_i)
\]  

(4.7)

under the assumption that (for convenience):

\[
s_0 = 0 \land s_n = 1.
\]  

(4.8)

Consider the channelling clauses \([GN04]\)

\[
\bigwedge_{i=1}^{n} (s_i \land \neg s_{i-1}) \leftrightarrow x_i
\]  

(4.9)

and combine with the conjunction of (4.7) and (4.8) one obtains the ladder encoding:

\[
\bigwedge_{i=1}^{n} [(\neg s_{i-1} \lor s_i) \land (\neg s_i \lor s_{i-1} \lor x_i) \land (\neg x_i \lor s_i) \land (\neg x_i \lor \neg s_{i-1})]
\]  

(4.10)

It is easy to check that the clauses \((\neg s_i \lor s_{i-1} \lor x_i)\) occurring in (4.10) is redundant since it does not affect the correctness of the AMO constraint. Removing these redundant clauses leads to:

\[
\bigwedge_{i=1}^{n} [(\neg s_{i-1} \lor s_i) \land (\neg x_i \lor s_i) \land (\neg x_i \lor \neg s_{i-1})]
\]  

(4.11)

Replacing (4.11) by (4.8) yields the AMO sequential counter encoding:

\[
(\neg x_1 \lor s_1) \land (\neg x_n \lor \neg s_{n-1}) \bigwedge_{1<i<n} [(\neg x_i \lor s_i) \land (\neg s_{i-1} \lor s_i) \land (\neg x_i \lor \neg s_{i-1})]
\]  

(4.12)

Thus the ladder encoding without a set of redundant clauses is exactly the AMO sequential counter encoding.

Second, instead of adding (4.9), Prestwich added the following condition:

\[
\bigwedge_{i=1}^{n} [(s_i \land \neg s_{i-1}) \leftrightarrow x_i].
\]  

(4.13)

This condition eliminates the redundant clauses in (4.10) and immediately leads to the AMO sequential counter encoding.

As a result, the AMO sequential counter encoding and the AMO relaxed ladder encoding are identical.
4.2. Encoding the At-Most-One Constraint

Argelich et al. [ACLM10] also noticed that the AMO sequential counter encoding is a reformulation of a regular encoding [AM04]. One should observe that Tamura et al. [TTKB09] used the ladder structure in the order encoding to translate CSPs to SAT in their SAT-based solver, Sugar. Bailleux et al. [BB03] also used this structure, under the name unary representation, during their translation of cardinality constraints and pseudo-Boolean constraints to SAT formulas [BB03, ES06, BBR09].

Recently, Martins et al. [MML11] compared both encodings, the AMO sequential counter encoding and the AMO ladder encoding, and consequently the experiment results obtained show an insignificant difference between the two encodings.

In conclusion, we have shown the relationships among the ladder structure, AMO sequential counter encoding, relaxed ladder encoding, regular encoding, unary representation, and order encoding.

### 4.2.7.2 Overview

Table 4.9 presents the key features of many approaches for encoding the AMO constraint (column enc). The columns clauses and aux vars depict the number of required clauses and auxiliary variables, respectively. The column UPaAC indicates whether the encoding has the UPaAC property. The column origin refers to the original publications where the encoding had been introduced. The disjointed subsets by dividing the set of propositional variables \(\{x_1, \ldots, x_n\}\) in the AMO bimander encoding is denoted by \(m\). In addition to the encodings of the AMO constraint presented in previous parts, we also mention other encodings that are mainly used for cardinality constraints, the at-most-k constraints \(\leq_k (x_1, \ldots, x_n)\). In this thesis, we only consider these for the case \(k = 1\).

As we can see in Table 4.9, the AMO bimander encoding requires the least auxiliary variables – with the exception of the AMO pairwise encoding – among known encodings. The totalizer encoding proposed by Bailleux et al. [BB03] requires clauses of size at most 3, and the AMO commander encoding proposed by Klieber and Kwon [KK07] needs \(m\) (number of disjointed subsets) clauses of size \(\lceil \frac{n}{m} + 1 \rceil\), whereas the AMO product, sequential counter, binary and bimander encoding require only binary clauses. Note that binary clauses may speed up SAT solvers significantly compared to longer clauses.

Figures 4.2, 4.3, and 4.4 show the number of variables and the number of clauses required by different AMO encodings for the pigeon-hole, Langford, and all-interval series problems, respectively. In the figures, the AMO bimander, AMO binary, AMO commander, AMO product, AMO pairwise, and AMO sequential counter encodings are abbreviated as bim, bin, cmd, pro, pw, and seg, respectively.

As can be seen, the AMO pairwise encoding requires the smallest number of variables, however the largest number of clauses. On the contrary, the AMO sequential counter encoding needs the largest number of variables, whereas it needs the second smallest number of clauses. The AMO bimander proposed here seems to be a good tradeoff between the number of variables and the number of required clauses.
Table 4.9: A summary of most well-known AMO SAT-encodings, where some encodings come from cardinality constraints noted by CAR.

<table>
<thead>
<tr>
<th>enc</th>
<th>clauses</th>
<th>aux vars</th>
<th>UPaAC</th>
<th>origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>pairwise</td>
<td>( \binom{n}{2} )</td>
<td>0</td>
<td>yes</td>
<td>folklore</td>
</tr>
<tr>
<td>linear (CAR.)</td>
<td>8n</td>
<td>2n</td>
<td>no</td>
<td>[War98]</td>
</tr>
<tr>
<td>totalizer</td>
<td>( O(n^2) )</td>
<td>( O(n \log(n)) )</td>
<td>yes</td>
<td>[BB03]</td>
</tr>
<tr>
<td>binary</td>
<td>( n \log_2 n )</td>
<td>( \left\lfloor \log_2 n \right\rfloor )</td>
<td>yes</td>
<td>[FPDN05]</td>
</tr>
<tr>
<td>sequential counter</td>
<td>( 3n - 4 )</td>
<td>( n - 1 )</td>
<td>yes</td>
<td>[Sin05]</td>
</tr>
<tr>
<td>sorting networks (CAR.)</td>
<td>( O(n \log_2(n)) )</td>
<td>( O(n \log_2^2(n)) )</td>
<td>yes</td>
<td>[ES06]</td>
</tr>
<tr>
<td>commander</td>
<td>( \sim 3n )</td>
<td>( \frac{n}{2} )</td>
<td>yes</td>
<td>[KK07]</td>
</tr>
<tr>
<td>product</td>
<td>( 2n + 4\sqrt{n} + O(\sqrt{n}) )</td>
<td>( 2\sqrt{n} + O(\sqrt{n}) )</td>
<td>yes</td>
<td>[Che10]</td>
</tr>
<tr>
<td>card. networks(CAR.)</td>
<td>( 6n - 9 )</td>
<td>( 4n - 6 )</td>
<td>yes</td>
<td>[ANORC11]</td>
</tr>
<tr>
<td>PHFs-based (CAR.)</td>
<td>( n \log_2 n )</td>
<td>( \left\lfloor \log_2 n \right\rfloor )</td>
<td>yes</td>
<td>[BHMM12]</td>
</tr>
<tr>
<td>bimander</td>
<td>( \frac{n^2}{2m} + n \log_2 m - \frac{n}{2} )</td>
<td>( \log_2 m, 1 \leq m \leq n )</td>
<td>yes</td>
<td>[HN13b, HN13a]</td>
</tr>
<tr>
<td>bimander ((m = \frac{n}{2}))</td>
<td>( n \log_2 n - \frac{n}{2} )</td>
<td>( \left\lfloor \log_2 n \right\rfloor - 1 )</td>
<td>yes</td>
<td>[HN13b, HN13a]</td>
</tr>
</tbody>
</table>

Figure 4.2: A comparison of AMO encodings on the Pigeon-Hole problem

Figure 4.3: A comparison of AMO encodings on the Langford problem
Figure 4.4: A comparison of AMO encodings on the all-interval series problem
4.3 Encoding Linear CSP Constraints

4.3.1 The Sparse and Order Encodings of Linear CSP Constraints

When transforming a CSP problem into a SAT instance, one typically adopts one of two common encodings, either the sparse encoding or the order encoding. While some efforts are taken to encode constraints as efficiently as possible [ACLM10], we are unaware of any work specifying clearly which encoding one should use when facing a particular CSP.

In this section, we address the case of SAT encodings for common CSP problems in which most of the constraints have the form $X \pm c \succ Y$, where $X$ and $Y$ are integer variables, $c$ is a constant, and $\succ$ is a relational operator. Not surprisingly, we confirm that the order encoding is better for problems with dominant inequality constraints ($\succ \in \{>, \geq, <, \leq \}$) but in problems where this is not the case the advantages of one or the other encoding are not obvious, and one is left with the problem of choosing the adequate encoding.

In the following section, we address this problem and not only discuss some features of the initial CSP problems that might be considered when selecting one or the other encodings, but also study combinations of both encodings.

In particular, we study the effectiveness of redundant modelling in SAT encodings, as done in constraint programming, where [CLW96] showed that redundant modelling can speed up constraint propagation. On the one hand, this approach is already incipient in the AMO sequential encoding [Sin05] that implements the at-most-one constraint required by the sparse encoding by means of extra variables similar to the order encoding variables (see Section 4.2.5). On the other hand, BEE is a SAT-based solver using the order encoding, which optimizes a CNF formula yielding considerable speed-ups in SAT solving time [MC12], uses extra variables, similar to those used in the sparse encoding to encode the all-different constraint [MC12]. We go a step further and discuss the more general conditions in which the combination of the sparse and order encodings pays off, i.e., when the overhead incurred by maintaining both encodings is compensated by the pruning achieved, resulting in better runtimes.

In CP, to improve the performance one can combine different representations of the same problem by bridging them via so-called channeling constraints [Wal01, Smi02, DLVC03]. Bordeaux, Hamadi and Zhang, who provided a comprehensive survey with a broad overview of the two areas CSP and SAT [BHZ06], stated that channeling constraints have not been studied in SAT (see page 10 in [BHZ06]). Here we adapt the use of channeling constraints in CP for the new SAT encodings.

We now focus on encoding finite binary constraints of the form $X \pm c \succ Y$, where $\succ$ is a relational operator in $\{=, \neq, >, \geq, <, \leq \}$. Without loss of generality and to simplify notation, we restrict $\pm$ to be $+$ and $c$ to be a positive integer.

- In the spirit of its semantics (a “negation”), a disequality constraint of the form
4.3. Encoding Linear CSP Constraints

$X + c \neq Y$ is modelled by a set of conflict clauses

$$\bigwedge_{i=1}^{n-c} \neg(X = i \land Y = i + c)$$

- encoded in the sparse encoding as:

$$\bigwedge_{i=1}^{n-c} (\neg d_i^X \lor \neg d_{i+c}^Y)$$

- and in the order encoding as

$$\bigwedge_{i=1}^{n-c} (\neg \sigma_{i-1}^X \lor \sigma_i^X \lor \neg \sigma_{i+c-1}^Y \lor \sigma_{i+c}^Y).$$

- In contrast, an equality constraint of the form $X + c = Y$ is modeled as:

$$\bigwedge_{i=1}^{n-c} (X = i \iff Y = c + i)$$

together with clauses to tighten the domain bounds, either on variable $X$ ($X \leq n - c$) or on variable $Y$ ($Y > c$). The constraint is thus encoded as

- in the sparse encoding (together with bounding clauses $\bigwedge_{i=1}^{n-c} \neg a_{n-c+i}^X$):

$$\bigwedge_{i=1}^{n-c} [(\neg d_i^X \lor d_{i+c}^Y) \land (d_i^X \lor \neg d_{i+c}^Y)]$$

- in the order encoding, and adopting the optimisation proposed in [ACLM10] that equates the order vectors:

$$\bigwedge_{i=1}^{n-c} [(\neg \sigma_i^X \lor \sigma_{i+c}^Y) \land (\sigma_i^X \lor \neg \sigma_{i+c}^Y)]$$

together with the bounding conditions $\neg \sigma_{n-c}^X$ and $\sigma_c^Y$.

We illustrate the representation of inequality CSP constraints with a constraint of the form $X + c \leq Y$ (the other operators $\{>, \geq, <, \}$ are similar).

- In the sparse encoding and assuming support clauses (conflict clauses would be similar), the range of $Y$ values compatible with $X = i$ are modelled by:

$$\bigwedge_{i=1}^{n-c} (X = i \rightarrow \bigvee_{j=c}^{n-i} Y = i + j) \iff \bigwedge_{i=1}^{n-c} (\neg d_i^X \lor \bigvee_{j=c}^{n-i} d_{i+j}^Y)$$

- For the order encoding the range of $Y$ values may be represented directly:

$$\bigwedge_{i=1}^{n-c} (X = i \rightarrow Y \geq i + c) \iff \bigwedge_{i=1}^{n-c} (\neg \sigma_{i-1}^X \lor a_i^X \lor \sigma_{i+c-1}^Y)$$

together with the bounding condition $\sigma_c^Y$. 
4.3.2 The $Sp$-$Or_{\text{red}}$ and $Sp$-$Or_{\text{hyb}}$ Encodings

It has been widely observed that SAT solvers can deal with shorter clauses much better than longer clauses. Analyzing the size of the clauses required by the different encodings one may conjecture that:

- the sparse encoding outperforms the order encoding in most problems in which equalities and disequalities are the dominating constraints, whereas

- the order encoding outperforms the sparse encoding in problems in which inequalities are the dominating constraints.

For the above conjecture, we have not considered the AMO constraints required by the sparse encoding. These may be naturally modelled by the pairwise encoding

$$\bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} (\neg d_i^V \lor \neg d_j^V)$$

which requires $\frac{n(n-1)}{2}$ binary clauses, but there are several efficient ways which only need $O(n)$ clauses [GN04, Sin05, FG10, HN13b], thus requiring less clauses but auxiliary variables. In all these proposals, besides encoding the AMO constraint, these auxiliary variables do not get involved in other clauses.

It is worth noting that the AMO sequential counter encoding [Sin05] uses as auxiliary variables exactly those that are used by the order encoding to encode finite CSPs into SAT (for the detail, see the proof of the their similarity in Section 4.2.7.1).

Hence, rather than simply selecting either the sparse or the order encoding to encode a specific CSP, we conjecture that combining them may produce better results than adopting one of them in isolation. To implement such a combination there must be additional bridging clauses between the two representations of every CSP variable $X$, namely the following set of clauses:

$$Bridge := \bigwedge_{i=1}^{n} (d_i^X \leftrightarrow \sigma_{i-1}^X \land \neg \sigma_i^X)$$

$$= \bigwedge_{i=1}^{n} (\neg d_i^X \lor \sigma_i^X$$

$$\land (\neg d_i^X \lor \neg \sigma_i^X)$$

$$\land (d_i^X \lor \neg \sigma_{i-1}^X \lor \sigma_i^X))$$

The $Bridge$ clauses above play exactly the role of the channeling constraints mentioned above. A new redundant encoding, so-called $Sp$-$Or_{\text{red}}$ encoding proposed in this thesis (see Table 4.10) represents a CSP composed of equality and inequality constraints by the SAT formula

$$\mathcal{F}_{Sp-Or_{\text{red}}} := Axiom \land Bridge \land Eqlt \land Ineq,$$

where

- $Axiom$ is a conjunction of clauses representing the vector of order variables $\sigma$;
4.3. Encoding Linear CSP Constraints

- \textit{Bridge} defined above are the clauses channeling the sparse and order variables \((d\) and \(o\), resp.); 

- \textit{Eqlt} are the clauses that represent all CSP equalities and disequalities expressed both on the sparse variables and order variables \((d\) and \(o\) resp.); 

- \textit{Ineq} are the clauses that represent all CSP inequalities expressed both on the sparse variables and order variables \((d\) and \(o\) resp.).

In summary, we expect the \(F_{Sp-Or_{red}}\) to take advantage of the following features:

- the variables required to encode the CSP are the same required by the sparse encoding using the sequential encoding to translate the AMO constraint; 

- there is no cost for ALO and AMO constraints in the sparse encoding; 

- the redundant representation of constraints will improve the pruning obtained with each encoding in isolation.

Alternatively, we will also consider a hybrid encoding \(Sp-Or_{hyb}\) that aims at taking advantage of the most adequate encodings for the different types of constraints. It represents a CSP by the SAT formula

\[
F_{Sp-Or_{hyb}} := Axion \land Bridge \land Eqlt_{sparse} \land Ineq_{order},
\]

where

- \(Eqlt_{sparse}\) are the clauses that represent all CSP equalities and disequalities, \textit{solely} on the sparse variables \(d\);

- \(Ineq_{order}\) are the clauses that represent all CSP inequalities, \textit{solely} on the order variables \(o\).

Table 4.10 presents an example of SAT encodings for CSP domain and constraints by using the sparse, order, and \(Sp-Or_{red}\) encodings. Given the use of the combined encoding, the corollary below follows.

**Corollary 4.3.1** Unit propagation on the \(Sp-Or_{red}\) and \(Sp-Or_{hyb}\) encodings maintain different types of consistency in the original CSP problem:

- \textit{due to the sparse encoding:} arc-consistency or forward-checking, respectively when the sparse encoding uses support clauses (the support encoding [Gen02]) or conflict clauses (the direct encoding [Wal00]),

- \textit{due to the order encoding:} bounds-consistency [BB03, TTKB09, MC12].

If the CSP includes other constraints, these may be modeled either with the sparse or the order variables, but these are out of the scope of this thesis (in all our experiments the problems only had equalities, disequalities and inequalities).

The main results of this chapter are also published in [HN13a, BHN13, HN13b, BHN14a].
Table 4.10: SAT encodings of a CSP variable and two constraints, illustrated on the example with a CSP variable $V$ having a domain value \{1, 2, ..., 9\}. \textit{Sparse(seq)} indicates the sparse encoding using the sequential encoding. \textit{Domain constraints} represents the SAT clauses that guarantee that each CSP variable is assigned to exactly one value of its domain when translating to SAT.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Sparse (seq)</th>
<th>Order</th>
<th>Sp-Orred</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vars</td>
<td>$d_i^V, 1 \leq i \leq 9$</td>
<td>$o_i^V, 0 \leq i \leq 9$</td>
<td>$d_i^V, 1 \leq i \leq 9$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$o_i^V = 1 \land o_i^V = 0$</td>
<td>$o_i^V, 0 \leq i \leq 9$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$o_i^V = 1 \land o_i^V = 0$</td>
<td>$o_i^V = 1 \land o_i^V = 0$</td>
</tr>
<tr>
<td>Aux. vars</td>
<td>$o_i, 1 \leq i \leq 9$</td>
<td>$o_9 = 1 \land o_2 = 0$</td>
<td></td>
</tr>
<tr>
<td>Constraint</td>
<td>$d_i^V \lor ... \lor d_9^V$</td>
<td>$d_i^V \lor ... \lor d_9^V$</td>
<td>$d_i^V \lor ... \lor d_9^V$</td>
</tr>
<tr>
<td></td>
<td>$\bigwedge_{i=1}^8 (o_i \lor \neg o_{i+1})$</td>
<td>$\bigwedge_{i=1}^8 (o_i^V \lor \neg o_{i+1}^V)$</td>
<td>$\bigwedge_{i=1}^8 (o_i^V \lor \neg o_{i+1}^V)$</td>
</tr>
<tr>
<td></td>
<td>$\bigwedge_{i=1}^9 (-d_i^V \lor \neg o_i) \land (-d_i^V \lor o_{i-1})$</td>
<td>$\bigwedge_{i=1}^9 (d_i^V \lor o_{i-1} \land \neg o_i^V)$</td>
<td>$\bigwedge_{i=1}^9 (d_i^V \lor o_{i-1} \land \neg o_i^V)$</td>
</tr>
<tr>
<td>$V = 5$</td>
<td>$d_5^V$</td>
<td>$o_5 \land \neg o_5$</td>
<td>$d_5^V$</td>
</tr>
<tr>
<td>$V \neq 5$</td>
<td>$\neg d_5^V$</td>
<td>$\neg o_5 \lor o_6$</td>
<td>$\neg d_5^V$</td>
</tr>
<tr>
<td>$V \geq 5$</td>
<td>$d_5^V \lor ... \lor d_9^V$</td>
<td>$o_5$</td>
<td>$d_5^V \lor ... \lor d_9^V$</td>
</tr>
</tbody>
</table>
CHAPTER 5

Experimental Results

The chapter is concerned with experiments that illustrate the encodings studied in Chapters 3 and 4.

The experiments whose results are reported in this section were performed on a Intel Core 2 Quad processor with 2.66 Ghz and 3.8 GB of memory, under Ubuntu 10.04. Running times reported are in seconds, unless indicated otherwise. The used solvers are Riss3C [Man13] (SAT competition 2013 version), Lingeling [Bie13] (SAT competition 2013 version) and Clasp [GKS09] (clasp2.1.3x86_64linux) with default configurations. These three state-of-the-art conflict-driven clause learning SAT solvers were ranked best on application and craft benchmarks in different categories at recent SAT competitions [SAT].

This part evaluates several aspects that influence the performance of SAT instances:

- the number of variables (and/or literals) required (search space)
- the number of generated clauses (overhead when propagating variable assignments)
- the running time (CPU time - reported in seconds)
- the number of decisions (propagation strength)
- the number of conflicts (propagation strength)
- the memory consumption (reported in MB)

It is worth noting that due to the large number of experiments we performed, we show only the results obtained in some of these results that are illustrative. For example, in many cases, we will only present the result produced by one solver instead of all three.
5.1 SAT Encodings of Finite CSPs

The *AMO sequential counter encoding* [Sin05] is used for the at-most-one constraint in the representative-sparse encodings and the sparse encoding, unless indicated otherwise.

Rep$S_p_1$, Rep$S_p_2$, Rep$S_p_{sqrt}$, and Rep$S_p_{n/2}$ (Rep$O_r_1$, Rep$O_r_2$, Rep$O_r_{sqrt}$, and Rep$O_r_{n/2}$) refer to the representative-sparse encodings (the representative-order encodings) with corresponding partitions: 1, 2, $\sqrt{n}$ and $n/2$. It is worth recalling that the special case of the representative encodings Rep$S_p_1$ is the sparse encoding, whereas Rep$O_r_1$ is the order encoding.

The experiments focus on the comparison of the best of the six representative encodings (i.e., with corresponding partitions 2, $\sqrt{n}$ and $n/2$) with the sparse encoding (Rep$S_p_1$) and the order encoding (Rep$O_r_1$). The two latter are well-known and efficient SAT encodings of CSPs. To produce the informative figures, log scales are sometimes used on the axes as necessary.

**The Pigeon-Hole Problem**  Figures 5.1, 5.2 and 5.3 show the results on unsatisfiable Pigeon-Hole instances for different numbers of pigeons (parameter $\#pigeons$ in the figure) produced by Clasp, Lingeling, and Riss3G, respectively. As can be seen, the sparse encoding (Rep$S_p_1$) is the worst encoding and it is significantly worse than the second worst encoding (slower than one order of magnitude).

![Figure 5.1: Running time of various encodings produced by Clasp on unsatisfiable Pigeon-Hole instances. Running times are reported in seconds.](image)

Furthermore, the order encoding (Rep$O_r_1$) performs poorly with Clasp, the second worst encoding (see Fig. 5.1), as it is consistently and marginally faster with Lingeling (Fig. 5.2) and Riss3G (Fig. 5.3). The representative-order encodings, which perhaps benefit from the order encoding, outperform the representative-sparse encodings significantly. In particular, the two best representative-order encodings Rep$O_r_2$ and Rep$O_r_{sqrt}$ are the overall best encodings.
5.1. SAT Encodings of Finite CSPs

Figure 5.2: Running time of various encodings produced by Lingeling on unsatisfiable Pigeon-Hole instances. Running times are reported in seconds.

Figure 5.3: Running time of various encodings produced by Riss3G on unsatisfiable Pigeon-Hole instances. Running times are reported in seconds.

Figure 5.4: A comparison of the number of conflicts produced by Clasp on unsatisfiable Pigeon-Hole instances.
Figures 5.4, 5.5 and 5.6 present the numbers of conflicts in unsatisfiable Pigeon-Hole instances produced by Clasp, Lingeling, and Riss3G, respectively. These numbers of conflicts explain the reason for the performance of each encoding. The encodings that require less conflicts have a faster performance. Consequently, running time is very closely related to the number of conflicts, as well as to the number of decisions.

Figure 5.5: A comparison of the number of conflicts produced by Lingeling on unsatisfiable Pigeon-Hole instances.

Figure 5.6: A comparison of the number of conflicts produced by Riss3G on unsatisfiable Pigeon-Hole instances.

Throughout the figures, one can observe a strong correlation between:

- the running time and the number of decisions produced by Clasp (Fig. 5.7);
- the running time and the number of conflicts produced by Lingeling (Fig. 5.8);
- the number of conflicts and the number of decisions produced by Riss3G (Fig. 5.9).
5.1. SAT Encodings of Finite CSPs

Figure 5.7: The correlation of the running time and the number of decisions produced by Clasp on unsatisfiable Pigeon-Hole instances.

Figure 5.8: The correlation of the running time and the number of conflicts produced by Lingeling on unsatisfiable Pigeon-Hole instances.

Figure 5.9: The correlation of the number of conflicts and the number of decisions produced by Riss3G on unsatisfiable Pigeon-Hole instances.
The speedups show a large variation, depending on the solver, but the most relevant finding is that, regardless of different performance, all solvers (with the exception for Lingeling with the order encoding - RepOr₁) exhibit a speedup of one to two orders of magnitude, when the problem is encoded with the representative encodings. In general, a smaller number of variables might allow solvers to run faster whereas longer clauses possibly makes solvers slower. The main reason for this speedup is that compared to other encodings a much smaller number of variables required by the representative encodings can compensate the longer clauses to encode disequality constraints (see Table 3.3).

Interestingly, some representative encodings are more memory efficient than the sparse and order encodings. Fig. 5.10 and Fig. 5.11 show that the sparse encoding consumes significantly more memory than other encodings, whereas the order encoding consumes more memory than the two encodings RepOr₂ and RepOrₜ₉₄₉ for Lingeling.

![Figure 5.10](image1.png)

**Figure 5.10**: A comparison of memory in MB consumed by Lingeling on unsatisfiable Pigeon-Hole instances.

![Figure 5.11](image2.png)

**Figure 5.11**: A comparison of memory in MB consumed by Riss3G on unsatisfiable Pigeon-Hole instances.
5.1. SAT Encodings of Finite CSPs

The main reason for the lower memory consumption is the number of SAT variables required to encode the problems, that is much smaller with the representative encodings.

Regardless of the variability of run times in different SAT solvers on unsatisfiable Pigeon-Hole instances, the experiment indicates that:

- The representative-sparse encodings (i.e., with corresponding partitions $2$, $\sqrt{n}$ and $n/2$) not only require significantly less running time, but also consume remarkably lower memory than the sparse encoding.

- The order encoding is quite competitive compared to the representative-order encodings. However it usually runs slower and consumes more memory than either $RepO_{r2}$ and $RepO_{sqrt}$.

The Graph Colouring Problem  We used various benchmarks from [Tri] with 245 instances. Most of these instances are unsatisfiable. Figures 5.12, 5.13 and 5.14 summarize the results on graph colouring instances performed with Clasp, Lingeling, and Riss3G, respectively.

The simpler disequality constraints between node colors in CSP constraints and the unsatisfiable nature of the tested instances make this problem to exhibit similar features as the pigeon hole problem.

The effect of the representative encodings is much stronger in the sparse case, where in both solvers speedups with one or two orders of magnitude are achieved with the representative-sparse encodings, typically preferred $\sqrt{n}$ number of partitions. Furthermore, Fig. 5.12 shows that $RepS_{sqrt}$ is quite competitive with the representative-order encodings. This is probably due to the trade off between level one and level two of $RepS_{sqrt}$.

![Figure 5.12: Running time of various encodings produced by Clasp on graph colouring instances. Running times are reported in seconds.](image-url)
Chapter 5. Experimental Results

Figure 5.13: Running time of various encodings produced by Lingeling on graph colouring instances. Running times are reported in seconds.

The representative-order encodings RepOr\textsubscript{1}, RepOr\textsubscript{2}, and RepOr\textsubscript{sqrt} have a similar performance. The order encoding performs worse than RepOr\textsubscript{2} and RepOr\textsubscript{sqrt} within 2,000 seconds for all three SAT solvers. On the other hand, given a longer timeout (between 2,000 and 5,000 seconds) the first encoding runs slightly faster than the last two encodings in Clasp and Lingeling (see Fig. 5.12 and Fig. 5.13). Interestingly, the RepOr\textsubscript{2} consistently outperforms the order encoding (see Fig. 5.14).

Like in the pigeon hole problem, the order encoding performs better than the sparse encoding, and in the former the representative encodings have similar running times, with some marginal speedup.

Figure 5.14: Running time of various encodings produced by Riss3G on graph colouring instances. Running times are reported in seconds.

Figures 5.15, 5.16 and 5.17 compare the number of conflicts between the order encoding and RepOr\textsubscript{2}, the best of the representative-order encodings on graph colouring instances. As shown in these figures, in all three solvers RepOr\textsubscript{2} requires a smaller number of conflicts compared with RepOr\textsubscript{1} in Lingeling and Riss3G. However, some instances performed by RepOr\textsubscript{2} need a remarkably larger number of
conflicts than by $RepOr_1$ (see Fig. 5.16 and Fig. 5.17). This fact is mostly due to the longer clauses generated by $RepOr_2$, which result in a bad performance for Lingeling and Riss3G.

Figure 5.15: A comparison of the number of conflicts between $RepOr_1$ and $RepOr_2$ produced by Clasp on graph colouring instances.

Figure 5.16: A comparison of the number of conflicts between $RepOr_1$ and $RepOr_2$ produced by Lingeling on graph colouring instances.

Figure 5.17: A comparison of the number of conflicts between $RepOr_1$ and $RepOr_2$ produced by Riss3G on graph colouring instances.
Figures 5.18, 5.20 and 5.22 compare the number of conflicts between $RepSp_1$ and $RepSp_{\text{sqrt}}$ on graph colouring instances.

Figure 5.18: A comparison of the number of conflicts between $RepSp_1$ and $RepSp_{\text{sqrt}}$ produced by Clasp on graph colouring instances.

Figure 5.19: A comparison of the number of decisions between $RepSp_1$ and $RepSp_{\text{sqrt}}$ produced by Clasp on graph colouring instances.

Figure 5.20: A comparison of the number of conflicts between $RepSp_1$ and $RepSp_{\text{sqrt}}$ produced by Lingeling on graph colouring instances.
Figures 5.19, 5.21 and 5.23 compare the number of decisions between \( \text{RepSp}_1 \) and \( \text{RepSp}_{\text{sqrt}} \) on graph colouring instances.

Figure 5.21: A comparison of the number of decisions between \( \text{RepSp}_1 \) and \( \text{RepSp}_{\text{sqrt}} \) produced by Lingeling on graph colouring instances.

Figure 5.22: A comparison of the number of conflicts between \( \text{RepSp}_1 \) and \( \text{RepSp}_{\text{sqrt}} \) produced by Riss3G on graph colouring instances.

Figure 5.23: A comparison of the number of decisions between \( \text{RepSp}_1 \) and \( \text{RepSp}_{\text{sqrt}} \) produced by Riss3G on graph colouring instances.
In general, from these figures an unsurprising observation is that there is a strong connection between the number of conflict that are found during the search and the number of decisions, which correspond to the number of nodes that

As shown in the figures, in most instances RepSp\textsubscript{1} requires remarkably more conflicts and decisions compared to RepSp\textsubscript{sqrt}. In Fig. 5.22, many instances encoded by RepSp\textsubscript{1} cannot be solved within the allowed running time of 5000 seconds. This comes to no surprise since the search space generated by RepSp\textsubscript{1} is larger, while the propagation strength is not stronger than in RepSp\textsubscript{sqrt}.

Fig. 5.24 and Fig. 5.25 compare the memory consumption for solving RepOr\textsubscript{1} and RepOr\textsubscript{2}, whereas Fig. 5.26 and Fig. 5.27 compare RepSp\textsubscript{1} and RepSp\textsubscript{sqrt}.

As can be seen, a number of instances generated by RepOr\textsubscript{1} and RepOr\textsubscript{2} require similar memory performed by Lingeling (Fig. 5.24), while many instances generated by RepOr\textsubscript{1} require more memory than RepOr\textsubscript{2} when using Riss3G (see Fig. 5.25).

![Figure 5.24](image1)

Figure 5.24: A comparison on consumed memory between RepOr\textsubscript{1} and RepOr\textsubscript{2} produced by Lingeling on graph colouring instances, reported in MB.

![Figure 5.25](image2)

Figure 5.25: A comparison on consumed memory between RepOr\textsubscript{1} and RepOr\textsubscript{2} produced by Riss3G on graph colouring instances, reported in MB.

In contrast to a slight difference between RepOr\textsubscript{1} and RepOr\textsubscript{2}, the memory consumption for RepSp\textsubscript{1} and RepSp\textsubscript{sqrt} is significantly different. Most instances generated by RepSp\textsubscript{1} require more memory and a few instances require less memory.
5.1. SAT Encodings of Finite CSPs

than $RepSp_{sqrt}$ for both Lingeling and Riss3G. The lower number of variables mostly pays-off in these cases.

Figure 5.26: A comparison on consumed memory between $RepSp_1$ and $RepSp_{sqrt}$ produced by Lingeling on graph colouring instances, reported in MB.

Figure 5.27: A comparison on consumed memory between $RepSp_1$ and $RepSp_{sqrt}$ produced by Riss3G on graph colouring instances, reported in MB.

An interesting observation is that the encodings with $\sqrt{n}$ number of partitions have a good performance. In particular, $RepSp_{sqrt}$ significantly outperforms the sparse encoding, whereas $RepOr_{sqrt}$ is usually competitive with the order encoding. Clearly, when the number of SAT variables in level one and level two are balanced ($\sqrt{n}$) the lower number of variables does pay-off in this type of problems.

The Open Shop Scheduling Problem We used various benchmarks from [Tai] with 90 instances. In particular, for each benchmark, we choose two consecutive instances: one is unsatisfiable and the other is satisfiable with an optimal solution. All CSP constraints of this problem are inequalities.
Figure 5.28: Running time of various encodings produced by Clasp on open shop scheduling instances. Running times are reported in seconds.

Figure 5.29: Running time of various encodings produced by Lingeling on open shop scheduling instances. Running times are reported in seconds.

Fig. 5.28, Fig. 5.29 and Fig. 5.30 evaluate the running time of various encodings on open shop scheduling instances produced by Clasp, Lingeling, and Riss3G, respectively. As expected, in problems where CSP inequality constraints dominate, the order encoding (RepOr$_1$) is much faster than the sparse encoding (RepSp$_1$). The former is particularly adequate to propagate changes in the bounds, and no speedups were expected with representative encodings (and in fact a slow-down is observed). Nevertheless, one of the representative-order encoding RepOr$_2$ has a slightly worse performance with Clasp and Lingeling (see Fig. 5.28, Fig. 5.29) compared with RepOr$_1$, as it is very competitive with RepOr$_1$ when Riss3G is used (see Fig. 5.30).
5.1. SAT Encodings of Finite CSPs

Figure 5.30: Running time of various encodings produced by Riss3G on open shop scheduling instances. Running times are reported in seconds.

Figure 5.31: A comparison of the number of conflicts between RepOr₁ and RepOr₂ produced by Lingeling on open shop scheduling instances.

Figure 5.32: A comparison of the number of conflicts between RepOr₁ and RepOr₂ produced by Riss3G on open shop scheduling instances.
In all three solvers, the \textit{representative-sparse} encodings perform very similarly to the \textit{representative-order} encodings. This effect might be due to the fact that the partition of the large size domains (size 192 to 416) efficiently propagates the changes in the bounds, for which the sparse encoding is not suited. Overall, the representative encodings perform worse than the order encoding, but significantly better than the sparse encoding, especially when the numbers of SAT variables in levels one and two are balanced ($\text{RepSp}_{\sqrt{n}}$).

A comparison of the number of conflicts is shown in Fig. 5.31 and Fig. 5.32. The order encoding ($\text{RepOr}_1$) has a smaller number of conflicts than $\text{RepOr}_2$. This comes to no surprise given that the order encoding has a trade off between the large size domains (size 192 to 416) and the propagation strength in the bounds, whereas $\text{RepOr}_2$ generates longer clauses, which cannot be dealt with by SAT solvers.

We provide conclusions and future work for this section in Section 7.2.
5.2 SAT Encodings of the At-Most-One Constraint

This section provides an empirical study of the AMO encodings (see Section 4.2). For the experimental evaluation we have selected some well-known problems, which have been used in recent CSP and SAT competitions. Note that we show the memory used only by Lingeling and Riss3G since Clasp does not provide this information.

The *AMO bimander, binary, commander, product, pairwise, and sequential counter* encoding are abbreviated as *bim, bin, cmd, pro, pw, and seq*, respectively. For some AMO encodings, the set of *n* variables is divided into *m* disjoint subsets. Number *m* is chosen in such a way that the corresponding AMO encoding conducted on all benchmarks should give the best result in terms of the average running time:

- the AMO *product* encoding: *m* = 2;
- the AMO *bimander* encoding: *m* = √*n*;
- the AMO *commander* encoding: *m* = 2.

**The Pigeon-Hole Problem**

![Graphs showing running time and correlation](image)

Figure 5.33: A comparison of AMO encodings for Clasp on unsatisfiable Pigeon-Hole instances.

For Clasp, as shown in Fig. 5.33, the *AMO binary* encoding is the clear winner, and the *AMO pairwise* encoding is clearly the worst encoding in terms of running
time for unsatisfiable Pigeon-Hole instances. The AMO sequential counter encoding performs poorly, whereas the AMO bimander, commander, product encoding perform equally well.

![Graphs showing running time and correlation of running time and #conflicts](image)

**Figure 5.34**: A comparison of AMO encodings for Lingeling on unsatisfiable Pigeon-Hole instances.

For Lingeling, Fig. 5.34 summarizes the results which compare several aspects of different AMO encodings on unsatisfiable Pigeon-Hole instances. As for Clasp, the AMO pairwise encoding is clearly the worst encoding for Lingeling in terms of running time. Unlike for Clasp, the AMO product encoding is the best encoding, followed by the AMO commander encoding. The others are quite similar.

Fig. 5.35 evaluates different AMO encodings with Riss3G on unsatisfiable Pigeon-Hole instances. As for Lingeling and Clasp, the AMO pairwise encoding is the slowest encoding, followed by the AMO binary encoding. The remaining AMO encodings show no clear pattern.
5.2. SAT Encodings of the At-Most-One Constraint

There are some other interesting observations. First, the running time of different AMO encodings are quite diverse. For example, the AMO binary is a winner with Clasp (see Fig. 5.33), but it is the second worst encoding with Lingeling (Fig. 5.34) and Riss3G (Fig. 5.35). However, the AMO pairwise encoding is the worst encoding for all three solvers. Second, the running time is very closely related to the number of conflicts, as well as to the number of decisions, so one explains the others, specially in Fig. 5.33. Third, faster encodings seem to consume less memory (see Fig. 5.34 and Fig. 5.35).

The Hidoku Problem

As shown in Fig. 5.36 for Clasp, the AMO pairwise encoding performs well except for the last instance, whereas the AMO commander encoding is clearly a poor encoding. The AMO binary encoding gives rather good results, while the remaining encodings roughly show a similar performance.

With satisfiable Hidoku, several AMO encodings are solved with Lingeling and results are presented in Fig. 5.37. The AMO bimander and pairwise encoding are the best encodings. The AMO sequential counter encoding is the worst encoding, whereas the other encodings are similar.
Figure 5.36: A comparison of AMO encodings for Clasp on satisfiable Hidoku instances.

Figure 5.37: A comparison of AMO encodings for Lingeling on satisfiable Hidoku instances.
5.2. SAT Encodings of the At-Most-One Constraint

![Graphs showing running time, correlation between running time and conflicts, correlation between conflicts and decisions, and memory usage.]

Figure 5.38: A comparison of AMO encodings for Riss3G on satisfiable Hidoku instances.

Fig. 5.38 summarizes the results of different AMO encodings for Riss3G on satisfiable Hidoku instances. The AMO pairwise encoding performs remarkably well for Lingeling but it runs very slow for Riss3G, especially on the last two instances. The five other encodings show no clear pattern.

Regardless of six AMO encodings produced by three SAT solvers, several observations can be drawn on satisfiable Hidoku instances:

1. The running time of different AMO encodings are very diverse, even more than for the Pigeon-Hole problem.

2. There is a slight correlation of: the running time, the number of conflicts, and the number of decisions.

3. The running time is somehow related to the consumed memory. Particularly, the faster encoding tends to consume a smaller amount of memory.
The Golomb Ruler Problem

Figures 5.39, 5.40, and 5.41 compare different AMO encodings on satisfiable Golomb ruler instances for Clasp, Lingeling, and Riss3G, respectively.

In terms of running times the AMO sequential counter and bimander encoding perform very well with the three solvers. The AMO binary and product encoding are quite good, whereas the AMO pairwise performs very poor. The AMO commander encoding performs unpredictably since it is the worst encoding for Clasp and Lingeling, whereas it is the best encoding for Riss3G.

![Figure 5.39: A comparison of AMO encodings for Clasp on satisfiable Golomb ruler instances.](image)

Regardless of the different solvers, another interesting observation is that the running time is slightly related to the number of conflicts. The number of conflicts is closely related to the number of decisions. In terms of the consumed memory, one can say that a faster encoding tends to consume less memory than a slower encoding. However, this statement does not always happen: In Fig. 5.40 the AMO sequential counter encoding performs very well, especially on the largest instance, but it requires much memory (e.g., instances 9, 10, and 12).
5.2. SAT Encodings of the At-Most-One Constraint

Figure 5.40: A comparison of AMO encodings for Lingeling on satisfiable Golomb ruler instances.

Figure 5.41: A comparison of AMO encodings for Riss3G on satisfiable Golomb ruler instances.
The Open Shop Scheduling Problem

Figures 5.42, 5.43, and 5.44 present the results of the AMO encodings for Clasp, Lingeling, and Riss3G on the open shop scheduling problem, respectively.

As can be observed in Fig. 5.42 by Clasp, the AMO \textit{pairwise} encoding performs worse. The AMO \textit{bimander} encoding performs quite well with small instances, but it runs slowly with large instances. The AMO \textit{product} encoding and the AMO \textit{binary} encoding seem to be the best encodings in terms of running time. These two encodings require a smaller number of conflicts compared to the others. The other AMO encodings show no clear pattern.

![Comparison of AMO encodings for Clasp on open shop instances](image)

Figure 5.42: A comparison of AMO encodings for Clasp on open shop instances.

Unlike for the previous problems, the running time has a weak relation to the number of conflicts. On the other hand, the number of decisions has a slight relation to the number of conflicts, especially when using Clasp. Interestingly, there is a strong correlation between the running time and the consumed memory (see Fig. 5.43 and Fig. 5.44).
5.2. SAT Encodings of the At-Most-One Constraint

Figure 5.43: A comparison of AMO encodings for Lingeling on open shop instances.

Figure 5.44: A comparison of AMO encodings for Riss3G on open shop instances.
The All-Interval Series Problem

Since Lingeling and Riss3G do not offer a configuration for finding all solutions for a CNF instance, the all-interval series problem is only evaluated with Clasp.

Fig. 5.45 presents the result of different AMO encodings for finding all solutions on all-interval series instances.

Figure 5.45: A comparison of AMO encodings for Clasp on all-interval series instances

The figure shows that the AMO pairwise encoding is clearly better than the others on the last two instances. On the contrary, the AMO sequential counter encoding performs badly on the last two instances. The AMO product and binary encodings are quite poor, while the AMO commander and bimander encoding are rather good.

Like for the Golomb ruler problem, the running time is rather related to the number of conflicts, whereas the number of conflicts is strongly related to the number of decisions.
5.2. SAT Encodings of the At-Most-One Constraint

The Langford Problem

Since Lingeling and Riss3G do not offer a configuration for finding all solutions for a CNF instance, the langford problem is only evaluated with Clasp. A comparison of different AMO encodings for finding all solutions on langford instances is shown in Fig. 5.46.

It can be seen that the AMO pairwise encoding is worse than the others, while the AMO sequential counter, product, and commander encoding perform quite similarly, followed by the AMO binary and bimander encoding.

Interestingly, the figure at the top-right corner of Fig. 5.46 demonstrates that the running time (at the log scale) is extremely strongly related to the order of the Langford problem. In other words, we may evaluate the running time via the order of the Langford problem.

![Figure 5.46: A comparison of AMO encodings for Clasp on all-interval series instances](image)

Furthermore, Fig. 5.46 also reveals that the running time, the number of conflicts, and the number of decisions have a strong correlation.
A Brief Conclusion of the Bimander Encoding  The new encoding, namely the AMO bimander encoding which we proposed, is very competitive with others. Particularly, with only one exceptions on the open shop problem, the AMO bimander encoding is one of the best encodings. It is the second best on many problems, for example, on the pigeon-hole, all-interval series, Golomb, Hidoku problems.

We will give detailed conclusions and future work for the AMO encodings in Section 7.3.
5.3 SAT Encodings of Linear CSP Constraints

We now focus on encoding finite binary constraints of the form $X \pm c \triangleright Y$, where $X$ and $Y$ are variables, and $\triangleright$ is a relational operator in \{=$, \neq, >, \geq, <, \leq\}. With no loss of generality and to simplify notation, we restrict $\pm$ to be $+$ and $c$ to be a positive integer.

To understand the importance of the encodings in different conditions, several experiments were conducted on various CSP problems that present a different mix of equality ($\triangleright$ is \{\}\), disequality ($\triangleright$ is \{\neq\}) and inequality ($\triangleright$ is a relational operator in \{\{>, \geq, <, \leq\}\}) constraints.

In the tables presented below, the columns $Spa$ and $Ord$ refer to the sparse and order encodings, respectively. Columns $Sp-Or_{red}$ and $Sp-Or_{hbg}$ refer to their encodings, respectively. All running times are reported in seconds.

The Open Shop Scheduling Problem
Tamura et al. solved many benchmarks successfully with the order encoding [TTKB09]. The benchmarks were generated by the Taillard’s method [Tal93]. Column Instance identifies the instances of the problem, where $a_b$ refers to the $b^{th}$ instance of the benchmark with $a$ jobs and $a$ machines. For the AMO constraints required by the sparse encoding we used the bimander encoding introduced in [HN13b].

Table 5.1: The running time comparison of various encodings for Clasp and Lingeling on open shop scheduling instances. $M$ is used as the makespan. $S/U$ indicates the satisfiability or unsatisfiability of instances.

| Instance | $M$ | $S$ | Clasp | |          | Lingeling |
|----------|-----|-----|-------|----------|-----------|
|          |     |     | $Spa$ | $Ord$ | $Sp-Or_{red}$ | $Spa$ | $Ord$ | $Sp-Or_{red}$ |
| 41       | 249 | U   | 42.80 | 0.13  | 0.33 | 23.03 | 0.25  | 0.46  |
|          | 250 | S   | 44.13 | 0.14  | 0.47 | 27.11 | 0.21  | 0.67  |
| 45       | 294 | U   | 77.12 | 0.12  | 1.62 | 41.87 | 0.34  | 1.01  |
|          | 295 | S   | 78.51 | 0.14  | 1.55 | 42.16 | 0.28  | 1.27  |
| 51       | 299 | U   | 104.26| 0.62  | 2.38 | 192.11| 1.07  | 6.73  |
|          | 300 | S   | 101.72| 0.57  | 4.02 | 185.54| 0.40  | 2.49  |
| 52       | 261 | U   | 130.90| 0.83  | 2.15 | 132.14| 1.02  | 5.40  |
|          | 262 | S   | 129.55| 0.71  | 4.32 | 106.02| 0.61  | 2.91  |

These instances of the open shop scheduling problem show the order encoding to be more efficient than either the sparse or any of the redundant $Sp-Or_{red}$ encoding. As mentioned in Section 4.3, in CSP problems where inequality constraints are dominant, the order encoding significantly outperforms the sparse encoding, but less so regarding the $Sp-Or_{red}$ encoding.
The Two-Dimensional Strip Packing Problem

All the CSP constraints in this problem are inequalities, and the benchmarks by Hopper and Turton [HT01] are used to evaluate SAT encodings. There are five categories, each consisting of three or four instances. The bold instances of the column Height indicate the optimal strip height. In the column Instance, Catb refers to the bth instance of the category a in Hopper and Turton [HT01]. Note that in each benchmark, the width (not shown) is fixed. The best results for the sparse encoding, which are reported in the column Instance, were obtained with the AMO sequential counter encoding [Sin05].

Table 5.2: The running time comparison of SAT encodings for Clasp on two-dimensional strip packing instances. H is the height of used rectangles. S/U indicates the satisfiability or unsatisfiability of instances. Running times reported are in seconds. The dashes mean running times larger than 3600 seconds.

<table>
<thead>
<tr>
<th>Instance</th>
<th>H</th>
<th>S/U</th>
<th>Spa</th>
<th>Ord</th>
<th>Sp-Or</th>
<th>Instance</th>
<th>H</th>
<th>S/U</th>
<th>Spa</th>
<th>Ord</th>
<th>Sp-Or</th>
</tr>
</thead>
<tbody>
<tr>
<td>C101</td>
<td>19</td>
<td>U</td>
<td>2.2</td>
<td>0.4</td>
<td>0.6</td>
<td>C401</td>
<td>59</td>
<td>U</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>S</td>
<td>0.5</td>
<td>0.0</td>
<td>0.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>60</td>
</tr>
<tr>
<td>C102</td>
<td>19</td>
<td>U</td>
<td>19.3</td>
<td>1.1</td>
<td>1.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>S</td>
<td>-</td>
<td>0.1</td>
<td>2.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>59</td>
</tr>
<tr>
<td>C103</td>
<td>19</td>
<td>U</td>
<td>3.7</td>
<td>0.7</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>S</td>
<td>3.0</td>
<td>0.0</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>61</td>
</tr>
<tr>
<td>C201</td>
<td>14</td>
<td>U</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C403</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>S</td>
<td>-</td>
<td>37.2</td>
<td>100.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>60</td>
</tr>
<tr>
<td>C202</td>
<td>14</td>
<td>U</td>
<td>-</td>
<td>1,459.0</td>
<td>2,763.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>S</td>
<td>-</td>
<td>132.6</td>
<td>764.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>89</td>
</tr>
<tr>
<td>C203</td>
<td>14</td>
<td>U</td>
<td>-</td>
<td>2,967.4</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>S</td>
<td>284.1</td>
<td>0.7</td>
<td>1.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>91</td>
</tr>
<tr>
<td>C301</td>
<td>29</td>
<td>U</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>S</td>
<td>-</td>
<td>104.9</td>
<td>255.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>89</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>S</td>
<td>400.1</td>
<td>0.2</td>
<td>12.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>90</td>
</tr>
<tr>
<td>C302</td>
<td>29</td>
<td>U</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>S</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>S</td>
<td>17.1</td>
<td>0.4</td>
<td>2.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>89</td>
</tr>
<tr>
<td>C303</td>
<td>29</td>
<td>U</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>S</td>
<td>-</td>
<td>485.0</td>
<td>782.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>S</td>
<td>49.2</td>
<td>0.2</td>
<td>1.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>92</td>
</tr>
</tbody>
</table>

Table 5.2 summarizes the results from three encodings on two-dimensional strip packing instances. As can be seen from the table, the order encoding remarkably outperforms the sparse encoding on all instances.
5.3. SAT Encodings of Linear CSP Constraints

We also show the results obtained with the $Sp$-$Or$ encoding, where inequalities are redundantly modeled with $d$ variables. It is clear that this redundancy does not pay off: Any extra pruning obtained with this redundant representation does not compensate the overhead of maintaining it.

The Quasigroup With Holes Problem (QWH)

QWH instances can be considered as a multiple permutation problem in which the variables may occur in more than one permutation, which requires the alldifferent constraint. In a permutation the dominating constrains are disequalities of type $X \neq Y$, where $X$ and $Y$ are variables. We experimented with QWH instances of different levels of hardness, using the bimander encoding, which is an encoding of at-most-one CSP constraint into SAT introduced in [HN13b], for the sparse encoding.

Table 5.3: The running time comparison of encodings performed by Clasp and Lingeling on satisfiable QWH instances.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Clasp</th>
<th>Lingeling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$Sp$</td>
<td>$Or$</td>
</tr>
<tr>
<td>order10.holes100</td>
<td>0.0</td>
<td>0.1</td>
</tr>
<tr>
<td>order18.holes120</td>
<td>0.0</td>
<td>0.1</td>
</tr>
<tr>
<td>order20.holes400</td>
<td>0.0</td>
<td>0.1</td>
</tr>
<tr>
<td>order30.holes320</td>
<td>0.2</td>
<td>0.7</td>
</tr>
<tr>
<td>order30.holes316</td>
<td>0.2</td>
<td>1.7</td>
</tr>
<tr>
<td>order30.holes900</td>
<td>0.7</td>
<td>6.1</td>
</tr>
<tr>
<td>order33.holes381</td>
<td>79.7</td>
<td>&gt;7200</td>
</tr>
<tr>
<td>order35.holes405</td>
<td>3.5</td>
<td>858.7</td>
</tr>
<tr>
<td>order40.holes528</td>
<td>103.9</td>
<td>&gt;7200</td>
</tr>
<tr>
<td>order40.holes544</td>
<td>100.8</td>
<td>&gt;7200</td>
</tr>
<tr>
<td>order40.holes560</td>
<td>21.2</td>
<td>3,757.9</td>
</tr>
<tr>
<td>order40.holes1600</td>
<td>9.9</td>
<td>2,317.1</td>
</tr>
</tbody>
</table>

As we can see, Table 5.3 presents the running time for finding the first solution of satisfiable QWHPs instances of different sizes. The table shows that the sparse encoding is far better than the order encoding, as would be expected since the order encoding was proposed for interval variables, subject to CSP inequality constraints that do not exist in this case. Generally, the $Sp$-$Or_{red}$ encoding produces similar results to the sparse encoding. Specifically, the redundancy results in a more significant slow down of the run time for Clasp, whereas its performance is clearly better than the sparse encoding for Lingeling.
The Hamiltonian Cycle Problem

Like for the QWH problem, the dominating CSP constrains in this benchmark are disequalities $X \neq Y$. Most of them are structured as *alldifferent* constraints. We experimented with instances from [Tri], using the bimander encoding [HN13a] for the sparse encoding.

Table 5.4: The running time comparison of encodings for *Clasp* on Hamiltonian cycle instances. Running times reported are in seconds.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Spa</th>
<th>Ord</th>
<th>$Sp$–$Or_{red}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>anna</td>
<td>1.2</td>
<td>&gt;10,000.0</td>
<td>5.8</td>
</tr>
<tr>
<td>david</td>
<td>96.6</td>
<td>&gt;10,000.0</td>
<td>7,195.0</td>
</tr>
<tr>
<td>jean</td>
<td>0.2</td>
<td>&gt;10,000.0</td>
<td>1.2</td>
</tr>
<tr>
<td>huck</td>
<td>1,887.1</td>
<td>&gt;10,000.0</td>
<td>9,341.8</td>
</tr>
<tr>
<td>miles750</td>
<td>22.9</td>
<td>&gt;10,000.0</td>
<td>56.4</td>
</tr>
<tr>
<td>miles1000</td>
<td>8.2</td>
<td>&gt;10,000.0</td>
<td>8.1</td>
</tr>
<tr>
<td>miles1500</td>
<td>3.1</td>
<td>&gt;10,000.0</td>
<td>3.8</td>
</tr>
<tr>
<td>queen14_14</td>
<td>2.2</td>
<td>11.7</td>
<td>18.4</td>
</tr>
<tr>
<td>queen15_15</td>
<td>3.3</td>
<td>18.6</td>
<td>28.3</td>
</tr>
<tr>
<td>queen16_16</td>
<td>5.1</td>
<td>27.5</td>
<td>43.1</td>
</tr>
<tr>
<td>myciel5</td>
<td>10.1</td>
<td>&gt;10,000.0</td>
<td>31.0</td>
</tr>
<tr>
<td>myciel6</td>
<td>4,575.8</td>
<td>&gt;10,000.0</td>
<td>&gt;10,000.0</td>
</tr>
</tbody>
</table>

The results obtained with the Hamiltonian cycle problem are shown in Table 5.4. They show a similar pattern than the previous problem: the *sparse encoding* is much better than the *order encoding*, and the $Sp$–$Or_{red}$ encoding is slightly worse than the *sparse encoding*. In two instances the difference between the $Sp$–$Or_{red}$ and the *sparse encoding* in run time exceeds the order of magnitude.

The Graph Colouring Problem

We experimented with widely-used hard unsatisfiable instances, obtained from different generators [colb]. For the first six benchmarks we got optimal solutions (the instance with lowest $K$ is unsatisfiable and the other is satisfiable), and the remaining benchmarks are unsatisfiable. The obtained results are shown in Table 5.5, adopting the AMO sequential counter encoding [Sin05] for the AMO constraints.

Although there are no inequality constraints in these problems, the order encoding is better than the sparse encoding. There are two possible reasons for this “unexpected” behavior. A possible reason is the smaller size of the domains: vertices may have about 10 colors, whereas in the previous problems the domains were larger. Another possible explanation is the structure of the disequality constraints. Whereas in the former problems these constraints were structured as *alldifferent* constraints, in the graph coloring this is not the case. Although a more comprehen-
Table 5.5: The running time comparison of various encodings for Clasp and Lingeling on graph colouring instances. $K$ is the number of colors used.

<table>
<thead>
<tr>
<th>Instance</th>
<th>$K$</th>
<th>Clasp</th>
<th>Lingeling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$Sp$</td>
</tr>
<tr>
<td>1-Insertion_4</td>
<td>4</td>
<td>17.2</td>
<td>34.6</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.5</td>
<td>2.0</td>
</tr>
<tr>
<td>2-FullIns_5</td>
<td>6</td>
<td>22.5</td>
<td>19.6</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>0.3</td>
<td>0.6</td>
</tr>
<tr>
<td>4-Fullins_4</td>
<td>7</td>
<td>5.9</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.2</td>
<td>0.3</td>
</tr>
<tr>
<td>anna</td>
<td>10</td>
<td>13.5</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>huck</td>
<td>10</td>
<td>6.8</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>miles500</td>
<td>10</td>
<td>51.2</td>
<td>3.7</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1458.3</td>
<td>5.2</td>
</tr>
<tr>
<td>miles750</td>
<td>10</td>
<td>109.5</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>2535.5</td>
<td>7.9</td>
</tr>
<tr>
<td>miles1000</td>
<td>10</td>
<td>179.8</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>3908.2</td>
<td>8.6</td>
</tr>
<tr>
<td>miles1500</td>
<td>10</td>
<td>254.0</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>4632.8</td>
<td>6.8</td>
</tr>
<tr>
<td>queen12_12</td>
<td>10</td>
<td>16.9</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>91.3</td>
<td>2.6</td>
</tr>
<tr>
<td>queen13_13</td>
<td>10</td>
<td>35.9</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>275.2</td>
<td>3.8</td>
</tr>
<tr>
<td>queen14_14</td>
<td>10</td>
<td>75.6</td>
<td>1.4</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>679.7</td>
<td>6.0</td>
</tr>
</tbody>
</table>

Sieve study needs to be done, we favor this justification as it is more in line with the engineering of BEE. There the order encoding is adopted for modeling bounds consistency in the CP solver (hence inequalities) but sparse encodings were also used when all different constraints need to be represented [MC12]. More interestingly, the $Sp-Or_{red}$ encoding now produces better results than the result for the order encoding for most of the instances.

Hence, we expect that in problems where disequality constraints are dominant (with respect to inequalities) and they are not structured as all different constraints, the overhead incurred by maintaining $d$ and $o$ variables and expressing the disequality constraints redundantly in both representations is more than compensated by the superior pruning obtained, resulting in significant execution speed ups.
The Pigeon-Hole Problem

Table 5.6: The running time comparison of encodings for Clasp and Lingeling on unsatisfiable Pigeon-Hole instances.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Clasp</th>
<th>Lingeling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Spa</td>
<td>Ord</td>
</tr>
<tr>
<td>10</td>
<td>0.2</td>
<td>0.3</td>
</tr>
<tr>
<td>11</td>
<td>2.1</td>
<td>3.3</td>
</tr>
<tr>
<td>12</td>
<td>26.0</td>
<td>13.2</td>
</tr>
<tr>
<td>13</td>
<td>64.9</td>
<td>72.0</td>
</tr>
<tr>
<td>14</td>
<td>560.0</td>
<td>1,013.4</td>
</tr>
<tr>
<td>15</td>
<td>6918.5</td>
<td>7,394.6</td>
</tr>
</tbody>
</table>

In the tested problems the domains are small (10 to 15 holes) which given the discussion above should favor the order encoding. However, the disequality CSP constraints are organized as an alldifferent constraint, which favors the sparse encoding. The results obtained with these encodings shown in Table 5.6 reflect this trade off, and the running times with both encodings are quite similar (again, the bimander encoding was used to model AMO with the sparse encoding). The results in Table 5.6 also show that the $S_p$–Or<sub>red</sub> encoding outperforms both the other encodings on all instances, suggesting that in these conditions the redundant representation of domains and constraints significantly pays-off.

The Round Robin and Golomb Ruler Problem

The *Golomb ruler problem* aims at finding an increasing sequence of numbers where all differences between the numbers are different (CSPLIB prob006 in [GW99]). The problem requires both disequality and inequality CSP constraints. In fact the Golomb ruler problem requires non-binary CSP constraints such as $A + B > C + D$. These constraints can be transformed in binary constraints $X > Y$ after introducing ternary equality constraints such as $X = A + B$, which requires a straightforward generalization of the techniques described above.

The *round robin problem* has a number of variants. It is widely used for scheduling tournaments (CSPLIB prob026 in [GW99]).

The results in Table 5.7 show a similar trade off between the order and the sparse encoding. The small size of the domains and the existence of inequality constraints favor the former, whereas the alldifferent constraints favor the latter, but less than before. Once again a combination of both pays off, as shown in the table for the Sp-Or encoding. Given the mix of disequality and inequality constraints of this problem, we tried the $S_p$–Or<sub>hyb</sub> encoding, that explores less redundancy (as explained in Section 4.3.2). The results show that in Clasp the redundant representation significantly improves propagation and the pruning of the
Table 5.7: The running time comparison of encodings for Clasp and Lingeling on round robin and Golomb ruler instances. The column Inst shows the number of teams.

<table>
<thead>
<tr>
<th>Solvers</th>
<th>Problem</th>
<th>Instance</th>
<th>Spa</th>
<th>Ord</th>
<th>$Sp \cdot Or_{hyb}$</th>
<th>$Sp \cdot Or_{red}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clasp</td>
<td>Round</td>
<td>8</td>
<td>0.4</td>
<td>0.4</td>
<td><strong>0.3</strong></td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>Round</td>
<td>10</td>
<td>1.8</td>
<td>1.0</td>
<td><strong>0.8</strong></td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>Round</td>
<td>12</td>
<td>115.0</td>
<td>54.6</td>
<td>41.5</td>
<td><strong>38.8</strong></td>
</tr>
<tr>
<td></td>
<td>Round</td>
<td>&gt;7200</td>
<td>1,398.4</td>
<td>709.0</td>
<td></td>
<td>651.2</td>
</tr>
<tr>
<td></td>
<td>Golomb</td>
<td>G(9,44)</td>
<td>0.91</td>
<td>0.3</td>
<td>1.1</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>Golomb</td>
<td>G(10,55)</td>
<td>2.5</td>
<td>2.5</td>
<td>9.4</td>
<td><strong>1.8</strong></td>
</tr>
<tr>
<td></td>
<td>Golomb</td>
<td>G(11,72)</td>
<td>6.7</td>
<td>101.7</td>
<td>23.2</td>
<td>47.4</td>
</tr>
<tr>
<td></td>
<td>Golomb</td>
<td>G(12,85)</td>
<td>409.53</td>
<td>1,217.3</td>
<td>442.2</td>
<td><strong>141.2</strong></td>
</tr>
<tr>
<td></td>
<td>Ruler</td>
<td>8</td>
<td>0.1</td>
<td><strong>0.1</strong></td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>Ruler</td>
<td>10</td>
<td>1.4</td>
<td>3.9</td>
<td>2.2</td>
<td><strong>1.0</strong></td>
</tr>
<tr>
<td></td>
<td>Ruler</td>
<td>12</td>
<td><strong>19.2</strong></td>
<td>79.8</td>
<td>43.3</td>
<td>64.8</td>
</tr>
<tr>
<td></td>
<td>Ruler</td>
<td>&gt;7200</td>
<td>3,472.7</td>
<td>&gt;7200</td>
<td><strong>454.6</strong></td>
<td>2,104.3</td>
</tr>
<tr>
<td></td>
<td>Lingeling</td>
<td>G(9,44)</td>
<td>0.6</td>
<td><strong>0.4</strong></td>
<td>3.9</td>
<td>2.7</td>
</tr>
<tr>
<td></td>
<td>Lingeling</td>
<td>G(10,55)</td>
<td>9.7</td>
<td><strong>1.1</strong></td>
<td>8.0</td>
<td>14.9</td>
</tr>
<tr>
<td></td>
<td>Lingeling</td>
<td>G(11,72)</td>
<td>109.5</td>
<td>137.1</td>
<td>32.4</td>
<td><strong>24.5</strong></td>
</tr>
<tr>
<td></td>
<td>Lingeling</td>
<td>G(12,85)</td>
<td>568.0</td>
<td>702.5</td>
<td>152.7</td>
<td><strong>86.5</strong></td>
</tr>
</tbody>
</table>

search space, especially in larger instances, whereas with Lingeling the benefits of redundancy are not so clear and the $Sp \cdot Or_{hyb}$ encoding might be preferable to the $Sp \cdot Or_{red}$ encoding.

To demonstrate the Sp–Or$_{hyb}$ and Sp–Or$_{red}$ encodings, we analyze the round robin problem in detail, since the problem requires the consideration of mixed CSP constraints, including disequality, inequality, and cardinality constraints. The goal of the round robin problem is to schedule a given set of $m$ teams ($m$ is even) over $m - 1$ weeks, with each week divided into $m/2$ periods, such that the following constraints are satisfied:

1. Constraint: each period is divided into two slots, in which the first team in each slot plays at home, whilst the second plays away

2. Constraint: each team plays once a week

3. Constraint: each team plays at most twice in the same period

4. Constraint: each team plays with every other team.

Constraint 1 requires the inequality constraints in which, to break the symmetry, the smaller team is placed at the first slot. Consequently, Constraint 1 favors the order encoding. The number of generated clauses is about $\frac{m^3}{2}$. Constraint 2 requires alldifferent constraints which favor the sparse encoding. The number of generated clauses is about $\frac{m^3}{4}$. Constraint 3 requires the cardinality constraint in which the
sparse encoding generates shorter clauses (ternary clauses) than the order encoding (6-ary clauses). The number of generated clauses is about $\frac{n^6}{6}$. In Constraint 4, the sparse encoding generates 4-ary clauses, whereas the order encoding generates 8-ary clauses. The number of generated clauses is about $\frac{n^8}{8}$.

Table 5.8 shows the number of variables and clauses for four SAT encodings on round robin scheduling instances. The sparse and order encoding require similar numbers of variables and clauses. As can be seen, $Sp-Or_{hyb}$ and $Sp-Or_{red}$ use the same number of variables, which is about double the number of variables of the sparse or order encodings. However, the number of generated clauses by $Sp-Or_{hyb}$ is smaller than by $Sp-Or_{red}$.

Table 5.8: Number of variables and number of clauses for SAT encodings on round robin instances. The column $Inst$ shows the number of teams.

<table>
<thead>
<tr>
<th>$Inst$</th>
<th>$Spa$</th>
<th>$Ord$</th>
<th>$Sp-Or_{hyb}$</th>
<th>$Sp-Or_{red}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Vars</td>
<td>Cls</td>
<td>Vars</td>
<td>Cls</td>
</tr>
<tr>
<td>8</td>
<td>392</td>
<td>18,144</td>
<td>720</td>
<td>72,270</td>
</tr>
<tr>
<td>10</td>
<td>810</td>
<td>72,990</td>
<td>1,530</td>
<td>74,520</td>
</tr>
<tr>
<td>12</td>
<td>1,452</td>
<td>222,552</td>
<td>2,772</td>
<td>224,730</td>
</tr>
<tr>
<td>14</td>
<td>2,366</td>
<td>564,928</td>
<td>4,550</td>
<td>567,742</td>
</tr>
</tbody>
</table>

Table 5.9 presents the number of conflicts and running time, while Table 5.10 presents the number of conflicts and the number of decisions for four SAT encodings on round robin scheduling instances for Lingeling. Bold font indicates the minimum number of conflicts or decisions for each corresponding instance. The result reveals that there is a strong correlation among: the running time, the number of conflicts, and the number of decisions.

Table 5.9: The running time and the number of conflicts for SAT encodings on round robin scheduling instances for Lingeling. The column $Inst$ shows the number of teams. A dash “-” indicates an undefined number due to timeout (7200 seconds). $Conf$ and $Sec$ indicates the number of conflicts and running time, respectively.

<table>
<thead>
<tr>
<th>$Inst$</th>
<th>$Spa$</th>
<th>$Ord$</th>
<th>$Sp-Or_{hyb}$</th>
<th>$Sp-Or_{red}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$Conf$</td>
<td>$Sec$</td>
<td>$Conf$</td>
<td>$Sec$</td>
</tr>
<tr>
<td>8</td>
<td>275</td>
<td>0.1</td>
<td>50</td>
<td>0.1</td>
</tr>
<tr>
<td>10</td>
<td>24,231</td>
<td>1.4</td>
<td>48,500</td>
<td>3.9</td>
</tr>
<tr>
<td>12</td>
<td>246,054</td>
<td>19.2</td>
<td>513,248</td>
<td>79.8</td>
</tr>
<tr>
<td>14</td>
<td>24,638,987</td>
<td>3,472.7</td>
<td>- &gt;7200</td>
<td></td>
</tr>
</tbody>
</table>

One encoding that requires a smaller number of conflicts seems to use a smaller number of decisions and has a better performance. Interestingly, for the largest instance, both $Sp-Or_{hyb}$ and $Sp-Or_{red}$ both consume less running time than the sparse and order encoding.
Table 5.10: The number of conflicts and the number of decisions for SAT encodings on round robin scheduling instances for Lingeling. The column Inst shows the number of teams. A dash "-" indicates the timeout (7200 seconds). Conf and Dec indicate the number of conflicts and the number of decisions, respectively.

<table>
<thead>
<tr>
<th>Inst</th>
<th>Spa Conf</th>
<th>Spa Dec</th>
<th>Ord Conf</th>
<th>Ord Dec</th>
<th>Sp Ord Conf</th>
<th>Sp Ord Dec</th>
<th>Sp Or Ord Conf</th>
<th>Sp Or Ord Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>275</td>
<td>2,189</td>
<td>50</td>
<td>377</td>
<td>314</td>
<td>1,862</td>
<td>1,118</td>
<td>6,187</td>
</tr>
<tr>
<td>10</td>
<td>24,231</td>
<td>158,143</td>
<td>48,500</td>
<td>218,998</td>
<td>24,618</td>
<td>137,690</td>
<td>11,137</td>
<td>61,307</td>
</tr>
<tr>
<td>12</td>
<td>246,054</td>
<td>1,985,964</td>
<td>513,248</td>
<td>2,617,062</td>
<td>321,231</td>
<td>1,512,097</td>
<td>691,375</td>
<td>3,822,824</td>
</tr>
<tr>
<td>14</td>
<td>24,638,987</td>
<td>191,219,684</td>
<td>-</td>
<td>5,681,742,39,580,417</td>
<td>11,227</td>
<td>110,63,624,249</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.4 Discussion

5.4.1 The AMO Encodings - Related Work

Chen [Che10] carried out evaluation experiments on a variant of the edge-matching problem [Heu09] with the AMO product, sequential counter, binary, and pairwise encodings. The author showed that the AMO product encoding had six out of ten instances faster than the AMO sequential counter and binary encoding, and seven instances faster than the AMO pairwise encoding. The experiment was carried out only by CircleSAT [Che].

Frisch and Giannaros [FG10] noticed that a smaller encoding (wrt. the formula size) tends to run faster. The authors experimented only on the Pigeon-Hole problem and the propagation problem (i.e., a large instance of cardinality constraint \( \leq k \) is encoded, and \( k+1 \) variables are randomly chosen and set to 1). The authors also used only one SAT solver, MiniSat2 [ES03].

Although the AMO pairwise encoding has been the oldest and most often used AMO encoding, its generated clauses grows quadratically with the domain of a variable. For that reason, other proposed encodings generate a smaller number of clauses; however they introduce auxiliary variables. Marques-Silva and Lynce [SL07] conjectured that the large number of auxiliary variables leads to the lack of predictability for SAT solvers. In order to overcome this problem, they made a SAT solver ignore the existence of these auxiliary variables by branching only non-auxiliary variables. Experimental results evaluated on six problems showed that a SAT solver becomes remarkably more robust. However, the authors only used one adapted MiniSat2 solver (see MiniSat2 at [ES03]) for only two AMO encodings: the AMO sequential counter and pairwise encodings.

Unlike previous work, we provide an empirical study of widely used and efficient AMO encodings with three state-of-the-art SAT solvers on a number of problems. In general, our work (see Section 5.2) does not fully support the results given by Chen [Che10] and Frisch and Giannaros [FG10], based on few benchmarks and only one SAT solver. The empirical study in this section provides three insights.

1. Firstly, the AMO encodings for SAT solvers are unpredictably. This empirical
evaluation may be explained by the large number auxiliary variables, which cause the lack of predictability.

2. Secondly, the running time of AMO encodings are very different. In particular, one AMO encoding may have a good performance on some problems, but it may have a bad performance on other problems. Even on a particular problem, one AMO encoding is faster in one SAT solver, but slower in another SAT solver compared with another encoding.

3. Thirdly, there is a correlation (sometimes strong correlation) among of: the running time, the number of conflicts, and the number of decisions (i.e., corresponding to the number of nodes in the search tree). Furthermore, the running time is rather related to the consumed memory.

5.4.2 What Feature Makes One Encoding Better Than Another?

Generally, different SAT encodings of CSPs yield different formula sizes and different run time behaviour of the used SAT solver. There does not seem to be general knowledge why a particular encoding performs better than others. However, before using a SAT solver a SAT instance (CNF) is mainly influenced by some of the following features:

- the number of variables (and/or literals) required (search space)
- the number of clauses (overhead when propagating variable assignments)
- the length of clauses (e.g., unit and binary)
- the strength of unit propagation (local consistency, e.g., forward checking and maintaining arc-consistency)
- the characteristics of the problems (e.g., the type of CSP constraints)

Nevertheless, one can give several counter-examples for each of the first four features. The last one, proposed in Section 4.3, should be more thoroughly tested, namely with other CSP problems and SAT solvers.

Firstly, consider the number of variables. Since the performance of SAT solvers may depend on the number of variables, it is worthwhile to propose an encoding that needs as few SAT variables as possible. Then hopefully the potential search space size can be minimized. The log encoding is such an encoding [IM94, EMW97, Wal00]. Nevertheless, from a theoretical point of view, Walsh proved (see Theorem 15 in [Wal00]) that the performance of unit propagation, performed by DPLL SAT solvers, in the log encoding is less powerful than in the direct encoding. Furthermore, from practical point of view, on the graph colouring problem, the multi-valued direct encoding (i.e., the direct encoding without at-most-one clauses) usually runs faster than the log encoding [Gel08]. More often the direct encoding outperforms the log encoding, for example, in the Hamiltonian path problem [Hoo99] or planning problems
5.4. Discussion

[EMW97]. Frisch and Peugniez pointed out [FPDN05] that the binary encoding, a variant of the log encoding, performs much worse than the direct encoding, and sometimes result in impractically large CNF formula.

Secondly, consider the number of required clauses. The experiments presented in Section 5.2 show that the AMO encoding that has a larger number of clauses may perform better than the AMO encoding that has a smaller number of clauses in terms of shorter running times, a smaller number of conflicts, and a smaller number of decisions. For example, the AMO pairwise encoding, which requires the largest number of clauses, may performs better than others. In particular, it is clearly the best on the last two instances of the all-interval series problem. Another interesting example is related to the redundant and hybrid encodings (Sp-Or_{red} and Sp-Or_{hyb} in Section 4.3). The reported results (Section 5.3) confirm the advantages over the sparse and order encodings, which are remarkably more compact than Sp-Or_{red} and Sp-Or_{hyb}.

Thirdly, consider the length of clauses. The experiments presented in Section 5.1 reveal that although the representative encodings generate long clauses, they are very competitive and usually outperform the sparse and order encodings despite the fact that these two encodings generate shorter clauses.

Fourthly, with respect to the strength of unit propagation, given equivalent branching heuristics the modern SAT solvers obtain a propagation similar to maintaining arc consistency (MAC) on the support encoding (see Corollary 7 in [Gen02]), and to a weaker form of MAC, forward checking (FC) on the direct encoding (see Theorem 13 in [Wal00]) of a original CSP. Despite what was noted above, the support encoding may perform worse than the direct encoding. For example, Prestwich showed that the support encoding has no advantage (it is much worse in many instances) over the direct encoding for the graph colouring problem [Pre03a]. Prestwich conducted his experiment with a local search SAT solver. However, it also happens in the experiment we did with systematic SAT solvers (not shown here) on the graph colouring problem.

To summarize, although reducing the size and the strength of unit propagation of an encoding is preferable, the performance of the encoding should be thoroughly tested to obtain more definite conclusions. One defining feature of the SAT community is that SAT solving significantly depends on experimental results.

Note that the main results of this chapter are also published in [HMNS12, HN13a, BHN13, HN13b, NVB13, BHN14b, BHN14a].
CHAPTER 6

The SAT vs. CP Approaches

Although SAT is a sub-field of CP, the former states and solves problems with a black-box approach, whereas the latter aims at being tunable and programmable. CP provides a powerful modelling paradigm in the sense that an application can be expressed in a natural way with a rich library of constraints. In contrast, SAT offers a minimal language in which a problem can be modelled (in conjunctive normal form - CNF).

One of the most widely used constraints naturally occurring in practical problems is the constraint \textit{alldifferent}(x_1, \ldots, x_n), which specifies that the values assigned to the variables \(x_1, \ldots, x_n\) must be pairwise different. Due to a substantial development in CP [vH01, vHK06], this constraint is intensively exploited in finite domain solvers by means of specialised global constraints. For instance, by using graph-based algorithms (see [Rég94]), the state-of-the-art CP solvers can deal with the pigeon-hole problem with 100 holes within one second. On the other hand, there is only one approach for SAT to deal with this constraint by posing the at-least-one (ALO) and at-most-one (AMO) clauses (see Section 4.2). In fact, the state-of-the-art SAT solvers need more than one hour for tackling the pigeon-hole problem with only 16 holes (see Section 5.1).

Throughout this chapter we model two problems: all-interval series (AIS) and quasigroup with holes (QWH) in SAT and CP. The two problems can be considered as a multiple permutation problem in which the variables may occur in more than one permutation problem [HSW11]. Furthermore, a permutation problem requires the constraint \textit{alldifferent}, an application-specific constraint in CP. Nevertheless, with the high performance of the SAT approach and appropriate encodings, we will show that SAT gives a much better performance over CP.
6.1 The All-Interval Series Problem

**Specification** The all-interval series (AIS) problem was first described by Hoos [Hoo98], and then it was proposed as the problem prob007 to the CSPLib [GW99]. The AIS problem can be expressed as follows.

**Problem** [GW99]. Given a positive integer \( n \), find a vector \( s = (s_0, \ldots, s_{n-1}) \), such that

1. the vector \( s \) is a permutation of \( Z_n = \{0, 1, \ldots, n - 1\} \); and
2. the interval vector \( v = (v_1, \ldots, v_{n-1}) \), where \( v_i = |s_i - s_{i-1}|, \quad 1 \leq i \leq n - 1 \), is a permutation of \( Z_n \setminus \{0\} = \{1, \ldots, n - 1\} \).

A vector \( v \) satisfying these conditions is called an AIS of size \( n \); the problem of finding such a series is the AIS problem of size \( n \).

In this chapter, we are interested in finding all possible series of a given size.

Let take a small example for the case of \( n = 6 \), the following sequence is one of solutions and the differences between the numbers are below the numbers:

\[
\begin{array}{cccccc}
0 & 5 & 1 & 4 & 2 & 3 \\
5 & 4 & 3 & 2 & 1 \\
\end{array}
\]

The AIS problem is inspired by a well-known problem occurring in serial musical composition. Most approaches successfully solving this problem have come from CP [SB, RP, GMS03, GKL+05]. On the SAT side, Hoo showed that the order 12 of the problem causes great difficulties for local search methods [Hoo98]. Alsinet et al. [ABC+02] also used a local search method for SAT to find one solution by adding redundant constraints.

This chapter studies the SAT and CP approach for finding all solutions for the AIS problem, which is an apparently hard problem and presents potential challenges [SB, RP, GMS03, GKL+05].

**Symmetry Breaking**

When a search space contains symmetries, a solver has to revisit equivalent spaces on the set of truth assignments. By breaking symmetries, the solver can avoid redundant search in sub-spaces by only visiting a few representative spaces in each equivalent state. As a result, the solver can save a significant reduction without affecting the completeness of the search. Many researches on symmetry breaking have been reported in both SAT [CGLR96, ASM03, Sak09, Alo10] and CP [GKL+05, Rég99, AN06, GPP06, AN07].

There are four symmetries in the problem AIS pointed out [GMS03]. It is illustrated by the the following example. From the above solution, one can get another solution by:
6.1. The All-Interval Series Problem

1. using the conditional symmetry [GMS03]:
\[ \{ s_0 \mapsto s_3, s_1 \mapsto s_4, s_2 \mapsto s_5, s_3 \mapsto s_0, s_4 \mapsto s_1, s_5 \mapsto s_2 \} \]
\[ 4 \ 2 \ 3 \ 0 \ 5 \ 1 \]

2. reversing the sequences:
\[ 3 \ 2 \ 4 \ 1 \ 5 \ 0 \]

3. mapping each element \( x \) of the sequence onto \( (n - 1) - x \):
\[ 5 \ 0 \ 4 \ 1 \ 3 \ 2 \]

4. doing both (2 and 3): reversing and then mapping (or mapping and then reversing):
\[ 2 \ 3 \ 1 \ 4 \ 0 \ 5 \]

There are two main methods for breaking symmetries in a model: during search and before search. Symmetry 1 can be represented by the former method, whereas the others can be represented by the latter method.

Symmetry 1 is explained as follows. The idea is that one can always find out two consecutive numbers having the same difference as the difference between the first number and the last number. Then one can split the sequence between the two numbers. In the above example the difference between the first number (0) and the last number (5) is 3, so we can split the sequence between the two consecutive numbers: 3 and 4. Gent et al. [GMS03] dealt with Symmetry 1 by proposing breaking conditional symmetry using symmetry breaking during search. Another way to break this symmetry will be studied in Section 6.1.3.

In order to eliminate Symmetry 2, one can add the constraint \( v_1 < v_{n-1} \). To eliminate Symmetry 3, the constraint \( s_0 < s_1 \) is added ([RP]). By combining these two constraints, Symmetries 2, 3 and 4 are eliminated.

Some observations should be noticed. Obviously, the difference \( n-1 \) must occur, so the numbers 0 and \( n-1 \) must be put next to each other; otherwise, there is no solution. To get the difference \( n-2 \), there are two possible cases: either the series 0,\( n-2 \) or the series 1,\( n-1 \) must occur. Therefore, the solution sequence must include the sequence either 0,\( n-1,1 \) or 0,\( n-2,0,n-1 \). However, note that by applying Symmetry 3 one can get the sequence 0,\( n-1,1 \) from the sequence \( n-2,0,n-1 \). Hence, if we only permit the sequence 0,\( n-1,1 \) or 1,\( n-1,0 \) to appear in solutions, we eliminate Symmetry 3. By fixing the order of the sequence 0,\( n-1,1 \), we are able to eliminate Symmetry 2, and simultaneously eliminate Symmetry 4. Consequently, the number of all solutions with symmetry-breaking constraints is reduced four times compared to those without symmetry-breaking constraints.
6.1.1 The CP Modeling

The AIS problem is modelled in the MiniZinc modelling language, which is input format for CPX, as follows [MS].

```plaintext
include "alldifferent.mzn";
int: n;
array[0..n-1] of var 0..n-1: s; % the vector of numbers
array[1..n-1] of var 1..n-1: v; % the interval vector

constraint alldifferent(v);
%%constraint between v and s
constraint forall (i in 1..n-1) (v[i]=abs(s[i]-s[i-1]));
constraint alldifferent(v);

%% symmetry breaking constraints, fixing the order of the
%% sequence 0,n-1,1
constraint forall (i in 0..n-2) (s[i]<=0<=>s[i+1]==n-1);
constraint forall (i in 0..n-2) (s[i]==n-1<=>s[i+1]==1);
constraint forall (i in 0..n-3) (s[i]<=0<=>s[i+2]==1);

solve :: int_search(s, first_fail, indomain_min, complete)
satisfy;
%output ["s = ",show(s),"\n", "v = ",show(v),"\n"];
```

In the line beginning with `solve::`, the parameters `first_fail`, `indomain_min`, and `complete` indicate the variable selection for integer variables, the value choice for integer variables, and the search strategy, respectively (see [MS] for further details).

As we can see, CP language allows us to express the relations between the variables of the AIS problem in the most possibly direct way. The first command line `include "alldifferent.mzn"` indicates that the global constraint `alldifferent` is used by the model. To improve the performance in CP, an important feature of a model is to use global constraints as much as possible.

6.1.2 The SAT Encoding

To demonstrate the SAT encoding of the AIS problem, we use the direct encoding, which was also used in [Hoo98, Pre09]. In the direct encoding, AMO clauses are generated by the AMO pairwise encoding since it gives a better performance compared to the other AMO encodings for the AIS problem (see Section 5.2).

Let $s_j^i$, $1 \leq i \leq n$, $0 \leq j \leq n - 1$, be Boolean variables. The variable $s_j^i$ is set to 1 if and only if the $i^{th}$ position is the integer $j$.

The SAT encoding of the constraint `alldifferent(s)`. In order to guarantee that all positions are all different, the SAT model requires the following formulas.
6.1. The All-Interval Series Problem

1. To guarantee that each position takes exactly one integer, the SAT model requires:

   - the ALO clauses: one position takes at-least-one integer
     \[
     \bigwedge_{i=1}^{n} (s_{i}^0 \lor s_{i}^1 \lor \cdots \lor s_{n-1}^i)
     \]

   - the AMO clause: one position takes at-most-one integer
     \[
     \bigwedge_{i=1}^{n} \bigwedge_{j=0}^{n-2} \bigwedge_{j'=j+1}^{n-1} \neg(s_j^i \land s_{j'}^i) \equiv \bigwedge_{i=1}^{n} \bigwedge_{j=0}^{n-2} \bigwedge_{j'=j+1}^{n-1} (\neg s_j^i \lor \neg s_{j'}^i)
     \]

2. To guarantee that each integer is assigned to exactly one position, the SAT model requires:

   - the ALO clauses: one integer is assigned to at-least-one position
     \[
     \bigwedge_{j=0}^{n-1} (s_j^0 \lor s_j^1 \lor \cdots \lor s_j^n)
     \]

   - the AMO clauses: one integer is assigned to at-most-one position:
     \[
     \bigwedge_{j=0}^{n-1} \bigwedge_{i=1}^{n-1} \bigwedge_{i'=i+1}^{n} \neg(s_j^i \land s_{j'}^i) \equiv \bigwedge_{j=0}^{n-1} \bigwedge_{i=1}^{n-1} \bigwedge_{i'=i+1}^{n} (\neg s_j^i \lor \neg s_{j'}^i)
     \]

The SAT encoding of the constraints between \( u \) and \( s \).

Let \( v_m^k \), \( 1 \leq m \leq n-1 \), \( 1 \leq k \leq n-1 \) be Boolean variables, and \( v_m^k \) is 1 if and only if the difference between \( s^i \) and \( s^{i+1} \) is \( k \). Here we must add the constraints between two sets of variables \( x_j^i \) (\( 1 \leq i \leq n \), \( 0 \leq j \leq n-1 \)) and \( v_m^k \) (\( 1 \leq m \leq n-1 \), \( 1 \leq k \leq n-1 \)).

\[
\bigwedge_{i=1}^{n-1} \bigwedge_{j_1=0}^{n-1} \bigwedge_{j_2\neq j_1}^{n-1} \left([s_{j_1}^i \land s_{j_2}^{i+1}] \rightarrow v_{j_1-j_2}^i\right)
\]

which is equivalent to

\[
\bigwedge_{i=1}^{n-1} \bigwedge_{j_1=0}^{n-1} \bigwedge_{j_2\neq j_1}^{n-1} \left[\neg s_{j_1}^i \lor \neg s_{j_2}^{i+1} \lor v_{j_1-j_2}^i\right]
\]

The SAT encoding of the constraint \textit{alldifferent}(v). To guarantee that all differences are all different, the SAT model requires the following formulas.

1. To guarantee that each difference takes exactly one value, the SAT model requires:
• the ALO clause; one difference takes at-least-one value
\[
\bigwedge_{m=1}^{n-1} (v_1^m \lor v_2^m \lor \cdots \lor v_{n-1}^m)
\]

• the AMO clauses: one difference takes at-most-one value
\[
\bigwedge_{m=1}^{n-1} \bigwedge_{k=1}^{n-1} \bigwedge_{k'=k+1}^{n-1} \neg(v_k^m \land v_{k'}^m) \equiv \bigwedge_{m=1}^{n-1} \bigwedge_{k=1}^{n-2} \bigwedge_{k'=k+1}^{n-1} \neg(v_k^m \lor \neg v_{k'}^m)
\]

2. To guarantee that each value is assigned to exactly one difference, the SAT model requires:

• the ALO clauses: one value is assigned to at-least-one integer
\[
\bigwedge_{k=1}^{n-1} (v_1^k \lor v_2^k \lor \cdots \lor v_k^{n-1})
\]

• the AMO clauses: one value is assigned to at-most-one difference\(^1\)
\[
\bigwedge_{k=1}^{n-1} \bigwedge_{m=1}^{n-2} \bigwedge_{m'=m+1}^{n-1} \neg(v_k^m \land v_k^{m'}) \equiv \bigwedge_{k=1}^{n-1} \bigwedge_{m=1}^{n-2} \bigwedge_{m'=m+1}^{n-1} \neg(v_k^m \lor \neg v_k^{m'})
\]

The SAT encoding of symmetry-breaking constraints. To guarantee the order of the sequence 0, n-1, 1 the SAT model requires the following formulas, each corresponds to three lines of breaking-symmetry constraints at Section 6.1.1:
\[
\bigwedge_{i=1}^{n-1} \left[ s_0^i \leftrightarrow s_{n-1}^{i+1} \right]
\]
\[
\bigwedge_{i=1}^{n-1} \left[ s_{n-1}^i \leftrightarrow s_1^{i+1} \right]
\]
\[
\bigwedge_{i=1}^{n-2} \left[ s_0^i \leftrightarrow s_1^{i+2} \right]
\]
which are semantically equivalent to
\[
\bigwedge_{i=1}^{n-1} \left[ \neg s_0^i \lor s_{n-1}^{i+1} \land (s_0^i \lor \neg s_{n-1}^{i+1}) \right]
\]
\[
\bigwedge_{i=1}^{n-1} \left[ \neg s_{n-1}^i \lor s_1^{i+1} \land (s_{n-1}^i \lor \neg s_1^{i+1}) \right]
\]
\[
\bigwedge_{i=1}^{n-2} \left[ \neg s_0^i \lor s_1^{i+2} \land (s_0^i \lor \neg s_1^{i+2}) \right]
\]

\(^1\)Note that Prestwich [Pre09] introduced another model that also can express this constraint. However, instead of introducing variables \( v_m^i \) the model needs a very complex formula in the sense that it generates a set of quaternary clauses of the form (see [Pre09]).
6.1.  The All-Interval Series Problem

6.1.3  Reformulation

Gent et al. [GMS03] introduced a new model for the all-interval series problem which can eliminate all symmetries. With the new reformulation, the results (shown in [GMS03]) had a speedup of more than 100 times of improvement on the best case compared with their own technique, namely conditional symmetry breaking and symmetry breaking during search.

Reformulation of AIS problem (Definition 2 in [GMS03]). Given \( n > 3 \), find a vector \( s = (s_0, \ldots, s_{n-1}) \), such that

1. \( s \) is a permutation of \( Z_n = \{0, 1, \ldots, n - 1\} \); and
2. the interval vector \( v = (v_1, \ldots, v_{n-1}) \), where \( v_i = |s_i - s_{i-1}| \),
   \( 1 \leq i \leq n - 1 \), contains every integer \( Z_n \setminus \{0\} = \{1, \ldots, n - 1\} \)
   with exactly one integer repeated; and
3. \( s_0 = 0, s_1 = n - 1, s_2 = 1 \).

Interestingly, the authors proved that the reformulation has no combinations of the four symmetries (see Section 6.1); this leads to a distinct solution (for the proof, see Lemma 2 in [GMS03]). A MiniZinc model for the above reformulation is shown below.

```plaintext
include "alldifferent.mzn";
int: n;
array[0..n-1] of var 0..n-1: s; % the vector of numbers
array[1..n-1] of var 1..n-2: v; % the interval vector

constraint alldifferent(s);
%% fix the first three elements
constraint (s[0]=0 /\ s[1]=n-1 /\ s[2]=1);
constraint forall(i in 1..n-1) (v[i]=abs(s[i]-s[i-1]));
constraint (v[n-1]=abs(s[1]-s[n]));

predicate atleastone(array[1..n-1] of var 1..n-2: t,
  var int: k, var bool:y)=
  exists(i in 0..n-1)(y \ t[i]=k);
constraint forall(x in 1..n-2)
  (atleastone(v,x,false));
solve :: int_search(x, first_fail, indomain_min, complete)
satisfy;
%output ["s = ",show(s),"\n", "v = ",show(v),"\n"];```

6.1.4 Experimental Evaluation

The experiments, reported in this section, were performed on a Intel Core 2 Quad processor with 2.66 Ghz and 3.8 GB of memory, under Ubuntu 10.04. Run times are reported in seconds.

Since Lingeling and Riss3G do not offer a configuration for finding all solutions for a CNF instance, the AIS problem is only evaluated with Clasp [GKS09] (clasp2.1.3x86_64linux). The used CP solver is Opturion’s CPX discrete optimiser, which is a constraint solver for discrete satisfaction or optimisation problems. CPX combines CP and SAT techniques in a way that allows it to take advantage of the strengths of both approaches. Like CP, CPX allows a problem to be encoded by a compact representation (see Section 6.1.1), rather than generating a SAT representation. Like SAT, CPX is able to learn from failure, a central role in the success of SAT [OSC09, FS09]. The most important reason for choosing Opturion CPX is that it is one of the state-of-the-art CP solvers. Opturion CPX dominated the others in the last CP competition with two gold medals and two bronze medals in the 2013 MiniZinc Challenge [CPc].

**Without symmetries breaking** In the experiment, the columns $S_1, S_2, S_{\sqrt{n}}$, and $S_{n/2}$ ($O_1, O_2, O_{\sqrt{n}}$, and $O_{n/2}$) refer to the representative-sparse encodings (the representative-order encodings) with corresponding partitions 1, 2, $\sqrt{n}$, and $n/2$, respectively.

Table 6.1: The running time comparison of SAT encodings performed for Clasp on AIS instances. Columns n and #Sol indicate the instance and the number of all solutions, respectively. Running times are reported in seconds.

<table>
<thead>
<tr>
<th>n</th>
<th>#Sol</th>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_{\sqrt{n}}$</th>
<th>$S_{n/2}$</th>
<th>$O_1$</th>
<th>$O_2$</th>
<th>$O_{\sqrt{n}}$</th>
<th>$O_{n/2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>296</td>
<td>1.2</td>
<td>2.8</td>
<td>3.2</td>
<td>1.6</td>
<td>2.0</td>
<td>4.1</td>
<td>3.7</td>
<td>1.6</td>
</tr>
<tr>
<td>11</td>
<td>648</td>
<td>6.9</td>
<td>19.7</td>
<td>19.7</td>
<td>34.3</td>
<td>14.0</td>
<td>18.0</td>
<td>20.1</td>
<td>30.0</td>
</tr>
<tr>
<td>12</td>
<td>1328</td>
<td>46.6</td>
<td>108.6</td>
<td>137.5</td>
<td>53.5</td>
<td>87.9</td>
<td>138.4</td>
<td>151.0</td>
<td>60.0</td>
</tr>
<tr>
<td>13</td>
<td>3200</td>
<td>309.4</td>
<td>923.6</td>
<td>832.5</td>
<td>1,645.7</td>
<td>473.5</td>
<td>880.0</td>
<td>892.9</td>
<td>1,594.8</td>
</tr>
<tr>
<td>14</td>
<td>9912</td>
<td>2,067.6</td>
<td>5,472.7</td>
<td>5,661.8</td>
<td>2,279.5</td>
<td>3,540.4</td>
<td>6,695.5</td>
<td>6,796.6</td>
<td>2,915.9</td>
</tr>
</tbody>
</table>

Table 6.1 presents the results of SAT encodings on the AIS problem. The table reveals that the representative-sparse encodings seem to perform better than the representative-order encodings. This is due to the fact that the alldifferent(s) constraint favor the representative-sparse encodings (see Section 4.3). Furthermore, the direct encoding ($S_1$) outperforms the other representative-sparse encodings ($S_2$, $S_{\sqrt{n}}$ and $S_{n/2}$) in terms of the running time. The AIS problem includes the complex disequalities involving 4-ary CSP variables: the differences between the numbers are alldifferent(s). That is a possible explanation why the latter encodings are slower than $S_1$: the overhead of expressing complex disequalities on the representative encodings is not compensated by the lower number of Boolean variables. Surprisingly, if the domains are even (10, 12, and 14) the representative encodings with the
6.1. The All-Interval Series Problem

Partitions $n/2$, $S_{n/2}$ and $O_{n/2}$, are very competitive with $S_1$ and $O_1$, respectively. Particularly, $S_{n/2}$ is quite close to $S_1$, whereas $O_{n/2}$ is considerably and consistently faster than $O_1$. The result may be explained by the different behaviour of the specific partitions of the representative encodings. This observation could lead to an interesting study.

Table 6.2: A comparison of the SAT approach using the direct encoding (abbreviated as $S_1$) produced by Clasp, and the CP approach produced by Opturion CPX (abbreviated as CPX) on AIS instances without symmetry-breaking constraints. Columns $n$ and $#Sol$ indicate the instance and the number of all solutions, respectively. Columns $Var$, $Conf$, and Decs indicate the number of variables, conflicts, and decisions, respectively. Column sec reports the running time in seconds. Bold font indicates the minimum running time for each corresponding instance.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$#Sol$</th>
<th>$S_1$</th>
<th>CPX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Var</td>
<td>Conf</td>
<td>Decs</td>
</tr>
<tr>
<td>10</td>
<td>296</td>
<td>181</td>
<td>110,248</td>
</tr>
<tr>
<td>11</td>
<td>648</td>
<td>221</td>
<td>525,711</td>
</tr>
<tr>
<td>12</td>
<td>1328</td>
<td>265</td>
<td>3,091,730</td>
</tr>
<tr>
<td>13</td>
<td>3200</td>
<td>313</td>
<td>17,040,749</td>
</tr>
<tr>
<td>14</td>
<td>9912</td>
<td>365</td>
<td>92,212,427</td>
</tr>
</tbody>
</table>

Table 6.2 summarizes the results of the SAT and CP approach on the AIS problem without symmetry-breaking constraints. As we can see, although SAT requires a significantly large number of variables, conflicts and decisions, SAT consistently and considerably outperforms CP for all instances by more than an order of magnitude, except for the first instance $n = 10$. This is probably due to the efficiency of the state-of-the-art SAT solver on dealing with conflicts and decisions. One interesting observation is that Clasp takes the similar number of conflicts compared to the number of decisions, whereas Opturion CPX needs a number of decisions which is twice as much as the number of conflicts. This is basically due to the different principle of two solvers.

With symmetries breaking Table 6.3 compares the results of the SAT and CP approach on the AIS problem with symmetry-breaking constraints. Based on the results from Tables 6.2 and 6.3, one can see that the numbers of conflicts and decisions produced by Clasp and Opturion CPX have been drastically reduced. Consequently, running times are remarkably reduced for both approaches. This significant improvement is due to the symmetry-breaking constraints. By using these constraints, Opturion CPX can solve an instance $n = 14$ in less than 1,000 seconds (Table 6.3), whereas without these constraints the instance cannot be solved in more than 10,000 seconds (Table 6.2).
Table 6.3: A comparison of the SAT approach using the direct encoding (abbreviated as $S_1$) produced by Clasp, and the CP approach produced by Opturion CPX (abbreviated as CPX) on AIS instances with symmetry-breaking constraints. Columns $n$ and $\#Sol$ indicate the instance and the number of all solutions, respectively. Columns $Var$, $Conf$, and $Decs$ indicate the number of variables, conflicts, and decisions, respectively. Column $sec$ reports the running time in seconds. Bold font indicates the minimum running time for each corresponding instance.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$#Sol$</th>
<th>$S_1$</th>
<th>CPX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$Var$</td>
<td>$Conf$</td>
</tr>
<tr>
<td>10</td>
<td>74</td>
<td>5,597</td>
<td>5,843</td>
</tr>
<tr>
<td>11</td>
<td>162</td>
<td>24,312</td>
<td>25,142</td>
</tr>
<tr>
<td>12</td>
<td>332</td>
<td>108,673</td>
<td>112,047</td>
</tr>
<tr>
<td>13</td>
<td>800</td>
<td>313</td>
<td>565,858</td>
</tr>
<tr>
<td>14</td>
<td>2478</td>
<td>365</td>
<td>2,749,196</td>
</tr>
<tr>
<td>15</td>
<td>2478</td>
<td>421</td>
<td>14,974,961</td>
</tr>
</tbody>
</table>

Table 6.4: A comparison of the SAT approach using the direct encoding (abbreviated as $S_1$) produced by Clasp, and the CP approach produced by Opturion CPX (abbreviated as CPX) on AIS instances between with and without symmetry-breaking (SB) constraints. Column $n$ indicates the instance. Running times are reported in seconds. Column $Speedup$ reports the factor by which these runtime performed by using SAT over CP. A dash means no information.

<table>
<thead>
<tr>
<th>$n$</th>
<th></th>
<th>without SB</th>
<th></th>
<th>with SB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$S_1$</td>
<td>CPX</td>
<td>Speedup</td>
<td>$S_1$</td>
</tr>
<tr>
<td>10</td>
<td>1.2</td>
<td>6.8</td>
<td>5.7</td>
<td>0.1</td>
</tr>
<tr>
<td>11</td>
<td>6.9</td>
<td>93.3</td>
<td>13.5</td>
<td>0.3</td>
</tr>
<tr>
<td>12</td>
<td>46.6</td>
<td>1,115.1</td>
<td>23.9</td>
<td>1.5</td>
</tr>
<tr>
<td>13</td>
<td>309.4</td>
<td>5,978.6</td>
<td>19.3</td>
<td>9.2</td>
</tr>
<tr>
<td>14</td>
<td>2,067.6</td>
<td>&gt; 10,000</td>
<td>-</td>
<td>56.2</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>393.8</td>
</tr>
</tbody>
</table>

As can be observed in Table 6.4, SAT consistently outperforms CP with runtime speedup of more than one magnitude in both models, without and with symmetry-breaking constraints. The interesting question is: *how do symmetry-breaking constraints affect SAT and CP?* Without symmetry-breaking constraints, the SAT model obtains speedups over CP between 5.7 and 23.9 times, whereas with symmetry-breaking constraints the speedups are between 2.0 and 17.3 times. We suspect that symmetry-breaking constraints affect the CP model more than the SAT model. This may be explained by the fact that a CP model, which is supported by rich tools for the expression of the structure of problems, can be very
structure-aware. On the contrary, SAT solvers lack any awareness of the structure of the problem since the information of a problem is lost during the SAT encoding process, resulting in a flat and homogeneous format.

**Reformulation** As can be seen in Table 6.5, the SAT approach consistently and significantly outperforms the CP approach.²

<table>
<thead>
<tr>
<th>Size</th>
<th>CPX</th>
<th>SAT</th>
<th>Speedup</th>
<th>#Sol</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.05</td>
<td>0.01</td>
<td>5.0</td>
<td>37</td>
</tr>
<tr>
<td>11</td>
<td>0.20</td>
<td>0.06</td>
<td>3.3</td>
<td>81</td>
</tr>
<tr>
<td>12</td>
<td>0.75</td>
<td>0.28</td>
<td>2.7</td>
<td>166</td>
</tr>
<tr>
<td>13</td>
<td>4.61</td>
<td>0.53</td>
<td>7.9</td>
<td>400</td>
</tr>
<tr>
<td>14</td>
<td>35.40</td>
<td>0.82</td>
<td>43.2</td>
<td>1,239</td>
</tr>
<tr>
<td>15</td>
<td>185.32</td>
<td>2.55</td>
<td>72.7</td>
<td>3,199</td>
</tr>
<tr>
<td>16</td>
<td>3658.83</td>
<td>3.17</td>
<td>1,154.2</td>
<td>6,990</td>
</tr>
<tr>
<td>17</td>
<td>16743.24</td>
<td>7.21</td>
<td>2,322.2</td>
<td>17,899</td>
</tr>
<tr>
<td>18</td>
<td>&gt; 360,000</td>
<td>21.44</td>
<td>16,791.0</td>
<td>63,837</td>
</tr>
<tr>
<td>19</td>
<td>&gt; 360,000</td>
<td>58.13</td>
<td>-</td>
<td>181,412</td>
</tr>
<tr>
<td>20</td>
<td>&gt; 360,000</td>
<td>199.14</td>
<td>-</td>
<td>437,168</td>
</tr>
<tr>
<td>21</td>
<td>&gt; 360,000</td>
<td>1,288.81</td>
<td>-</td>
<td>1,306,478</td>
</tr>
<tr>
<td>22</td>
<td>&gt; 360,000</td>
<td>4,728.95</td>
<td>-</td>
<td>4,821,338</td>
</tr>
<tr>
<td>23</td>
<td>&gt; 360,000</td>
<td>15,892.77</td>
<td>-</td>
<td>14,864,374</td>
</tr>
<tr>
<td>24</td>
<td>&gt; 360,000</td>
<td>200,302.30</td>
<td>-</td>
<td>39,404,484</td>
</tr>
</tbody>
</table>

### 6.2 The Quasigroup With Holes Problem

**Specification** A quasigroup is a square of values \(q_{ij}\), where \(1 \leq i \leq n\) and \(1 \leq j \leq n\). Each number \([1..n]\) occurs exactly once in each row and column. Achlioptas et al. [AGKS00] introduced a method for generating satisfiable quasigroup with holes (QWH) instances in which some of the \(q_{ij}\) are given. QWH is a NP-complete problem [Col84]. This problem has been used as a benchmark for SAT and CP algorithms. Like the AIS problem, QWH can be considered as a multiple permutation problem with \(2 \times n\) intersecting permutation constraints. Furthermore, Achlioptas et

²Clasp used the configuration: --number=0 --sat-p=20,25,240,-1,1 --trans-ext=dynamic --heuristic=Voids --restarts=0 --deletion=3,50 --del-init=500,10500 --del-grow=1.1,2,0.0,100,1.5 --del-cfl=+.1,0000,2000 --del-algo=sort --del-glue=2 --strengthen=local --update-lbd --ofs=2 --save-p=75 --counter-restarts=2 --counter-bump=1023 --reverse-arcs=2 --contraction=250 --loops=common
al.[AGKS00] have shown that the problem becomes the hardest when approximately 40% of the quasigroup is empty.

### 6.2.1 The CP Modeling

The QWH problem is modelled in the MiniZinc modelling language, which is a standard format for CPX [MS]. We discuss two models for the QWH problem: the integer model and the Boolean model. The first is an integer model as follow.

```plaintext
include "alldifferent.mzn";

int: n; % order of the quasigroup.
array[1..n,1..n] of var 1..n: a;

c constraint forall(i in 1..n)(
       alldifferent(j in 1..n)(a[i,j]) \/
       alldifferent(j in 1..n)(a[j,i])
);

solve satisfy;

% output | show(a[i,j]) ++ if j == n then "\n" else " " endif |
%   i in 1..n, j in 1..n |;
```

Like in the AIS problem, the first command line include "alldifferent.mzn" indicates that the global constraint alldifferent is used by the model. There is another model for the QWH problem, called the Boolean model where the variables are restricted to be Boolean. Note that each integer array element a[i,j] in the integer model is represented by an array of Booleans in the Boolean model. The following problem is from [MS].

```plaintext
int: n; % size of latin square
array[1..n,1..n,1..n] of var bool: a;

predicate atmostone(array[int] of var bool:x) =
    forall(i,j in index_set(x) where i < j)(
      (not x[i] \/
       not x[j]));

predicate exactlyone(array[int] of var bool:x) =
    atmostone(x) \/
    exists(x);

constraint forall(i,j in 1..n)(
   exactlyone(k in 1..n)(a[i,j,k]) \/
   exactlyone(k in 1..n)(a[i,k,j]) \/
   exactlyone(k in 1..n)(a[k,i,j])
);

solve satisfy;

% output | if fix(a[i,j,k]) then
% show(k) ++ if j == n then "\n" else " " endif |
% else "" endif | i,j,k in 1..n |;
```
6.2. The Quasigroup With Holes Problem

6.2.2 The SAT Encoding

To demonstrate the SAT encoding of the QWH problem, we use the direct encoding, which was also used in [Hoo98, Pre09]. In the direct encoding, AMO clauses are generated by the AMO bimander encoding (see Section 4.2.6), which is efficient for the QWH problem.

Let $q_{i,j}^z$, $1 \leq i, j, z \leq n$, be Boolean variables. The variable $q_{i,j}^z$ is set to 1 if and only if the cell $(i,j)$ at the row $i$ and column $j$ is assigned the value $z$. Hence, each cell $i,j$ requires $n$ Boolean variables.

Each number appears exactly once in every row
In order to guarantee the condition, the SAT model requires the following formulas.

1. The ALO clauses - one cell takes at-least-one integer $[1..n]$:

   $$
   \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} (q_{i,j}^1 \lor q_{i,j}^2 \lor \cdots \lor q_{i,j}^n).
   $$

2. The AMO clause - one cell takes at-most-one integer $[1..n]$, using the AMO bimander encoding.

   - First, each set $\{q_{i,j}^1, q_{i,j}^2, \ldots, q_{i,j}^n\}$ is divided into $[\sqrt{n}]$ subsets of size $[\sqrt{n}]$: $\{q_{i,j}^{1,1}, \ldots, q_{i,j}^{1,[\sqrt{n}]}\}, \ldots, \{q_{i,j}^{[\sqrt{n]},1}, \ldots, q_{i,j}^{[\sqrt{n}],[\sqrt{n}]}\}$. Then, we use the AMO pairwise encoding:

     $$
     \bigwedge_{i=1}^{[\sqrt{n}]} \bigwedge_{j=1}^{[\sqrt{n}]} \bigwedge_{l=1}^{[\sqrt{n}]} \bigwedge_{m=1}^{[\sqrt{n}]} \text{AMO}(q_{i,j}^{l,[\sqrt{n}]*m-1+1}, \ldots, q_{i,j}^{l,[\sqrt{n}]*m}).
     $$

   - Second, for each cell $(i,j)$ the bimander encoding requires a set of auxiliary propositional variable $b_{i,j}^{1}, \ldots, b_{i,j}^{[\log_2{\sqrt{n}}]}$. Then, the following clauses are generated by the constraints between each variable of a cell and auxiliary propositional variable in a subset:

     $$
     \bigwedge_{i=1}^{[\sqrt{n}]} \bigwedge_{j=1}^{[\sqrt{n}]} \bigwedge_{l=1}^{[\sqrt{n}]} \bigwedge_{m=1}^{[\log_2{\sqrt{n}}]} \neg q_{i,j}^{l,m} \land \left( \bigvee_{k=1}^{[\log_2{\sqrt{n}}]} \phi(i,j,m,k) \right)
     $$

     $\phi(i,j,m,k)$ denotes $b_{i,j}^{k}$ (or $\neg b_{i,j}^{k}$) if the bit $k$ of $m - 1$ represented by a unique binary string is 1 (or 0).

Each number appears exactly once in every column
In a similar way, we can easily generate the SAT model for every columns.
6.2.3 Experimental Evaluation

The configuration of the SAT solver and CP solver used is the same as in Section 6.1.4. The benchmark introduced by Achlioptas et al. [AGKS00] can tune the generator to output hard instances. We experimented with instances with different levels of hardness. We choose two models for CP, the integer and Boolean model. Table 6.6 presents the results of the SAT and CP approach on the QWH problem. As we can see, SAT completely outperforms both CP models for all instances.

Table 6.6 presents the CP approach and the SAT approach on the QWH problem. As can be seen, last several instances, except for the last one, consumes a large amount of running time. This result is explained since these instances consist of approximately 40% of holes are very hard, observed by Achlioptas et al. [AGKS00].

Table 6.6: Results of CP and SAT approaches for the QWH problem. Column *Instance* indicate the instance. *SAT* refers to the SAT model. Columns *CPX - Integer* and *CPX - Boolean* refer to the integer model and the Boolean model for CP, respectively. Running times are reported in seconds.

<table>
<thead>
<tr>
<th>Instance</th>
<th>SAT</th>
<th>CPX - Integer</th>
<th>CPX - Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>qwh.order10.holes100</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
</tr>
<tr>
<td>qwh.order15.holes225</td>
<td>0.01</td>
<td>0.01</td>
<td>0.39</td>
</tr>
<tr>
<td>qwh.order18.holes120</td>
<td>0.16</td>
<td>11.516</td>
<td>&gt; 10,000</td>
</tr>
<tr>
<td>qwh.order20.holes300</td>
<td>0.10</td>
<td>0.312</td>
<td>2.18</td>
</tr>
<tr>
<td>qwh.order30.holes316</td>
<td>0.14</td>
<td>&gt; 10,000</td>
<td>&gt; 10,000</td>
</tr>
<tr>
<td>qwh.order30.holes320</td>
<td>0.59</td>
<td>&gt; 10,000</td>
<td>&gt; 10,000</td>
</tr>
<tr>
<td>qwh.order33.holes381</td>
<td>48.91</td>
<td>&gt; 10,000</td>
<td>&gt; 10,000</td>
</tr>
<tr>
<td>qwh.order35.holes405</td>
<td>6.71</td>
<td>&gt; 10,000</td>
<td>&gt; 10,000</td>
</tr>
<tr>
<td>qwh.order40.holes528</td>
<td>36.98</td>
<td>&gt; 10,000</td>
<td>&gt; 10,000</td>
</tr>
<tr>
<td>qwh.order40.holes544</td>
<td>68.87</td>
<td>&gt; 10,000</td>
<td>&gt; 10,000</td>
</tr>
<tr>
<td>qwh.order40.holes560</td>
<td>28.78</td>
<td>&gt; 10,000</td>
<td>&gt; 10,000</td>
</tr>
<tr>
<td>qwh.order40.holes1600</td>
<td>1.82</td>
<td>&gt; 10,000</td>
<td>&gt; 10,000</td>
</tr>
</tbody>
</table>

To obtain a further comparison between SAT and CP, we try to find all the solutions for an empty QWH (i.e., no holes). The empty QWH, usually called Latin squares, occur in many applications, for example in statistics and mathematics and error correcting codes (see Part III.1 in [CD06] and [CKL04]).

Table 6.7 shows the results of the CP approach and the SAT approach. As we can see, the performance of SAT is clearly superior to the performance of CP again.
Table 6.7: Results of CP and SAT approaches for the QWH problem. Column \#Sol indicates the number of all solutions, while the column \( n \) indicate the order of QWH. SAT refers to the SAT model. Columns CP - Integer refers to the integer model for CP, which produces a much better result compared to the Boolean model. Running times are reported in seconds.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( SAT )</th>
<th>CPX - Integer</th>
<th>#Sol</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.000</td>
<td>0.001</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>0.000</td>
<td>0.037</td>
<td>576</td>
</tr>
<tr>
<td>5</td>
<td>1.030</td>
<td>63.194</td>
<td>161,280</td>
</tr>
<tr>
<td>6</td>
<td>8.944.4</td>
<td>&gt; 10,000</td>
<td>812,851,200</td>
</tr>
</tbody>
</table>

6.3 The Langford Problem

**Specification**

Given a value of \( n \), a Langford sequence is a permutation of the sequence of \( 2 \times n \) numbers \( 1, 1, 2, 2, \ldots, n, n \), in which the two 1s are one unit apart, the two 2s are two units apart, and more generally the two \( k \)s are \( k \) units apart \((1 \leq k \leq n)\). The Langford problem (CSPLIB prob024 in [GW99]) is the task of constructing all Langford sequences. For example, the sequence \( 2 - 3 - 1 - 2 - 1 - 3 \) is a Langford sequence with \( n = 3 \).

In a Langford sequence, a number \( i \in [1..n] \) occurs twice. We call the first \( i \) and the second \( i \). To encode the problem more easily, we consider the second \( i \) as a number \( i + n \). We can re-formalize the Langford problem as follow.

Finding a sequence of a permutation of \( 2 \times n \) integer ranging from 1 to \( 2 \times n \) in such a way that there are exact \( i \) numbers appearing between the number \( i \) and the number \( n+i \) \((1 \leq i \leq n)\).

Consequently, the Langford sequence in the above example becomes: \( 2 - 3 - 1 - 5 - 4 - 6 \).

Is has been observed that from a given Langford sequence, one can obtain another solution by simply reversing the given one. It means that the Langford problem contains a symmetry. For example, once we have a solution \( 2 - 3 - 1 - 2 - 1 - 3 \), then we have another one by reversing the given one: \( 3 - 1 - 2 - 1 - 3 - 2 \). The symmetry constraints can be simply broken by ordering the first and last number.

Knuth [Knu08] mentioned that the problem of finding all Langford sequences is equivalent to solving an instance of the exact cover problem [GJ90]. In 1967 at Mathematical games of Scientific American [Gar78], one mathematician proved that the solutions of the Langford problem \((L(2, n))\) exist if only if \( n \) is in the form of \( 4k \) or \( 4k - 1 \) for \( k \in N\setminus\{0\} \).
6.3.1 The CP Modeling

The Langford problem is one of constraint satisfaction problems [Smi01] and it has been approached in different ways.

The Langford problem is modelled in the MiniZinc modelling language, which is a standard format for CPX [MS].

```zinc
include "alldifferent.mzn";
int: n;
set of int: positionDomain = 1..2*n;
array[positionDomain] of var positionDomain: position;
array[positionDomain] of var 1..n: solution;

constraint forall(i in 1..n) (  
    position[i+n] = position[i] + i+1  
    /\ solution[position[i]] = i  
    /\ solution[position[n+i]] = i
);
constraint all_different(position);

% symmetry breaking
constraint solution[1] < solution[2*n];
solve satisfy;
%
% output [ show(solution), "\n"];
```

6.3.2 The SAT Encoding

Let $t^i_j$, $1 \leq i \leq 2*n$, $1 \leq j \leq 2*n$, be Boolean variables. The variable $t^i_j$ is set to 1 if and only if the number $i$ appears at the position $j$.

Each number appears exactly once in the sequence

1. The ALO clauses - one cell takes at-least-one integer $[1..2*n]$:

   $$  
   \bigwedge_{i=1}^{2n} \bigwedge_{j=1}^{2n} (t^1_{i,j} \lor t^2_{i,j} \lor \cdots \lor t^{2n}_{i,j}).  
   $$

2. The AMO clause - one position takes at-most-one integer $[1..2*n]$, using the AMO bimander encoding (see Section 4.2.6).

   - First, each set $t^1_{i,j} \lor t^2_{i,j} \lor \cdots \lor t^{2n}_{i,j}$ is divided into $\lceil \sqrt{2*n} \rceil$ subsets of size $\lceil \sqrt{2*n} \rceil$: $\{t^1_{i,j}, \ldots, t^{\lceil \sqrt{2*n} \rceil}_{i,j}\}$, $\{t^1_{i,j} - \lceil \sqrt{2*n} \rceil, \ldots, t^{\lceil \sqrt{2*n} \rceil}_{i,j}\}$, $\ldots$, $\{t^1_{i,j}, \ldots, t^{2*n} - \lceil \sqrt{2*n} \rceil + 1, \ldots, t^{2*n}_{i,j}\}$. Then, we use the AMO pairwise encoding:

   $$  
   \bigwedge_{i=1}^{\lceil \sqrt{2*n} \rceil} \bigwedge_{l=1}^{\lceil \sqrt{2*n} \rceil} \bigwedge_{m=1}^{\lceil \sqrt{2*n} \rceil} AMO(t^1_{i,j} \lceil \sqrt{2*n} \rceil (m-1)+1, \ldots, t^{\lceil \sqrt{2*n} \rceil \lceil \sqrt{2*n} \rceil m)).
   $$
6.3. The Langford Problem

- Second, for each position \((i,j)\) the bimander encoding requires a set of auxiliary propositional variable \(b^1_{i,j}, \ldots, b^{[\log_2 \sqrt{2n}]}_{i,j}\). Then, the following clauses are generated by the constraints between each variable of a cell and auxiliary propositional variable in a subset:

\[
\bigwedge_{i=1}^{\lfloor \sqrt{2n} \rfloor - 1} \bigwedge_{j=1}^{\sqrt{2n} - 1} \bigwedge_{m=1}^{\lfloor \sqrt{2n} \rfloor - 1} \bigwedge_{p=1}^{\lfloor \sqrt{2n} \rfloor - 1} \left[ -l_{i,j}^{(\lfloor \sqrt{2n} \rfloor - 1)^p} + p \land \left( \bigvee_{k=1}^{\lfloor \log_2 \sqrt{2n} \rfloor} \phi(i, j, m, k) \right) \right],
\]

where \(\phi(i,j,m,k)\) denotes \(b^k_{i,j}\) (or \(-b^k_{i,j}\)) if the bit \(k\) of \(m - 1\) represented by a unique binary string is 1 (or 0).

There are exact \(i\) numbers between the number \(i\) and the number \(n + i\) \((1 \leq i \leq n)\)

If the number \(i\) appears at the position \(j\), then the number \(i + n\) must appear at the position \(j + i\):

\[
\bigwedge_{i=1}^{\lfloor \sqrt{2n} \rfloor - 1} \bigwedge_{j=1}^{\lfloor \sqrt{2n} \rfloor - 1} \left[ l_{i,j} \leftrightarrow l_{i,j+i+1} \right],
\]

which is semantically equivalent to:

\[
\bigwedge_{i=1}^{\lfloor \sqrt{2n} \rfloor - 1} \bigwedge_{j=1}^{\lfloor \sqrt{2n} \rfloor - 1} \left[ (-l_{i,j} \lor l_{i,j+i+1}) \land (l_{i,j} \lor -l_{i,j+i+1}) \right].
\]

Breaking symmetry
The number at the first sequence must be less than the number at the last sequence:

\[
\bigwedge_{i=1}^{n} \left[ l_{i,1} \rightarrow \bigvee_{i'=i+1}^{2n} l_{i',2n} \right],
\]

which is semantically equivalent to:

\[
\bigwedge_{i=1}^{n} \left[ l_{i,1} \lor \bigvee_{i'=i+1}^{2n} l_{i',2n} \right].
\]

6.3.3 Experimental Evaluation

The configuration of the SAT solver and CP solver used is the same as in Section 6.1.4. Table 6.8 compares the running time between the CP approach and the SAT approach in finding the first solution for the Langford problem. As the result shown, SAT is far faster than CP.
Table 6.8: Results of CP and SAT approaches for the Langford problem in finding the first solution. Columns $n$ indicates the instance. Column *Speedup* reports the factor by which these runtime performed by using SAT over CP. A dash means no information. Running times are reported in seconds.

<table>
<thead>
<tr>
<th>Size</th>
<th>CPX</th>
<th>SAT</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>7.29</td>
<td>0.08</td>
<td>91.1</td>
</tr>
<tr>
<td>36</td>
<td>3.62</td>
<td>0.15</td>
<td>24.1</td>
</tr>
<tr>
<td>39</td>
<td>68.32</td>
<td>0.21</td>
<td>325.3</td>
</tr>
<tr>
<td>40</td>
<td>16.91</td>
<td>0.08</td>
<td>221.3</td>
</tr>
<tr>
<td>43</td>
<td>59.52</td>
<td>0.14</td>
<td>425.1</td>
</tr>
<tr>
<td>44</td>
<td>23.85</td>
<td>0.23</td>
<td>103.7</td>
</tr>
<tr>
<td>47</td>
<td>211.17</td>
<td>0.22</td>
<td>918.1</td>
</tr>
<tr>
<td>48</td>
<td>97.61</td>
<td>0.23</td>
<td>424.4</td>
</tr>
<tr>
<td>51</td>
<td>33.97</td>
<td>0.32</td>
<td>106.1</td>
</tr>
<tr>
<td>52</td>
<td>37.10</td>
<td>0.34</td>
<td>109.1</td>
</tr>
<tr>
<td>55</td>
<td>240.70</td>
<td>0.71</td>
<td>339.0</td>
</tr>
</tbody>
</table>

To compare further SAT and CP, Table 6.9 presents the results between CP and SAT in finding all the solution (returning UNSAT for unsatisfiable instances). Again, the SAT approach significantly and constantly outperforms the CP approach.

Table 6.9: Results of CP and SAT approaches for the Langford problem. Columns $n$ indicates the instance. *#Sol* shows the number of all solutions, and *UNSAT* means the instance is unsatisfiable. Column *Speedup* reports the factor by which these runtime performed by using SAT over CP. A dash means no information. Running times are reported in seconds.

<table>
<thead>
<tr>
<th>Size</th>
<th>CPX</th>
<th>SAT</th>
<th>Speedup</th>
<th>#Sol</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0.08</td>
<td>0.01</td>
<td>8.0</td>
<td>26</td>
</tr>
<tr>
<td>8</td>
<td>0.83</td>
<td>0.02</td>
<td>41.5</td>
<td>150</td>
</tr>
<tr>
<td>9</td>
<td>8.25</td>
<td>0.24</td>
<td>34.4</td>
<td>UNSAT</td>
</tr>
<tr>
<td>10</td>
<td>297.31</td>
<td>1.68</td>
<td>176.9</td>
<td>UNSAT</td>
</tr>
<tr>
<td>11</td>
<td>&gt; 10,000</td>
<td>7.15</td>
<td>&gt; 1,398.5</td>
<td>17,702</td>
</tr>
<tr>
<td>12</td>
<td>&gt; 10,000</td>
<td>54.48</td>
<td>&gt; 183.6</td>
<td>108,144</td>
</tr>
</tbody>
</table>

6.4 The Golomb Ruler Problem

**Specification**

A Golomb ruler (CSPLIB prob006 in [GW99]) *golomb(n,d)* aims at finding a vector $g$, with $n$ elements in strictly increasing order with domain [0..d], such that all differences between any two elements are different. In fact, there are $n(n - 1)/2$
such differences, that is, $g_j - g_i$ ($1 \leq i < j \leq n$). To break a symmetry, we add the constraint to guarantee that the first difference is less than the last difference: $g_2 - g_1 < g_n - g_{n-1}$. Such a ruler is said of order $n$ and length $d$.

The Golomb ruler problem appears in numerous practical applications, for example information theory and error correction [RB67], radio frequency selection [FS77], radio antenna placement [TMGWS08].

In the context of SAT encoding, the order encoding (Section 3.2) slightly outperforms the sparse encoding (Section 3.1), and the $Sp-Or$ encoding (see Section 4.3.2). Here we do not present the SAT encoding, but show the experiment.

### 6.4.1 Experimental Evaluation

To model the Golomb ruler problem, we use the order encoding for SAT (see Section 3.2), and MiniZinc for CP. The configuration of CP solver used is the same as in Section 6.1.4. In this problem, although the performance of SAT outperforms CP, but it is not so strong as the previous problems. Hence, we shows the results produced by three SAT solvers: Claps, Lingeling and Riss3G.

Table 6.10 summarizes the results from CP and SAT approaches. It is observed that SAT is clear better than CP for all instances. It seems that the unsatisfiable instances are “harder” than the satisfiable ones for each order.

Table 6.10: Results of CP and SAT approaches for the Golomb ruler problem in finding the first solution. Columns $n$ indicates the order. $UN/SAT$ indicates the satisfiability or unsatisfiability of instances. The bold instances of the column $d$ indicate the optimal length for the corresponding order. Running times are reported in seconds.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$d$</th>
<th>CPX</th>
<th>Clasp</th>
<th>Riss3G</th>
<th>Lingeling</th>
<th>UN/SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>24</td>
<td>0.23</td>
<td>0.05</td>
<td>0.05</td>
<td>0.01</td>
<td>UN</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>0.03</td>
<td>0.02</td>
<td>0.03</td>
<td>0.02</td>
<td>SAT</td>
</tr>
<tr>
<td>8</td>
<td>33</td>
<td>1.71</td>
<td>0.38</td>
<td>0.26</td>
<td>0.41</td>
<td>UN</td>
</tr>
<tr>
<td></td>
<td>34</td>
<td>17.82</td>
<td>0.26</td>
<td>0.16</td>
<td>0.30</td>
<td>SAT</td>
</tr>
<tr>
<td>9</td>
<td>43</td>
<td>17.80</td>
<td>3.17</td>
<td>1.32</td>
<td>3.10</td>
<td>UN</td>
</tr>
<tr>
<td></td>
<td>44</td>
<td>12.11</td>
<td>0.31</td>
<td>0.41</td>
<td>0.41</td>
<td>SAT</td>
</tr>
<tr>
<td>10</td>
<td>54</td>
<td>383.76</td>
<td>29.03</td>
<td>6.13</td>
<td>16.90</td>
<td>UN</td>
</tr>
<tr>
<td></td>
<td>55</td>
<td>18.44</td>
<td>2.50</td>
<td>5.48</td>
<td>1.14</td>
<td>SAT</td>
</tr>
<tr>
<td>11</td>
<td>71</td>
<td>&gt;10,000</td>
<td>994.86</td>
<td>233.07</td>
<td>319.92</td>
<td>UN</td>
</tr>
<tr>
<td></td>
<td>72</td>
<td>269.08</td>
<td>101.07</td>
<td>28.23</td>
<td>137.58</td>
<td>SAT</td>
</tr>
<tr>
<td>12</td>
<td>84</td>
<td>&gt;10,000</td>
<td>4082.76</td>
<td>1591.15</td>
<td>2326.45</td>
<td>UN</td>
</tr>
<tr>
<td></td>
<td>85</td>
<td>&gt;10,000</td>
<td>1217.68</td>
<td>1925.63</td>
<td>702.51</td>
<td>SAT</td>
</tr>
</tbody>
</table>
6.5 Other Problems

In this section, we discuss other problems and how SAT and CP deal with them.

The Pigeon-Hole Problem
The problem is simply modelled by the constraint *alldifferent*. In CP, this constraint is exploited in finite domain solvers by means of specialised global constraints. Most CP solvers integrate the graph-based algorithms (see [Reg94]) to cope with the constraint *alldifferent*. Modern CP solvers can solve the Pigeon-Hole problem for the number of holes $h = 300$ less than one second, whereas state-of-the-art solvers requires more than one hour to deal with only $h = 17$.

The Graph Colouring Problem
The graph colouring problem is an important problem with applications in many domains. One can find the best upper bounds for many benchmark instances in [cola]. There are a large number of approaches to obtain the state-of-the-art results.

Few studies on the graph colouring problem have been investigated in SAT [Hoo99, Vel07a, Gel08]. On the other hand, CP has demonstrated as a powerful approach for solving this problem with numerous instances, provided by many techniques: hybrid evolutionary algorithms [GH99], coloration neighbourhood search [Pre02], adaptive memory algorithm [GHZ08], quantum annealing [TC11b], distributed hybrid quantum annealing algorithm [TC11a], independent set extraction [WH12]. It has been observed that CP is more efficient than SAT for the graph coloring problem.

The Hamiltonian Cycle and Hidoku Problems
Due to the Hidoku problem is an instance of the Hamiltonian cycle problem, we mainly discuss on the latter. A number of real world applications are closely related to the Hamiltonian cycle problem, for example, the the traveling salesman problem (TSP). In SAT, some studies have been introduced, e.g., the compact encoding [Hoo99] and the absolute encoding [Pre03b, VG]. In CP, many researchers have used different algorithms to deal with this problem [Mar83, Gou91, Koc92, ABC07]. By using linear programming, the largest practical TSP instances consisting of 85,900 cities was solved [ABC07] successfully.

Interestingly, Jäger and Zhang combined two well-known combinatorial problems, assignment problem and SAT, to get significant improvement [JZ10].

The Open Shop Problem
Several open instances have been recently solved by SAT solving. For example, Tamura et al. found and proved the optimal results for 192 instances, including three previously undecided problems [TTKB09]. Nevertheless, most of state-of-the-art results have been solved by CP [BTW93, BHJW97, ZTI08]. Furthermore, many real-world applications, which are variants of the open shop problem, have been
successfully solved by CP, for example flow shop sequencing [RY98, RMA06, RS07] and job shop scheduling [BJS94, Bec07, BFW11, TTDB14].

The Round Robin Problem
Like the open shop problem, few studies in SAT have been done for the round robin problem [BM, MM06], whereas many important results have been reported for this problem and its variants in CP [Régo1, Trí03, PW06, LRZ06, RT08, ZM12].

6.6 Conclusions

To get a broad overview of SAT and CP, we strongly refer the reader to [BH2006]. The authors presented important feathers of both areas, ranging from modellling to typical areas of application, as well as architecture and algorithms. In this thesis, we only compare SAT and CP based on the benchmarks, which have been used.

It has been observed that the all-interval series (AIS) and quasigroup with holes (QWH) problems include are a multiple permutation problem which consist $2 \times n$ intersecting permutation constraints. The Langford and Golomb ruler problems are a permutation problem. Each permutation constraint is equivalent to the alldifferent constraint, one of the most studied and used global constraint in CP [Régo94, vH90, vHK06]. On the other hand, to translate the permutation constraint to SAT, one has only one approach by decomposing the constraint into pairwise disequality constraints $X \neq Y$. Then, each of disequality constraints results in a set of binary clauses in SAT (see Section 4.3).

Nevertheless, we have shown that by using a state-of-the-art solver and an appropriate encoding the SAT approach obtains a higher performance (often very significantly) over the CP approach. In particular, the former surpasses the latter on all instances, with runtime speedups of one to two orders of magnitude in the AIS problem, and with runtime speedups of one to four orders of magnitude in the QWH and Langford problems. For the Golomb ruler problem, the gap between CP and SAT performances is smaller but SAT, demonstrated by three solvers, is still clear faster than CP.

In addition, with the reformulation on the AIS problem, the SAT approach is clearly superior to the CP approach, with runtime speedups of one to four orders of magnitude. Furthermore, the speedup is increasing with the size of the instance. To the best of your knowledge, the largest order of the problem which has been solved is 20, reported in [GMS03] (with 53,431.50 seconds). Interestingly, we have solved four open instances with respect to the order of problem from 21 to 24.

Throughout benchmarks, we roughly conjecture as follow.

1. SAT may outperform CP on combinatorial problems (e.g., the AIS, QWH, Langford, and Golomb ruler problems) in which the problems consist of simple constraints, e.g., equalities and/or disequalities, whereas

2. CP seems to be more efficient than SAT on scheduling problems (e.g., the round robin and open shop problems) in which the problems include many
different and complex constraints. For example inequalities and cardinality constraints (at-most-k). This result may be explained by the facts that:

- the complex constraints require an explosion of SAT clauses to encode;
- and
- CP, on the other hand, provides: (1) means to directly represent problems in such a way that a CP solver can tune to obtain the best performances; (2) a number of algorithms for dealing with hard applications, for example hybrid, evolutionary, heuristic, genetic, greedy, and local search algorithms; and (3) a rich of library for global constraints, especially the alldifferent constraint, which are embedded in most CP solvers.

One of the results of this chapter is published in [NS14].
CHAPTER 7

Conclusions

In computer science, it has been observed that Boolean satisfiability (SAT) is a very simple language for encoding many hard applications, ranging from artificial intelligence to industrial hardware design and verification. Interestingly, SAT solving has been one of the most successful automated reasoning technologies in the last decade. One can easily measure the great interest in SAT solvers through the regular SAT solver competitions [SAT]. Understanding SAT encodings, on the other hand, is still very limited and often challenging.

This thesis has provided a systematic study of SAT encodings of finite constraint satisfaction problems (CSPs). Its four main contributions and the respective issues for future work are summarized in the followings sections.

7.1 SAT Encodings of Finite CSPs

A CSP is defined by a set of variables, a set of domains, and a set of constraints. This thesis has studied SAT encodings by separately investigating SAT encodings of CSP domains and SAT encodings of CSP constraints. Based on our knowledge of SAT solving, the independent consideration of these two parts may help the SAT community to easily study, avoid confusion and facilitate progress in SAT encoding. This thesis has provided a novel approach (Chapters 3 and 4), presenting a comprehensive and informative survey, and a careful comparison among SAT encodings.

7.2 SAT Encodings of Finite CSP Domains

The more SAT-encoding methods one has, the better is the chance of achieving success with solving practical problems. This is probably one of the most important encouragements of the SAT community. This thesis has introduced two of the representative encodings (see Section 3.6) for modeling CSPs as SAT: the representative-sparse and representative-order encoding. These new encodings, professed representative variant of the sparse and order encodings, aim not only to take advantage of the sparse and order encodings but also to utilize a significantly smaller number of SAT variables.

From a theoretical point of view, the new encodings can be parameterised by different sizes of the first level. In general, it is not possible to compare these encodings with their flat counterparts (the sparse and order encodings) which are special cases of the former encodings with one single partition.
From a practical point of view, the experimental results (Section 5.1) with a set of benchmarks have shown that, regardless of the variability of run times in different SAT solvers, the representative encodings are competitive and usually outperform (sometimes very significantly) the sparse and order encodings, two most widely used SAT encodings. Note that this holds with the exception of CSP problems where inequality constraints dominate, where the order encoding is still the best option. When comparisons were made with the sparse and order encodings, the proposed encodings that used only two levels in the hierarchy performed very well whereas all experiments with three or more levels were clearly less efficient. We suspect that the representative encodings may get the greatest benefit of balance between the lower number of variables and the powerful propagation.

Future Work More work remains to be done to assess the merit of these new encodings. In particular, we intend to further investigate how to tune the number of partitions for a particular CSP to obtain a higher performance (see Table 6.1 in Section 6.1.4). One appealing idea would be to apply some redundancy for the second level of the representative encodings, as outlined in Section 4.3 (see also [BHN14a]). By using the representative encodings a SAT-based CSP solver, like Sugar [TTKB09] with advanced improvements, like BEE [MC12] could be developed.

7.3 SAT Encodings of the At-Most-One Constraint

Regardless of various SAT encodings proposed, the sparse encoding is the most widely used. Consequently, the AMO SAT-encodings, which are used in the sparse encoding, have attracted much investigation. With respect to this topic, presented in Section 4.2, this thesis has several contributions.

Firstly, we have proposed a new encoding, namely the AMO bimander encoding (Section 4.2.6), which is very competitive with others. Secondly, we have also presented a brief survey of most well-known AMO SAT-encodings with a running example. For each AMO SAT-encoding, we have shown not only the unit propagation strength but also the number of generated clauses.

Thirdly, a significant insight is the relationship between the auxiliary variables required by the AMO SAT-encoding and the variables used by its corresponding SAT encoding of finite CSP domains if it exists (Section 4.2). As a result, one could use a channeling constraint for the redundant and hybrid encoding. For example, the auxiliary variables required by the AMO sequential counter encoding are exactly the variables used by its corresponding SAT encoding, the order encoding (see Section 4.2.7.1). Consequently, these auxiliary variables can be used for the challenging constraint, which is used by $Sp$–Or$_{red}$ and $Sp$–Or$_{hyb}$ (see Section 4.3.2).

Fourthly, we have shown the similarity of the ladder, sequential, relaxed ladder, regular, unary representation, and order encodings (Section 4.2.7.1). We hope that this work could help some people to avoid the confusion among encodings, which were presented on some literature.
7.4. SAT Encodings of Linear CSP Constraints

Fifthly, we have been conducted an empirical study of various AMO SAT-encodings (Section 5.2). Regardless of three SAT solvers on various benchmarks, several important observations can be drawn from them:

- In terms of running time, the AMO SAT-encodings are significantly diverse. One AMO SAT-encoding may perform variably not only on a different benchmarks but also on the same benchmarks with different solvers compared to the other AMO SAT-encodings.

- In general the running time is somewhat related to the number of conflicts, whereas the number of conflicts is closely related to the number of decisions.

- Interestingly, the running time is nearly related to the memory used, i.e., an AMO SAT-encoding which performs faster may consume less memory than a slower AMO SAT-encoding.

Future Work An interesting topic would be studying how the number of disjoint subsets in some AMO SAT-encodings, like the product, bimander and commander encoding, could affect these encodings in realistic problems. Based on the guidelines in Section 7.4, the AMO SAT-encodings should be more thoroughly tested to obtain some definite guidelines. In particular, one would like to know which AMO SAT-encoding should be used for a particular type of problems.

7.4 SAT Encodings of Linear CSP Constraints

One of the biggest challenges for SAT encodings of CSPs is how to deal with encoding issues concerning the guidelines that could lead to effective encodings of new problems? To the best of our knowledge, this has been studied only in very few works. With respect to this topic presented in Section 4.3, this thesis has given several contributions.

Firstly, this thesis has investigated the efficiency of the two widely used SAT encodings, the sparse and order encodings, and proposed their combination in certain types of CSPs, in particular, the most common binary CSP problems with the usual relational operators (i.e., linear constraints).\(^1\)

Secondly, the experimental results (in Section 5.3) have revealed three important guidelines:

- The order encoding is much more efficient in problems where inequality constraints are dominant.\(^2\) In fact, this encoding is specially tailored to represent interval variables, and in this case, the CSP constraints only affect the variable domain bounds. More surprisingly, this encoding is also more efficient in problems dominated by disequality constraints over variables with small domains.

\(^1\)The constraints have the form $X \pm c \triangleright Y$, where $X$ and $Y$ are integer variables, $c$ is a constant, and $\triangleright$ is a relational operator ($\in \{\neq, \neq, \geq, \leq\}$).

\(^2\)inequality constraints have the form of $X \pm c \triangleright Y$, where $\triangleright \in \{>, \geq, <, \leq\}$. 
and with no particular structure in variables with small domains: we suspect that in these problems the small domains make bounds consistency the most effective consistency to maintain.

- The sparse encoding is usually more efficient in problems dominated by disequality constraints, especially when they are structured as several \textit{alldifferent} constraints.\footnote{disequality constraints have the form of }\mathit{X} \pm \mathit{c} \triangleright Y\text{, where }\triangleright \in \{=, \neq\}.

- In case of CSP problems with heterogeneous constraints, the order encoding is usually, but not necessarily, preferable to the sparse encoding.

Thirdly, we have shown how channeling constraints can be used in SAT (a term used in Constraint Programming - CP). In CP, to improve the high performance one can combine different representations of the same problem by bridging them via so-called channeling constraints [Wal01, Smi02, DdVC03]. However, channeling constraints have not been studied in SAT (see page 10 in [BHZ06]). We hope that this work provides the SAT community with an alternative encoding of CSPs and narrows the gap between CP and SAT modelling.

Fourthly, combining the two models through a channeling constraint often makes it possible to benefit from both worlds while not incurring significant overhead. Consequently, we have proposed two new encodings: \textit{Sp–Or\textsubscript{red}} - a redundant model and \textit{Sp–Or\textsubscript{hyb}} - a hybrid model. The reported results also confirm the advantage of the \textit{Sp–Or\textsubscript{red}} encoding, namely, when the CSP problems include disequality constraints which are not structured as \textit{alldifferent} constraints, with small to medium domains, possibly mixed with inequality constraints. \textit{Sp–Or\textsubscript{red}} \textit{redundantly} (inspired by the redundant modelling in CP) combines the propositional variables and constraints used in both the sparse and order encodings, and despite the overhead of maintaining both representations this redundancy pays off in execution time, due to the better pruning achieved by the SAT solvers. Even when either the sparse or the order encoding in isolation yield the fastest running times, the overhead presented by the \textit{Sp–Or\textsubscript{red}} encoding with respect to the best encoding is never very significant, making it a fairly 'robust' encoding; it is able to produce fast SAT encodings without sophisticated optimization techniques. Interestingly enough, we experimentally showed that \textit{Sp–Or\textsubscript{hyb}} significantly outperforms the sparse and order encodings on problems containing both, inequality and disequality constraints.

\textbf{Future Work} Several further research directions are opened:

- These above guidelines should be more thoroughly tested, namely with other CSPs and SAT solvers, together with the identification of structural features of relevance in these problems. In particular we are interested to investigate robustness of the proposed \textit{Sp–Or\textsubscript{red}} encoding with CSPs containing non-linear constraints and more general non-binary constraints, for example, table constraints.
7.5. SAT Encodings: an Ultimate Progress

- The experimental results should be complemented with a better characterisation of the code produced by the encodings: not only the number and length of the clauses produced but also the percentage coming from each type of CSP constraints. Additionally, ongoing experiment should have a more detailed profiling of the SAT solver execution so as to better understand the influence between the encodings and the features of solvers (e.g., no-good learning), and to understand how to tune the parameters of SAT solvers to improve their performance in solving CSPs with specific “constraint patterns”.

- By combining the log encoding [IM94, Wal00] and the sparse encoding, one can obtain new encodings, say $Sp-Lo_{red}$ and $Sp-Lo_{high}$. The challenge is to find a specific type of problems in which the log encoding may perform considerably well.

7.5 SAT Encodings: an Ultimate Progress

Many researchers claim that the tremendous advances in the speed and capacity of SAT solvers allow us to effectively solve many hard problems by appropriately encoding them into SAT instances [Hoo99, AGKS00, BB03, AdV+04, BB04, GARK07, TTKB09, Zha09, LZMS11, PJ11, JP12]. In Chapter 6 we have added additional weight to this claim by showing that SAT consistently and significantly outperforms CP on the all-interval series, quasigroup with holes, Langford, and Golomb ruler problems, which include the constraint \texttt{alldifferent}, one of the very application-specific constraints in CP. We also open a very interesting but intensive work to point out the characteristics of a specific problem that make us decide whether to use CP or SAT.

It is well-acknowledged that the success of SAT solving does not only depend on SAT solvers but also on how problems are encoded into SAT. However, the question: “What feature makes one encoding better than another?” (see Section 5.4.2) needs to be investigated further. Generally, one may not predict the performance of SAT solvers through several features of a SAT encoding (the number of variables/literals/clauses, the length of clauses and the strength of unit propagation). The lack of predictability is probably due to the large number of auxiliary variables, which are introduced during the process of SAT encodings (see Section 5.4.2). Interestingly, this thesis provides an interesting contribution by classifying the characteristics of problems, that is, the type of CSP constraints (see Sections 4.3.2 and 5.3). Through this contribution, the performance of many problems may be predictable, although it should be more thoroughly tested.

In conclusion, understanding an in-depth investigation of SAT encodings is always one of crucial goals for solving problems by SAT. This thesis is the first attempt at this progress.
Bibliography


[ACLM09] Josep Argelich, Alba Cabiscol, Inês Lynce, and Felip Manyà. Sequential Encodings from Max-CSP into Partial Max-SAT. In Kullmann [Kul09], pages 161–166. (Cited on pages 37 and 61.)


ACM Comput. Surv., 38(4), 2006. (Cited on pages 31, 74, 139 and 144.)


[BM] Ramón Béjar and Felip Manyà. Solving the Round Robin Problem Using Propositional Logic. (Cited on page 139.)


[CGLR96] James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and

[Cha04] Gregory Chaitin. Register Allocation and Spilling via Graph Color-

uka.de/sat-race-2010/descriptions/solver7.pdf. [Online; accessed
24-October-2013]. (Cited on page 115.)

[Che10] Jing-Chao Chen. A New SAT Encoding of the At-Most-One Con-
straint. In *Proc. of the Tenth Int. Workshop of Constraint Modelling
and Reformulation*, 2010. (Cited on pages 61, 62, 64, 65, 72 and
115.)

[CKL04] Charles J. Colbourn, Torleiv Kløve, and Alan C. H. Ling. Per-
mutation Arrays for Powerline Communication and Mutually Or-
thogonal Latin Squares. *IEEE Transactions on Information Theory*,

Up Constraint Propagation By Redundant Modeling. In Eugene C.
Freuder, editor, *CP*, volume 1118 of *Lecture Notes in Computer Sci-

[cola] DIMACS Graphs: Benchmark Instances and Best Upper Bounds.
http://www.info.univ-angers.fr/pub/porumbel/graphs/. [On-
line; accessed 24-April-2014]. (Cited on page 138.)

[colb] Graph Coloring Instances. http://mat.gsia.cmu.edu/COLOR/
instances.html. [Online; accessed 24-April-2014]. (Cited on
page 110.)

pages 16 and 129.)

In Michael A. Harrison, Ranar B. Banerji, and Jeffrey D. Ullman,
and 25.)

challenge2013/results2013.html. [Online; accessed 24-April-
2014]. (Cited on page 126.)


Bibliography


pages 72–73. University of Helsinki, Helsinki, Finland, 2013. (Cited on pages 29 and 79.)


[Pre09] Steven David Prestwich. CNF Encodings. In Biere et al. [BHvMW09b], pages 75–97. (Cited on pages 3, 31, 63, 122, 124 and 131.)

[PTS07] Duc Nghia Pham, John Thornton, and Abdul Sattar. Building Structure into Local Search for SAT. In Veloso [Vel07b], pages 2359–2364. (Cited on page 26.)


[Rin09] Jussi Rintanen. Planning and SAT. In Biere et al. [BHvMW09b], pages 483–504. (Cited on page 1.)


Erklärungen zur Eröffnung des Promotionsverfahrens

1. Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

2. Bei der Auswahl und Auswertung des Materials sowie bei der Herstellung des Manuskripts habe ich Unterstützungsleistungen von folgenden Personen erhalten: ... 


4. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und ist auch noch nicht veröffentlicht worden.

5. Ich bestätige, dass ich die geltende Promotionsordnung der Fakultät Informatik der Technischen Universität Dresden anerkenne.

Dresden, 12.12.2014

Unterschrift des Doktoranden