REFORMULATION OF CONSTRAINT MODELS INTO SMT

Miquel Palahí i Sitges

PhD Thesis

Reformulation of Constraint Models into SMT

Author:
Miquel Palahí Sitges

2015

Programa de Doctorat en Tecnologia

Supervisor:
Dr. Miquel Bofill Arasa
Dr. Mateu Villaret Auselle

Memòria presentada per optar al títol de doctor per la Universitat de Girona

Abstract

There exists a wide range of techniques to solve Constraint Satisfaction Problems (CSP). Recently reformulation into other formalisms with specific solvers like propositional satisfiability (SAT), mixed linear integer programming (MILP), etc., has shown to be effective. In this thesis we focus on reformulate CSPs into SAT Modulo Theories (SMT). SMT is an extension of SAT where the literals appearing in the formulas are not restricted to contain only propositional variables, instead they can have predicates from other theories, e.g., linear integer arithmetic.

We present two systems developed to reformulate CSPs into SMT. The first one, called `fzn2smt`, reads instances written in FlatZinc, a low-level language resulting from the reformulation from MiniZinc. These FlatZinc instances are reformulated into SMT instances which then are solved using an external SMT solver. Some CSPs also require to optimize an objective function, known as constraint optimization problems (COP). Since most SMT solvers do not support optimization, `fzn2smt` provides three algorithms for solving COPs: a linear search, a binary search and an hybrid search, which is a combination of the binary and the linear searches. We have used `fzn2smt` to find which SMT solver is the best one solving a set of MiniZinc problems, that is Yices 2. With the same set of problems, we show that `fzn2smt` using Yices 2 is competitive with several FlatZinc solvers. Finally we study the relation between the Boolean structure of the SMT instances and the performance of `fzn2smt` compared with the FlatZinc solvers.

The second system presented is called `WSimply`. It reads CSP, COP and weighted CSP (WCSP) instances written in its own high-level declarative language. In `WSimply` language the user can post soft constraints intensionally and, in addition, the user can also impose meta-constraints over them. Although in the initial version of `WSimply` the input instances were only reformulated into SMT, we extended it to allow the user to also reformulate them into pseudo-Boolean (PB) and linear programming (LP) formats. By default `WSimply` uses the Yices 1 SMT solver through API. This allows us to implement incremental algorithms for solving COPs and WCSPs, either by using a binary search over the objective function or by reformulating the problem into a weighted SMT instance. WSMT instances can be solved using the Yices default algorithm (which is a non-exact algorithm) or using our SMT version of the weighted MaxSAT algorithm WPM1. We show some experimental results of the usefulness of the meta-constraints, for instance, guaranteeing the fairness on the soft constraint violations. We also show some performance comparisons between the `WSimply` solving methods using SMT and PB.

We also present an incremental optimization algorithm based on using Binary Decision Diagrams (BDD) for WCSPs. The idea is to bound the WCSP objective function using SAT clauses instead of using an arithmetic expression. These SAT clauses are generated from a shared BDD, which allows the optimization algorithm to maintain a compact representation of

all tested bounds. We show that the performance of this new method outperforms the previous `WSimply` SMT methods. We also show that this new method is competitive with state-of-the-art COP, WCSP and LP solvers. We also generalize the method to use Multivalued Decision Diagrams (MDD) for representing linear integer arithmetic objective functions, and an special type of objective functions where we can use a weighted MDD (WMDD). We present some results on the performance of the new WMDD method on solving the nurse rostering problem.

We also show that SMT solvers are competitive with state-of-the-art solving methods for the scheduling problem of Resource Constrained Project Scheduling Problem (RCPSP). We also show the usefulness of `WSimply` when encoding problems with soft constraints and meta constraints. In particular, this is done with the Business-to-business (B2B) meetings scheduling problem, where apart from obtaining a very competitive results we also provide an elegant and natural encoding.

Finally, we report and analyse some anomalous behaviours of SMT solvers found when modelling finite domains and solving the Pedigree Reconstruction Problem (PRP).

Resum

Existeix una gran varietat de tècniques per a resoldre problemes de satisfacció de restriccions (CSP). Recentment s'ha demostrat que una manera efectiva de fer-ho és reformulant a altres formalismes amb resoledors específics com satisfactibilitat de lògica proposicional (SAT), programació entera lineal mixta (MILP), etc. En aquesta tesi ens centrem en la reformulació de CSPs a SAT Mòdul Teories (SMT). SMT és una extensió de SAT on els literals que apareixen a la fórmula no estan limitats a contenir només variables Booleanes, sinó que poden tenir-hi predicats d'altres teories, e.g., aritmètica lineal entera.

Presentem dos sistemes desenvolupats per reformular CSPs a SMT. El primer, anomenat `fzn2smt`, llegeix instàncies escrites en FlatZinc, que és un llenguatge de baix nivell resultant de la reformulació des de MiniZinc. Aquestes instàncies FlatZinc es reformulen a instàncies SMT que són resoltes posteriorment per un resoledor SMT extern. Alguns CSPs també tenen una funció objectiu que s'ha d'optimitzar, es coneixen com problemes d'optimització de restriccions (COP). Donat que molts resoledors SMT no suporten optimització, `fzn2smt` disposa de tres algorismes per resoldre COPs: un de cerca lineal, un de cerca binària i un de cerca híbrida, que és la combinació de les dues anteriors. Hem utilitzat `fzn2smt` per a trobar quin és el millor resoledor SMT al resoldre un conjunt de problemes de MiniZinc, que ha resultat ser el Yices 2. Amb el mateix conjunt de problemes, mostrem que `fzn2smt` amb Yices 2 és competitiu respecte a altres resoledors de FlatZinc. Finalment hem estudiat la relació entre l'estructura Booleana de les instàncies SMT i l'eficiència de `fzn2smt` comparada amb els resoledors de FlatZinc.

El segon sistema que presentem es diu `WSimply`. Aquest, llegeix instàncies CSP, COP i CSP amb pesos (WCSP) escrites en el seu propi llenguatge declaratiu d'alt nivell. En el llenguatge de `WSimply` l'usuari pot imposar restriccions toves intencionalment i, a més a més, també pot imposar meta restriccions sobre aquestes. Encara que en la versió inicial de `WSimply` les instàncies d'entrada només es refomulaven a SMT, l'hem estès perquè també permeti a l'usuari reformular-les als formats pseudo-Booleà (PB) i de programació lineal (LP). Per defecte `WSimply` utilitza el resoledor SMT Yices 1 mitjançant API. Això ens permet implementar algorismes incrementals per resoldre COPs i WCSPs, utilitzant, o bé una cerca binària sobre la funció objectiu, o bé reformulant el problema a una instància SMT amb pesos. Les instàncies SMT amb pesos es poden resoldre mitjançant l'algorisme per defecte de Yices (que és un algorisme no exacte) o utilitzant la nostra versió per a SMT de l'algorisme WPM1 procedent de MaxSAT. En els resultats exprimentals mostrem la utilitat de les meta restriccions, per exemple, garantint una violació justa de les restriccions toves. També realitzem diverses comparacions d'eficiència entre els diversos mètodes de resolució de `WSimply` utilizant SMT i PB.

A més a més, presentem un algorisme d'optimització incremental basat en utilitzar diagrames de decisió binaris (BDD) per a resoldre WCSPs. Es tracta d'acotar la funció ob-

jectiu del WCSP utilitzant clàusules Booleanes enlloc d'una expressió aritmètica. Aquestes clàusules Booleanes les generarem a partir d'un BDD compartit, que permet mantenir a l'algorisme d'optimització una codificació compacta de tots els acotaments provats. Mostrem que l'eficiència d'aquest nou mètode supera clarament la dels anteriors mètodes de `WSimply` utilitzant SMT. També mostrem que el nou mètode és competitiu amb els resoledors COP, WCSP i LP estat de l'art. Després generalitzem el mètode per utilitzar diagrames de decisió multivaluats (MDD) per representar funcions objectiu d'aritmètica lineal entera, i un cas especial de funcions objectiu on podem utilitzar MDDs amb pesos (WMDD). Presentem resultats sobre l'eficiència del nou mètode WMDD per a resoldre el problema de programació d'horaris d'infermeres.

Per altra banda, demostrem que els resoledors SMT són competitius amb els mètodes de resolució estat de l'art del Problema del Projecte de Programació amb Recursos Limitats (RCPSP). També mostrem la utilitat de `WSimply` per codificar problemes amb restriccions toves i meta restriccions. En particular, ho fem amb el problema de programació de trobades Business-to-business (B2B), on apart d'obtenir resultats molt competitius també donem una codificació elegant i natural.

Finalment, analitzem algunes anomalies en el comportament dels resoledors SMT trobades al modelar dominis finits i al resoldre el problema de reconstrucció d'arbres genealògics (PRP).

Resumen

Existe una gran variedad de técnicas para resolver problemas de satisfacción de restricciones (CSP). Recientemente se ha demostrado que una manera efectiva de hacerlo es reformulando a otros formalismos con solucionadores específicos como satisfacibilidad de lógica proposicional (SAT), programación entera lineal mixta (MILP), etc. En esta tesis nos centramos en la reformulación de CSPs a SAT Módulo Teorías (SMT). SMT es una extensión de SAT donde los literales que aparecen en la fórmula no están limitados a contener solo variables Booleanas, sino que pueden tener predicados de otras teorías, e.g., aritmética lineal entera.

Presentamos dos sistemas desarrollados para reformular CSPs a SMT. El primero, llamado `fzn2smt`, lee instancias escritas en FlatZinc, que es un lenguaje de bajo nivel resultante de la reformulación desde MiniZinc. Estas instancias FlatZinc se reformulan en instancias SMT que son resueltas posteriormente por un solucionador SMT externo. Algunos CSPs también tienen una función objetivo que optimizar, se conocen como problemas de optimización de restricciones (COP). Dado que muchos solucionadores SMT no soportan optimización, `fzn2smt` dispone de tres algoritmos parar resolver COPs: uno de búsqueda lineal, uno de búsqueda binaria y uno de búsqueda híbrida, que es la combinación de las dos anteriores. Hemos utilizado `fzn2smt` para encontrar cual es el mejor solucionador SMT al resolver un conjunto de problemas de MiniZinc, que ha resultado ser el Yices 2. Con el mismo conjunto de problemas, mostramos que `fzn2smt` con Yices 2 es competitivo respecto a otros solucionadores de FlatZinc. Finalmente hemos estudiado la relación entre la estructura Booleana de las instancias SMT y la eficiencia de `fzn2smt` comparada con los otros solucionadores FlatZinc.

El segundo sistema que presentamos se llama `WSimply`. Este, lee instancias CSP, COP y CSP con pesos (WCSP) en su propio lenguaje declarativo de alto nivel. En el lenguaje de `WSimply` el usuario puede imponer restricciones blandas intencionalmente y, además, también puede imponer meta restricciones sobre estas. Aunque en la versión inicial de `WSimply` las instancias de entrada sólo se reformulaban a SMT, lo hemos extendido para que también permita al usuario reformularlas a los formatos pseudo-Booleano (PB) y de programación lineal (LP). Por defecto `WSimply` utiliza el solucionador SMT Yices 1 a través de API. Esto nos ha permitido implementar algoritmos incrementales para resolver COPs y WCSPs, utilizando, o bién una búsqueda binaria sobre la función objetivo, o bién reformulando el problema en una instancia SMT con pesos. Las instancias SMT con pesos se pueden resolver mediante el algoritmo por defecto de Yices (que es un algoritmo no exacto) o utilizando nuestra versión para SMT de el algoritmo WPM1 procedente de MaxSAT. En los resultados experimentales mostramos la utilidad de las meta restricciones, por ejemplo, garantizando una violación justa de las restricciones blandas. También realizamos varias comparaciones de eficiencia entre los diversos métodos de resolución de `WSimply` utilizando SMT y PB.

Además presentamos un algoritmo de optimización incremental basado en utilizar diagramas de decisión binarios (BDD) para solucionar WCSPs. Se trata de limitar la función

objetivo del WCSP utilizando cláusulas Booleanas en lugar de una expresión aritmética. Estas cláusulas Booleanas las generaremos a partir de un BDD compartido, que permite mantener al algoritmo de optimización una codificación compacta de todos los límites testeados. Mostramos que la eficiencia de este nuevo método supera claramente a los métodos previos de `WSimply` utilizando SMT. También mostramos que el nuevo método es competitivo con los solucionadores COP, WCSP y LP estado del arte. Luego generalizamos el método para utilizar diagramas de decisión multivaluados (MDD) para representar funciones objetivo de aritmética lineal entera, y un caso especial de funciones objetivo dónde podemos utilizar MDDs con pesos (WMDD). Presentamos resultados sobre la eficiencia del nuevo método WMDD para resolver el problema de programación de horarios de enfermeras.

Por otra parte demostramos que los solucionadores SMT son competitivos con los métodos de resolución estado del arte del Problema del Proyecto de Programación con Recursos Limitados (RCPSP). También mostramos la utilidad de `WSimply` para codificar problemas de restricciones blandas y meta restricciones. En particular, lo hacemos con el problema de programación de encuentros Business-to-business (B2B), donde aparte de obtener resultados muy competitivos también damos una codificación elegante y natural.

Finalmente, analizamos algunas anomalías en el comportamiento de los solucionadores SMT encontradas modelando dominios finitos y resolviendo el problema de reconstrucción de árboles genealógicos (PRP).

Acknowledgments

Thanks!

Contents

Abstract **v**

Resum **vii**

Resumen **ix**

Acknowledgments **xi**

Contents **xvi**

List of Figures **xviii**

List of Tables **xx**

List of Algorithms **xxi**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Objectives . . . 1
1.3 Publications . . . 2
1.4 Outline of the Thesis . . . 4

2 Definitions, Terminology and Notation **5**
2.1 Constraint Satisfaction Problems (CSP) . . . 5
2.1.1 Global constraints . . . 6
2.1.2 Constraint Optimization . . . 6
2.1.3 Weighted CSP . . . 6
2.1.4 Meta-Constraints . . . 7
2.2 Propositional Satisfiability (SAT) . . . 8
2.2.1 (Weighted) MaxSAT . . . 9
2.3 Mixed Integer Linear Programming (MILP) . . . 10
2.3.1 Pseudo-Boolean Constraints . . . 11
2.4 Satisfiability Modulo Theories (SMT) . . . 11
2.4.1 (Weighted) MaxSMT . . . 12

3 CSP Solving Techniques **13**
3.1 Systematic Search . . . 13
3.1.1 Generate and Test . . . 14
3.1.2 Backtracking . . . 14
3.2 Consistency Techniques . . . 15
3.2.1 Node Consistency . . . 16
3.2.2 Arc Consistency . . . 16
3.2.3 Other Consistency Techniques . . . 17
3.3 Constraint Propagation . . . 17
3.3.1 Forward Checking . . . 18
3.3.2 Look Ahead . . . 19
3.3.3 Propagators . . . 19
3.4 Solving methods for Global Constraints . . . 20
3.5 Constraint Optimization . . . 20
3.5.1 Weighted CSP . . . 21
3.6 Solving CSPs by Reformulation . . . 21
3.6.1 Reformulation into SAT . . . 21
3.6.2 Reformulation into MILP . . . 25
3.6.3 Reformulation into SMT . . . 26
3.7 Other CSP Solving Methods . . . 26
3.7.1 Lazy Clause Generation . . . 26
3.7.2 Local search . . . 29
3.7.3 Genetic algorithms . . . 29

4 SAT and Satisfiability Modulo Theories (SMT) **31**
4.1 SAT . . . 31
4.1.1 Satisfiability Algorithms . . . 31
4.1.2 (Weighted) MaxSAT Algorithms . . . 34
4.2 SMT . . . 36
4.2.1 Eager and Lazy SMT Approaches . . . 37
4.2.2 Theories and Logics . . . 39
4.2.3 (Weighted) MaxSMT Algorithms . . . 42

5 A comparison of CP and SMT solvers on standard CSP benchmarks **43**
5.1 MiniZinc and FlatZinc . . . 43
5.2 Architecture of `fzn2smt` . . . 45
5.3 Reformulation . . . 46
5.3.1 Constant and Variable Declarations . . . 47
5.3.2 Constraints . . . 49
5.3.3 Solve Goal . . . 52
5.4 Experimental results . . . 54
5.4.1 Comparison of SMT solvers within `fzn2smt` . . . 54
5.4.2 Comparison of `fzn2smt` with other FlatZinc Solvers . . . 58
5.4.3 FlatZinc Solvers with Global Constraints . . . 60
5.4.4 Impact of the Boolean component of the instances in the `fzn2smt` performance . . . 61
5.5 Conclusions . . . 64

6 An SMT approach to the RCPSP **67**
6.1 The Resource-Constrained Project Scheduling Problem (RCPSP) 68
6.2 RCPSP to SMT 69
6.2.1 Preprocessing 70
6.2.2 Optimization 71
6.3 Encodings 72
6.3.1 Time formulation 73
6.3.2 Task formulation 74
6.3.3 Flow formulation 75
6.4 Experimental results 76
6.5 Conclusions 77

7 `Simply` and `WSimply` **79**
7.1 History 79
7.2 Architecture of the tool 80
7.3 Language description 81
7.3.1 Model File 81
7.3.2 Data file 89
7.4 WCSP reformulation into SMT 89
7.4.1 Meta-constraint reformulation 90
7.4.2 Constraints reformulation 93
7.4.3 WCSP and COP reformulations 94
7.5 WCSP reformulation into PB/LP 96
7.5.1 Meta-constraints reformulation 96
7.5.2 Constraints reformulation 97
7.5.3 WCSP and COP reformulations 98
7.6 The `WSimply` solving module 99
7.7 Experimental results 100
7.7.1 `Simply` first version 100
7.7.2 `WSimply` with meta-constraints 103
7.7.3 `WSimply` (PB) 113
7.8 Conclusions 117

8 BDD-based Incremental Optimization **119**
8.1 Binary Decision Diagrams (BDD) 119
8.2 BDD-based SAT encodings of PB constraints 120
8.2.1 SAT encodings of PB constraints using BDDs 121
8.2.2 SAT encodings of linear integer constraints using MDDs 122
8.3 Solving WCSPs by Incremental Optimization using SROBDDs 123
8.3.1 Incremental optimization algorithm 124
8.4 Experimental results 125
8.4.1 `WSimply` solving methods comparison 127
8.4.2 SBDD-based versus state-of-the-art CSP and WCSP solvers 128
8.4.3 SBDD incrementality 129
8.4.4 The Nurse Rostering Problem (NRP) 131
8.5 Conclusions 132

9 Scheduling of Business-to-business (B2B) meetings **133**
9.1 The B2B problem 134
9.2 Models 136
9.2.1 A WCSP model for the B2BSOP 136
9.2.2 A PB model for the B2BSOP 140
9.3 Experimental results 143
9.4 Conclusion 145

10 Anomalies Modelling Combinatorial Problems using SMT **147**
10.1 Anomalies in Representing Finite Domains 147
10.1.1 Anomaly 1 148
10.1.2 Anomaly 2 149
10.2 Anomalies in representing the PRP 150
10.2.1 Anomaly 3 154
10.2.2 Anomaly 4 155
10.3 Conclusions 156

11 Conclusions and future work **159**
11.1 Conclusions 159
11.2 Future Work 162

Bibliography **163**

List of Figures

3.1 Arc consistent CSP without solution. 17
3.2 Forward checking example for the 4-Queens problem 18
3.3 Look ahead example for the 4-Queens problem 19
3.4 Original architecture of LCG 27
3.5 New architecture of LCG 27

4.1 Application of the PM1 algorithm. 35
4.2 Application of the WPM1 algorithm. 37

5.1 Compiling and solving process of `fzn2smt` using linear search 46
5.2 `fzn2smt` and FlatZinc solvers time plot 60
5.3 `fzn2smt` and Gecode normalized difference with respect to the ratio of Boolean variables 62
5.4 `fzn2smt` and Gecode normalized difference with respect to the ratio of disjunctions 63

6.1 RCPSP example 68

7.1 `WSimply` basic architecture and SMT solving process 81
7.2 Grammar of the `WSimply` model file. 82
7.3 `WSimply` model of the classical n-Queens problem. 83
7.4 `WSimply` model of the classical n-Queens problem using a user defined reference. 85
7.5 `WSimply` data file grammar. 90
7.6 `WSimply` basic architecture and SMT solving process 95
7.7 `WSimply` (SMT) model for the NRP. 105
7.8 `WSimply` homogeneity meta-constraints for the NRP model 106
7.9 `WSimply` model for the SBACP. 110
7.10 `WSimply` homogeneity meta-constraints for the SBACP model 110
7.11 `WSimply` multilevel meta-constraints for the SBACP model 113
7.12 `WSimply` (PB) model for the SBACP. 114
7.13 `maxCost` meta-constraint for the SBACP problem. 115
7.14 `WSimply` (PB) model for the NSP. 116

8.1 ROBDD for the Boolean function $\overline{x_1} + x_1\overline{x_2} + x_1x_2\overline{x_3}$. 120
8.2 Shared ROBDD construction example for objective $2x_1 + 3x_2 + 4x_3 \leq K$ . . 126
8.3 Formula update example for objective $2x_1 + 3x_2 + 4x_3 \leq K$ 126

10.1 Anomaly 1: Yices 1 149

10.2 Anomaly 1: Yices 2 149
10.3 Anomaly 1: MathSat 5 149
10.4 Anomaly 1: Z3 149
10.5 Anomaly 2 150
10.6 Z3 statistics in function of domain size (n) 151
10.7 MathSat 5 scatter-plot for Anomaly 3 155
10.8 Yices 1 scatter-plot for Anomaly 3 155
10.9 Z3 scatter-plot for Anomaly 4 satisfiable case 1 157
10.10 Z3 scatter-plot for Anomaly 4 satisfiable case 2 157
10.11 Z3 scatter-plot for Anomaly 4 unsatisfiable case 1 157
10.12 Z3 scatter-plot for Anomaly 4 unsatisfiable case 2 157

List of Tables

5.1 `fzn2smt` SMT solvers comparison . . . 55
5.2 `fzn2smt` variable array translations comparison . . . 56
5.3 `fzn2smt` optimization algorithms comparison . . . 58
5.4 `fzn2smt` and FlatZinc solvers comparison . . . 59
5.5 `fzn2smt` and FlatZinc solvers comparison using global constraints . . . 61
5.6 Paired t-test for `fzn2smt` and Gecode . . . 64
5.7 Paired t-test for `fzn2smt` and lazy_fd . . . 64

6.1 Comparison of the three RCPSP encodings using SMT . . . 76
6.2 RCPSP comparison of `rcp2smt` and state-of-the-art solvers . . . 76

7.1 LP (PB) reformulation of the logical constraints. . . . 97
7.2 `Simply` and similar tools comparison . . . 101
7.3 `Simply` and FlatZinc solvers comparison . . . 103
7.4 `Simply` and `fzn2smt` comparison . . . 104
7.5 `WSimply` results of 5113 instances from the N25 set . . . 106
7.6 `WSimply` results of the NRP with the `homogeneousAbsoluteWeight` meta-constraint (factor 5) . . . 107
7.7 `WSimply` results of the NRP with the `homogeneousAbsoluteWeight` meta-constraint (factor 10) . . . 107
7.8 `WSimply` results of the NRP with the `homogeneousPercentWeight` meta-constraint (factor 6) . . . 108
7.9 `WSimply` results of the NRP with the `homogeneousPercentWeight` meta-constraint (factor 11) . . . 108
7.10 `WSimply` results of the SBACP with and without homogeneity. . . . 111
7.11 `WSimply` results of the SBACP with homogeneity and the `multiLevel` meta-constraint . . . 112
7.12 SBACP prerequisites example for the PB encoding. . . . 115
7.13 `WSimply` (PB) and `WSimply` (SMT) comparison for the SBACP . . . 116
7.14 `WSimply` (PB) and `WSimply` (SMT) comparison for the NRP . . . 117

8.1 WCSPs description: number of variables and weights . . . 127
8.2 `WSimply` +SBDD method and previous `WSimply` (SMT) methods comparison 128
8.3 `WSimply` +SBDD method and state-of-the-art COP/WCSPs comparison . . . 129
8.4 `WSimply` comparison of SBDD and LIA, with and without learning . . . 130
8.5 `WSimply` +WMDD method and previous `WSimply` (SMT) methods comparison 132

9.1 B2B solving methods comparison 144
9.2 B2B quality comparison 145

10.1 SMT solver statistics for Anomaly 1 150
10.2 Basic SMT(QF_LIA) model of the pedigree reconstruction problem 153
10.3 MathSat 5 solving times for Anomaly 3 154
10.4 Yices 1 solving times for Anomaly 3 154
10.5 Z3 solving times for Anomaly 4 156

List of Algorithms

1 Conflict-Driven Clause-Learning algorithm 32
2 PM1 Algorithm 35
3 WPM1 Algorithm 36
4 Bool+T Algorithm 38
5 `fzn2smt` Minimization Algorithm 52
6 BDD-based incremental optimization algorithm 124
7 WMDD mapping function example 131

Chapter 1

Introduction

1.1 Motivation

In our everyday life we deal with problems that are easy to understand, but finding a solution for them can be really hard. This is because the number of combinations of values of their variables grows exponentially with respect to the problem size. This kind of problems are known as combinatorial problems, and a few examples, among others, are timetabling problems, sport scheduling problems or solving puzzles like Sudoku.

In general, these problems can be represented as Constraint Satisfaction Problems (CSP), where the problem is defined by a set of constraints. There exists a wide range of techniques for dealing with CSPs, such as modelling the problem using a high-level CSP language, like MiniZinc, for which there exist several constraint solvers, or modelling the problem as a propositional satisfiability formula and solve this formula using a SAT solver, or even directly implementing an ad-hoc program to solve the problem using a common programming language like C, C++, Java, etc.

During the last two decades there have been big improvements and extensions in SAT solvers, making them very competitive tools for solving CSPs. In this thesis we focus on one of these extensions: a generalization of the formulas where instead of having only propositional variables they also can have predicates from other theories, e.g., linear integer arithmetic. This generalization is known as SAT Modulo Theories (SMT) and has become a completely new field of research.

Some papers have shown that SMT technology can be competitive for solving certain combinatorial problems (see, e.g., [NO06] for an application of an SMT solver on the Celar optimization problem, being competitive with the best weighted CSP solver with its best heuristic on that problem). In this thesis we want to go one step further showing that SMT solvers can be competitive for generic CSP solving.

1.2 Objectives

- The first objective of this thesis is to develop a competitive tool using SMT technology to solve CSPs (note that, at the begining of this thesis, none of the existing CSP languages has a tool that translates into SMT). The first step to achieve this objective

is to decide between using an existing high-level CSP language or designing a new one. Since we are interested in having a complete control on the language construction and their reformulation we will define a new language. The next step will be to implement a compiler that reads instances written in the new CSP language and reformulates them into SMT instances to be solved using an SMT solver. Finally, we also want to solve Constraint Optimization Problems (COP), because it is quite common to find CSPs where we have to optimize an objective function. But since SMT solvers are only decision procedures, we will need to implement at least a solving module for optimization, for instance by iteratively calling the SMT solver with successively tighter bounds.

- The second objective of this thesis consists in extending the system to deal with weighted CSPs (WCSP). Although WCSPs can be easily translated into COPs, in our case SMT optimization problems, we think that we can take more profit of the problem structure adapting Weighted MaxSAT algorithms. Therefore, first of all we need to extend the language of our new system to represent intentional weighted constraints, second we need to extend the system to allow to translate the WCSPs into COPs or Weighted MaxSMT instances (WSMT), and finally we need to extend the solving module to deal with WSMT, either by using one of the few SMT solvers which supports WSMT or implementing WSMT algorithms based on weigthed MaxSAT algorithms.

- Another objective of the thesis is the addition of meta-constraints (i.e. constraints over constraints) in the language, in order to allow the modelling of complex rules on violations, that are often present in real-life overconstrained problems, such as fairness, degree of violation, etc.

Note that the creation of new theories for SMT in the context of Constraint Programming, as well as performing optimization of the objective function "inside" the SMT solver, is out of the scope of this thesis.

1.3 Publications

The publications arising from this thesis are the following:

- Journals
  - Carlos Ansótegui, Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *Solving Weighted CSPs with Meta-Constraints by Reformulation into Satisfiability Modulo Theories*. Constraints journal, vol. 18, number 2, pages 236-268, 2013.
  - Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *Solving constraint satisfaction problems with SAT modulo theories*. Constraints journal, vol. 17, number 3, pages 273-303, 2012.
- Conferences
  - Miquel Bofill, Joan Espasa, Marc Garcia, Miquel Palahí, Josep Suy and Mateu Villaret, *Scheduling B2B Meetings*. Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP 2014), Lyon, France, pages 781–796, 2014

- Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *Solving Intensional Weighted CSPs by incremental optimization with BDDs*. Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP 2014), Lyon, France, pages 207–223, 2014
- Miquel Bofill, Joan Espasa, Miquel Palahí and Mateu Villaret, *An extension to* `Simply` *for solving Weighted Constraint Satisfaction Problems with Pseudo-Boolean Constraints*. Proceedings of the XII Spanish Conference on Programming and Computer Languages (PROLE), Almería, Spain, pages 141–155, 2012.
- Carlos Ansótegui, Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *Satisfiability modulo theories: An efficient approach for the resource-constrained project scheduling problem.*
  Ninth Symposium on Abstraction Reformulation and Approximation (SARA 2011), Cardona, Spain, pages 2-9, AAAI, 2011.
- Miquel Bofill, Miquel Palahí and Mateu Villaret, *A System for CSP Solving through Satisfiability Modulo Theories*. Proceedings of the IX Spanish Conference on Programming and Computer Languages (PROLE), Donostia, Spain, pages 303–312, 2009.

- Other publications

  - Alan M. Frisch and Miquel Palahí, *Anomalies in SMT Solving: Dificulties in Modelling Combinatorial Problems*. Proceedings of the 13th International Workshop on Constraint Modelling and Reformulation (ModRef), co-located with CP 2014, Lyon, France, pages 97–110, 2014.
  - Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *Boosting weighted CSP resolution with BDDs*. Proceedings of the CP Doctoral Program, co-located with CP 2013, Uppsala, Sweden, pages 121–126, 2013
  - Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *Boosting Weighted CSP Resolution with Shared BDDs*. Proceedings of the 12th International Workshop on Constraint Modelling and Reformulation (ModRef), co-located with CP 2013, Uppsala, Sweden, pages 57–73, 2013
  - Carlos Ansótegui, Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *W-MiniZinc: A Proposal for Modeling Weighted CSPs with MiniZinc*. Proceedings of the first MiniZinc workshop (MZN), co-located with CP 2011, Perugia, Italy, 2011
  - Carlos Ansótegui, Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *A Proposal for Solving Weighted CSPs with SMT*. Proceedings of the 10th International Workshop on Constraint Modelling and Reformulation (ModRef), co-located with CP 2011, Perugia, Italy, pages 5-19, 2011
  - Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *`Simply`: a Compiler from a CSP Modeling Language to the SMT-LIB Format*. Proceedings of the 8th International Workshop on Constraint Modelling and Reformulation (ModRef), co-located with CP 2009, Lisboa, Portugal, pages 30–44, 2009.

1.4 Outline of the Thesis

Chapters 2, 3 and 4 present the required background. In Chapter 2 we present the formal definitions and the basic concepts required to follow this thesis. In Chapter 3 we present different common ways to solve CSPs. Chapter 4 is devoted to SAT and SMT.

Chapters 5, 7 and 8 contain the main work of this thesis: the development of the system to model (W)CSPs and solve them using SMT called (`W`)`Simply`, as well as the development of the alternative `fzn2smt` tool. In Chapters 6, 9 and 10 we use SMT for solving specific problems either by direct reformulation into SMT or using `WSimply`.

The final conclusions of this thesis are presented in Chapter 11.

Chapter 2

Definitions, Terminology and Notation

In this chapter we introduce several relevant definitions and concepts which will be used during this thesis (for further deailts see [BHvMW09, RBW06]).

First of all we formally define what are CSPs, i.e., the problems we want to solve, as well as some of their variants.

Then we define propositional Satisfiability and Mixed Integer Linear Programming, two relevant methods that have been successful in solving CSPs by reformulation. Details of CSP reformulation into these methods are explained in Section 3.6.

Finally we define SAT modulo theories, which is the focus of this thesis.

2.1 Constraint Satisfaction Problems (CSP)

Definition 2.1.1 *A* constraint satisfaction problem (CSP) *is defined as a triple* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$*, where* $\mathcal{X} = \{X_1, \ldots, X_n\}$ *is a set of variables,* $\mathcal{D} = \{D(X_1), \ldots, D(X_n)\}$ *is a set of domains containing the values that each variable may take, and* $\mathcal{C} = \{C_1, \ldots, C_m\}$ *is a set of constraints. Each constraint* $C_i = \langle S_i, R_i \rangle$ *is defined as a relation* R_i *over a subset of variables* $S_i = \{X_{i_1}, \ldots, X_{i_k}\}$*, called the* scope of constraint C_i*. A relation* R_i *may be represented* intentionally*, in terms of an expression that states the relationship that must hold amongst the assignments to the variables it constrains, or it may be represented* extensionally*, as a subset of the Cartesian product* $D(X_{i_1}) \times \cdots \times D(X_{i_k})$ *(tuples) which represents the allowed assignments (*good tuples*) or the disallowed assignments (*no-good tuples*).*

Definition 2.1.2 *A* partial assignment *v for a CSP* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$*, being* $\mathcal{Y}$ *a subset of* $\mathcal{X}$*, is a mapping that assigns to every variable* $X_i \in \mathcal{Y}$ *an element* $v(X_i) \in D(X_i)$*. The set* $\mathcal{Y}$ *is the domain of v, denoted domain(v); when* $\mathcal{Y} = \mathcal{X}$*, v is simply an* assignment*. A (partial) assignment v satisfies a constraint* $\langle \{X_{i_1}, \ldots, X_{i_k}\}, R_i \rangle \in \mathcal{C}$ *if and only if* R_i *holds under the assignment v (when* R_i *is intentional) or* $\langle v(X_{i_1}), \ldots, v(X_{i_k}) \rangle \in R_i$ *(when* R_i *is extensional)*

A solution to a CSP *is an assignment such that every constraint is satisfied.*

A (partial) assignment v is consistent *iff satisfies all the constraints* $\langle S_i, R_i \rangle \in \mathcal{C}$ *such that* $S_i \subseteq$ *domain(v). Otherwise we say that it is inconsistent or there is a conflict.*

Definition 2.1.3 *A* label (X, val) *is a variable-value pair which represents the assignment of value* val *to variable* X*.*

Example 1 *Consider* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ *where* $\mathcal{X} = \{X_1, X_2, X_3\}, D(X_1) = D(X_2) = D(X_3) = \{1, 2, 3, 4\}$ *and* $\mathcal{C} = \{(X_3 < X_2), (X_2 < X_1)\}$*. A possible solution of this CSP instance is the assignment* $\{(X_1, 3), (X_2, 2), (X_3, 1)\}$*.*

2.1.1 Global constraints

One interesting property of the constraints, is that we can remove all the values of the variables involved in one constraint that are not consistent with such constraint. For instance, given the constraint $x + 3 = y$ where $D(x) = \{1, 3, 4, 5\}$ and $D(y) = \{4, 5, 8\}$, we can reduce the domain of variables x and y to $D(x) = \{1, 5\}$ and $D(y) = \{4, 8\}$.

In order to take advantage of this property, there exists a special type of constraints, known as global constraints, which define a relation between a non-fixed number of variables and, in general, are equivalent to a conjunction of simpler constraints.

For example, the *alldifferent*$(x_1, \ldots, x_n)$ constraint, specifies that the values assigned to the variables must be pairwise distinct; which is equivalent to define a conjunction of $\neq$ constraints for each pair of variables.

2.1.2 Constraint Optimization

In some problems, in addition to finding a solution that satisfies all the constraints, appears the necessity to find a good solution and, if possible, find the best solution (optimal solution). In order to decide which solution is better, these problems have what is known as *objective function*, i.e., an optimization criteria. This kind of problems are known as Constraint Optimization Problem (COP). In some cases finding the optimal solution is very expensive, and we will content ourselves with a suboptimal solution.

Definition 2.1.4 *A* Constraint Optimization Problem (COP) *is a tuple* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{O} \rangle$*, where* $\mathcal{X}$*,* $\mathcal{D}$ *and* $\mathcal{C}$ *represent, as in the case of a CSP, the variables, domains and constraints, respectively, and* $\mathcal{O}$ *is an objective function mapping every complete assignment to an integer (or real) value. An optimal solution of a COP is an assignment that minimizes (or maximizes)* $\mathcal{O}$ *and satisfies all the constraints.*

2.1.3 Weighted CSP

In other cases, there are so many constraints in the problem that makes impossible to find a solution that satisfies all the constraints. This kind of problems are known as over-constrained problems. The most common approach to solve over-constrained problems is to relax the problem allowing to violate (non satisfy) some of the constraints while trying to maximize the number of satisfied constraints. This problem is known as MaxCSP and is defined as follows:

Definition 2.1.5 *The* MaxCSP *problem for a CSP is the problem of finding an assignment that minimizes (maximizes) the number of violated (satisfied) constraints.*

Since not all the constraints have the same relevance in the problem there is another formalism known as Weighted CSP (WCSP) which generalises MaxCSP, where each constraint has an associated weight representing the violation cost of the constraint. We define WCSP as follows:

Definition 2.1.6 *A* weighted CSP (WCSP) *is a triple $\langle X, D, C\rangle$, where X and D are variables and domains, respectively, as in a CSP. A constraint C_i is now defined as a pair $\langle S_i, f_i\rangle$, where $S_i = \{X_{i_1}, \ldots, X_{i_k}\}$ is the constraint scope and $f_i : D(X_{i_1}) \times \cdots \times D(X_{i_k}) \to \mathbb{N} \cup \{\infty\}$ is a* cost (weight) function *that maps tuples to its associated weight (a natural number or infinity). We call those constraints whose associated cost is infinity* hard*, otherwise* soft*. The cost (weight) of a constraint C_i induced by an assignment v in which the variables of $S_i = \{X_{i_1}, \ldots, X_{i_k}\}$ takes values $b_{i_1}, \ldots, b_{i_k}$ is $f_i(b_{i_1}, \ldots, b_{i_k})$.*

A solution to a WCSP is an assignment in which the sum of the costs of the constraints is minimal.

Note that in the particular case when all penalties are equal to one, WCSP is identical to MaxCSP.

In the literature a WCSP is also referred to as a constraint optimization problem, which is a regular CSP whose constraints are weighted and the goal is to find a solution while minimizing the cost of the falsified constraints. Nevertheless, in this thesis we refer to COPs in a operations research-like style according to Definition 2.1.4. This allows us to stress the difference between instances explicitly containing soft constraints and instances just containing an optimization variable.

Example 2 *Consider a WCSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}\rangle$ where $\mathcal{X} = \{X_1, X_2, X_3\}, D(X_1) = D(X_2) = D(X_3) = \{1, 2, 3, 4\}$ and $\mathcal{C} = \{(X_3 < X_2, 2), (X_2 < X_1, 1), (X_1 < X_3, 3)\}$. The assignment $\{(X_1, 1), (X_2, 3), (X_3, 2)\}$ is a possible solution to this WCSP. This assignment satisfies the constraints C_1 and C_3, and does not satisfy the constraint C_2. This assignment has cost* 1*, and it is the optimal solution.*

It is very common to talk about extensional WCSPs, where restrictions and weights are defined extensionally (with tuples). For instance, in [BMR+99, RL09] there is another definition for (extensional) WCSP. But in this thesis we will be talking about WCSP defined intensionally if it is not otherwise specified.

2.1.4 Meta-Constraints

In order to provide a higher level of abstraction in the modelling of over-constrained problems, in [PRB00] several meta-constraints are proposed. By meta-constraint we refer to a constraint on constraints, definition first introduced in [Ber93]. Meta-constraints allow us to go one step further in the specification of the preferences on the soft constraint violations. For example, in the well-known Nurse Rostering Problem (NRP) [BCBL04], in order to obtain "fair" solutions with respect to the preferences of the nurses, it could be useful to impose certain homogeneity on the amount of violation of the different sets of preference constraints of the different nurses.

2.2 Propositional Satisfiability (SAT)

Although propositional satisfiability (SAT) is the first known NP-Complete problem, the improvements in SAT technology during the last two decades have made it viable for CSP solving.

Definition 2.2.1 *Let $P = \{x_1, \ldots, x_n\}$ be a finite set of propositional symbols (Boolean variables). Propositional symbols $x_i \in P$ may take values false or true. A* literal *l_i is a propositional symbol x_i or its negation $\neg x_i$. The complementary of a literal l, denoted by $\neg l$, is x if $l = \neg x$ and is $\neg x$ if $l = x$.*

A clause *(C) is a disjunction of literals $l_1 \vee \ldots \vee l_n$ and a CNF* formula *ϕ is a conjunction of clauses $C_1 \wedge \cdots \wedge C_m$. A clause is often presented as a set of literals, and a CNF formula as a set of clauses.*

Definition 2.2.2 *The* size of a clause *C, denoted by $|C|$, is the total number of literal occurrences in the clause. A clause with one literal is called unit clause, a clause with two literals is called binary clause and a clause with more than two literals is called n-ary clause. The special case of a clause with zero literals is the empty clause and its truth valuation is always false (in this thesis is denoted by $\square$). The size of a CNF formula ϕ, denoted by $|\phi|$, is the sum of the sizes of all its clauses.*

Definition 2.2.3 *A (partial) truth* assignment *M is a set of literals such that $\{l, \neg l\} \not\subseteq M$ for any literal l and $\neg l$ built from a set of propositional symbols of P. A literal l is true in M if $l \in M$, it is false in M if $\neg l \in M$ and it is undefined in M otherwise. M is total over P if no literal build with the propositional symbols of P is undefined in M. A clause C is true in M if at least one of its literals is in M (hence, the empty clause cannot be satisfied), it is false in M if all its literals are false in M, and it is undefined in M otherwise.*

Definition 2.2.4 *A formula ϕ is* satisfied *by M (ϕ is true in M), denoted by $M \models \phi$, if all its clauses are true in M. In that case, M is called a* model *of ϕ. If ϕ has no models then it is called* unsatisfiable*, otherwise is called* satisfiable*.*

Definition 2.2.5 *The* Satisfiability Problem (SAT) *for a CNF formula ϕ is the problem of deciding if there exists a truth assignment M that satisfies all the clauses of ϕ.*

Definition 2.2.6 *If ϕ and ϕ' are formulas, we write $\phi \models \phi'$ if ϕ' is true in all models of ϕ. Then we say that ϕ' is* entailed *by ϕ, or is* logical consequence *of ϕ. If $\phi \models \phi'$ and $\phi' \models \phi$, we say that ϕ and ϕ' are* logically equivalent*.*

Example 3 *Let us consider a CNF formula ϕ having three clauses c_1, c_2 and c_3:*

$$c_1 : x_1 \vee \neg x_2$$
$$c_2 : x_1 \vee x_3$$
$$c_3 : \neg x_1 \vee x_2 \vee x_3$$

Under the partial truth assignment $\{\neg x_1, \neg x_2\}$, clauses c_1 and c_3 are satisfied and clause c_2 is undefined (note that clause c_2 becomes unit, since literal x_1 can be deleted from it). Therefore, the CNF formula ϕ is undefined under this assignment.

Suppose now that this assignment is completed by adding the literal $\neg x_3$. Then, clause c_2 becomes unsatisfied. Finally, if we consider the assignment $\{\neg x_1, \neg x_2, x_3\}$, all the clauses are satisfied.

2.2.1 (Weighted) MaxSAT

As well as in the case of CSP, in some cases when a SAT formula is unsatisfiable we want to obtain the "best possible" truth assignment. This can be done using one of the optimization variants of the SAT problem: MaxSAT, Partial MaxSAT, Weighted MaxSAT and Partial Weighted MaxSAT. Next we define them and explain the differences between them.

(Partial) MaxSAT

Definition 2.2.7 *Given a CNF formula, the* maximum satisfiability (MaxSAT) problem *is the problem of finding a solution that maximizes the number of satisfied clauses.*

Definition 2.2.8 *Given a CNF formula with a set of clauses that can be violated (soft clauses) and a set of clauses that must be satisfied (hard clauses), the* Partial MaxSAT problem *is the problem of finding a solution that maximizes the number of satisfied soft clauses while ensuring the satisfaction of all the hard clauses.*

Note that the (partial) MaxSAT problem is equivalent to minimizing the number of falsified (soft) clauses of a CNF formula ϕ, which is the *cost* of ϕ.

In SAT, a CNF formula is considered to be a set of clauses, while in (partial) MaxSAT, a CNF formula is considered to be a multiset of clauses, because repeated clauses cannot be collapsed into a unique clause. For instance, in $\{x_1, \neg x_1, \neg x_1, x_1 \vee x_2, \neg x_2\}$, where a clause is repeated, there is a minimum of two unsatisfied clauses. If we consider the set $\{x_1, \neg x_1, x_1 \vee x_2, \neg x_2\}$, where repeated clauses are collapsed, then there is a minimum of one unsatisfied clause.

Example 4 *Let us consider a Partial MaxSAT instance ϕ having the following clauses:*

$$
\begin{aligned}
&c_1 : [x_1 \vee x_2] \\
&c_2 : [\neg x_1] \\
&c_3 : [\neg x_1 \vee \neg x_2] \\
&c_4 : (\neg x_2 \vee x_3) \\
&c_5 : (x_1 \vee \neg x_2) \\
&c_6 : (\neg x_3) \\
&c_7 : (x_1 \vee \neg x_2 \vee \neg x_3)
\end{aligned}
$$

Hard clauses are represented between square brackets, and soft clauses are represented between round brackets.

An optimal solution for ϕ is the assignment $\{\neg x_1, x_2, \neg x_3\}$, which satisfies all the hard clauses and maximizes the number of satisfied soft clauses. The number of falsified soft clauses (cost) is 2, and the satisfied soft clauses are c_6 and c_7.

Weighted (Partial) MaxSAT

Definition 2.2.9 *A Weighted CNF clause is a pair (C, w), where C is a clause and w is its weight. The weight w can be a natural number or infinity and its meaning is the penalty (cost) for falsifying the clause C. A clause is hard if its corresponding weight is infinity, and*

soft otherwise. When there are no hard clauses we speak of Weighted MaxSAT, and we speak of Weighted Partial MaxSAT otherwise. A Weighted CNF formula is a multiset of weighted clauses.

The Weighted MaxSAT problem *consists in finding a truth assignment such that the sum of the weights of the satisfied clauses is maximized or, equivalently, the total weight of the falsified clauses is minimized.*

Example 5 *Let us consider a Weighted MaxSAT instance having the following clauses:*

$$c_1 : (x_1 \vee x_2, 4)$$
$$c_2 : (\neg x_1, 3)$$
$$c_3 : (\neg x_1 \vee \neg x_2, 5)$$
$$c_4 : (\neg x_2 \vee x_3, 3)$$
$$c_5 : (x_1 \vee \neg x_2, 2)$$
$$c_6 : (\neg x_3, 1)$$

An optimal solution for this formula is the assignment $\{\neg x_1, x_2, x_3\}$*. The sum of weights of satisfied clauses is* 15*, and the sum of weights of falsified clauses is* 3*. This assignment falsifies the clauses* c_5 *and* c_6*.*

We remark that the MaxSAT problem can also be defined as Weighted MaxSAT restricted to formulas whose clauses have weight 1, and as Partial MaxSAT in the case that all the clauses are declared to be soft. The Partial MaxSAT problem is the Weighted Partial MaxSAT problem when the weights of the soft clauses are equal.

The notion of equivalence with respect to instances has some subtlety that we want also to remark. In SAT, two formulas are equivalent if they are satisfied by the same set of assignments. In MaxSAT, two formulas are considered to be equivalent if both have the same number of unsatisfied clauses for every assignment. In Weighted MaxSAT, two formulas are considered to be equivalent if the sum of the weights of unsatisfied clauses coincides for every assignment.

2.3 Mixed Integer Linear Programming (MILP)

An alternative method for CSP solving is based on Mixed Integer Linear Programming (MILP), which is defined as:

$$min\{c^T x : A\vec{x} \leq b, l \leq \vec{x} \leq u, \vec{x} \in \mathbb{R}^n, x_j \in \mathbb{Z}\ \forall j \in I\}$$

where $A \in \mathbb{Q}^{m*n}, c \in \mathbb{Q}^n, b \in \mathbb{Q}^m, l \in (\mathbb{Q} \cup \{-\infty\})^n, u \in (\mathbb{Q} \cup \{\infty\})^n$ and $I \subseteq \mathbb{N} = \{1, \ldots, n\}$. Here $c^T x$ is the objective function, $Ax \leq b$ are linear constraints, l and u are the lower and upper bounds on the problem variables x, and I is a subset of indices denoting the variables required to be integer.

2.3.1 Pseudo-Boolean Constraints

A particular case of integer linear constraints are *pseudo-Boolean constraints* (PBC) [BH02], where the variables of the inequalities are restricted to be 0-1 (Boolean).

PBCs take the form:

$$a_1x_1 \ + \ a_2x_2 \ + \ \cdots \ + \ a_nx_n \ \oplus \ r$$

where $a_1, \ldots, a_n$ and r are integer constants, $x_1, \ldots, x_n$ are Boolean literals, and $\oplus$ is a relational operator in $\{<, >, \leqslant, \geqslant, =\}$.

Pseudo-Boolean optimization (PBO) is a natural extension of the Boolean satisfiability problem (SAT). A PBO instance is a set of PBC plus a linear objective function taking a similar form:

$$min: \ b_1x_1 \ + \ b_2x_2 \ + \ ... \ + \ b_nx_n$$

where $b_1, ..., b_n$ are integer coefficients and $x_1, ..., x_n$ are Boolean literals. The objective is to find a suitable assignment to the problem variables such that all the constraints are satisfied and the value of the objective function is minimized.

2.4 Satisfiability Modulo Theories (SMT)

Satisfiability Modulo Theories (SMT) is an extension of SAT where the propositional symbols of the formula can be elements of a background theory instead of Boolean variables, for instance a linear integer inequality like $2x + y \leq 3z$. This hybrid framework is interesting for CSP solving because, we can (almost) directly reformulate the constraints of the problem into SMT.

Definition 2.4.1 *A* theory *is a set of first-order formulas closed under logical consequence. A theory T is said to be* decidable *if there is an effective method for determining whether arbitrary formulas are included in T.*

Definition 2.4.2 *A formula φ is* T-satisfiable or T-consistent *if $T \cup \{\varphi\}$ is satisfiable in the first-order sense. Otherwise, it is called* T-unsatisfiable or T-inconsistent.

Definition 2.4.3 *A (partial) truth assignment M of a propositional formula can be seen either as a set or as a conjunction of literals, and hence as a formula, (see Definition 2.2.3). If M is a T-consistent partial truth assignment and φ is a formula such that $M \models \varphi$, i.e., M is a (propositional) model of φ, then we say that M is a* T-model *of φ. From now, we use $\varphi \models_T \varphi'$ as an abbreviation for $T \cup \{\varphi\} \models \varphi'$.*

Definition 2.4.4 *The* SMT problem for a theory T *is the problem of determining, given a formula φ, whether φ is T-satisfiable, or, equivalently, whether φ has a T-model.*

Note that in contrast to SAT, an SMT formula φ does not have to be a CNF. Therefore, it is quite usual to find SMT clauses in φ with other connectives than conjunctions and disjunctions.

2.4.1 (Weighted) MaxSMT

We can adapt the concept of (Weighted) MaxSAT to SMT. A MaxSMT instance is an SMT instance where clauses may have an associated weight (cost) of falsification.

Definition 2.4.5 *Given an SMT formula, the maximum SMT (MaxSMT) problem consists in finding a solution that maximizes the number of satisfied SMT clauses.*

A Partial MaxSMT problem is a MaxSMT problem where there is a set of clauses that can be violated and a set of clauses that must be satisfied. The objective is then to minimize the number of falsified clauses.

As in SAT, an SMT formula is considered to be a set of clauses while, in (Partial) MaxSMT, a CNF formula is considered to be a multiset of clauses, because repeated clauses cannot be collapsed into a unique clause. More formally:

Definition 2.4.6 *A* Weighted SMT clause *is a pair (C, w), where C is an SMT clause and w is a natural number or infinity (indicating the penalty for violating C). A* Weighted (Partial) MaxSMT formula *is a multiset of Weighted SMT clauses*

$$\varphi = \{(C_1, w_1), \ldots, (C_m, w_m), (C_{m+1}, \infty), \ldots, (C_{m+m'}, \infty)\}$$

where the first m clauses are soft and the last m' clauses are hard.

If there are no hard clauses (clauses with infinite cost of falsification) we speak of Weighted SMT, and if there are we talk about Weighted Partial MaxSMT.

Given an assignment M, the cost of a formula is the sum of the weights of the falsified clauses by M. The optimal cost of a formula is the minimal cost of all its assignments. An optimal assignment is an assignment with optimal cost.

Definition 2.4.7 *The Weighted (Partial) MaxSMT problem for a Weighted (Partial) SMT formula is the problem of finding an optimal assignment for that formula.*

Chapter 3

CSP Solving Techniques

In this chapter we present the classical solving techniques used in CSP solvers. We first present systematic search, consistency techniques, constraint propagation, global constraints and constraint optimization. Then we briefly describe MaxCSP and Weighted CSP solving approaches. Then we present three ways to solve CSPs by reformulation: propositional SAT, SMT and MILP. Finally, we present Lazy Clause Generation (which is similar to SMT: a hybrid method that uses SAT), local search and genetic algorithms.

This chapter is partially based on [Bar05, RBW06].

3.1 Systematic Search

The first classical CSP solving algorithm consists in doing a systematic search through the possible assignments to the variables. This guarantees to find a solution, if one exists, and otherwise to prove that the problem has no solution. Therefore, systematic search algorithms are sound and complete. But that could take a very long time, which is a big disadvantage. There are two main classes of systematic search algorithms:

1. Algorithms that search the space for complete assignments, i.e., assignments to all variables, till they find an assignment that satisfies all the constraints (Generate and Test).

2. Algorithms that extend a partial consistent assignment to a complete assignment that satisfies all the constraints (Backtracking, Backjumping, etc.).

In general, CSPs are computationally intractable (NP-hard). However, the techniques and algorithms developed in recent decades show that many instances can be solved in a reasonable time, but there will always be instances with very high time resolution.

In this section we present basic representatives of both classes of systematic search algorithms: generate and test, and backtracking. Although these algorithms are simple and inefficient, they are very important because they make the foundation of other algorithms that exploit more sophisticated techniques like propagation or local search.

3.1.1 Generate and Test

The generate and test (GT) algorithm generates some complete assignment and then it tests whether this assignment satisfies all the constraints. If the test fails, then it considers another complete assignment. The algorithm stops as soon as a complete assignment satisfying all the constraints is found (it is a solution) or all complete assignments have been generated without finding the solution (the problem is unsolvable).

Since the GT algorithm systematically searches the space of complete assignments, i.e., it considers each possible combination of variable assignments, the number of combinations considered by this method is equal to the size of the Cartesian product of all variable domains.

The pure generate-and-test approach is not efficient because it generates many trivially wrong assignments of values to variables which are rejected in the testing phase. In addition, the generator ignores the reason of the inconsistency and it generates other assignments with the same inconsistency.

In Section 3.7 we briefly describe local search methods, a more clever approximation to search based on complete assignments.

3.1.2 Backtracking

The backtracking (BT) algorithm incrementally attempts to extend a partial assignment, that specifies consistent values for some of the variables, towards a complete and consistent assignment. This extension is done by repeatedly choosing a value for a not yet assigned variable, until a consistent value is found according to the current partial assignment. This is the most common algorithm for systematic search [DF98].

In the BT method, variables are instantiated sequentially and as soon as all the variables relevant to a constraint are instantiated, the validity of the constraint is checked. If a partial assignment violates any of the constraints, a backtracking step is performed by reconsidering the assignment to the most recently instantiated variable that still has available alternatives. Clearly, whenever a partial assignment violates a constraint, backtracking is able to prune a subspace from the Cartesian product of all variable domains. Consequently, backtracking is strictly better than generate-and-test. However, its running time complexity for most nontrivial problems is still exponential.

This basic form of backtracking is known as chronological backtracking because the algorithm backtracks to the last decision (according to chronological order) when an inconsistency is found.

The backtracking algorithm has many drawbacks, among which we can highlight:

1. **Thrashing.** Backtracking does not necessarily identify the real reason of the conflict (inconsistency). Therefore, search in different parts of the search space keeps failing for the same reason. In order to avoid trashing we can use backjumping [Gas79], which is described below.

2. **Redundant work.** Even if the conflict reasons are identified during backjumping, they are not remembered for immediate detection of the same conflict in subsequent computations. This problem can be solved using backchecking and backmarking [HE80], also described below.

3. **Late conflict detection.** The backtracking algorithm still detects the conflict too late as it is not able to do it before the conflict really occurs, i.e., after assigning the

values to all the variables of the conflicting constraint. This drawback can be avoided by applying consistency techniques to anticipate the possible conflicts. These techniques are explained in the next section.

Backjumping

The control of backjumping is exactly the same as in backtracking, except when the backtrack takes place. Both algorithms pick one variable at a time and look for a value for this variable making sure that the new assignment is compatible with values committed to so far. However, if backjumping finds an inconsistency and all the values in the domain of such variable are explored, it analyses the situation in order to identify the source of inconsistency using the violated constraints as a guidance to find out the conflicting variable. Once the analysis is made the backjumping algorithm backtracks to the most recent conflicting variable. Notice than the backtracking algorithm always returns to the immediate past variable.

Backchecking and Backmarking

The backchecking and its descendent backmarking are useful algorithms for reducing the number of compatibility checks. If backchecking finds that some label (Y, b) is incompatible with any recent label (X, a) then it remembers this incompatibility. As long as (X, a) is still committed to, (Y, b) will not be considered again.

Backmarking is an improvement over backchecking that avoids some redundant constraint checking as well as some redundant discoveries of inconsistencies. It reduces the number of compatibility checks by remembering for every label the incompatible recent labels. Furthermore, it avoids repeating compatibility checks which have already been performed and which have succeeded.

3.2 Consistency Techniques

Consistency techniques [Kum92] were introduced to improve the efficiency of the search algorithms. The number of possible combinations that should be explored by search algorithms can be huge, while only very few assignments are consistent. Consistency techniques effectively rule out many inconsistent assignments at a very early stage, and thus cut short the search for consistent assignment. Typically this pruning is done by removing values from the domain of the variables. Consistency techniques are sound, in the sense that when the domain of a variable becomes empty, i.e., the consistency algorithm fails achieving consistency, we can ensure that the CSP has no solution. However, it is not complete, since even achieving consistency, the CSP will not necessarily have a solution.

Example 6 *Consider the CSP instance* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ *with* $\mathcal{X} = \{A, B\}$, $D(A) = \{4, \ldots, 9\}$, $D(B) = \{1, \ldots, 6\}$ *and* $\mathcal{C} = \{B > A\}$. *The consistency techniques can make the domains smaller according to the constraints:* $D(A) = \{4, \ldots, 5\}$, $D(B) = \{5, \ldots, 6\}$. *Then, for each possible value in* $D(A)$, *it is possible to find a consistent value for* B *in* $D(B)$, *and vice versa. Note however that this reduction does not remove all inconsistent pairs of labels: for instance,* $\{(A, 5), (B, 5)\}$ *is still a feasible assignment according to the domains of the variables.*

Usually, consistency techniques use polynomial-time algorithms over constraint graphs allowing to prune the search space. A constraint graph is a binary CSP representation where

nodes are variables, edges between nodes are the scope of the (binary) constraints and unary constraints are cyclic edges. Every CSP can be converted into an equivalent binary CSP, where all constraints are binary [Mon74].

3.2.1 Node Consistency

The simplest consistency technique is known as *node consistency* (NC).

Definition 3.2.1 *A variable X is* node consistent *if, and only if, for every value x in the current domain $D(X)$, each unary constraint on X is satisfied. A CSP is* node consistent *if and only if all variables are node consistent, i.e., for all variables, all values in their domain satisfy the unary constraints on that variable.*

3.2.2 Arc Consistency

Since binary constraints are represented in a constraint graph as arcs, their consistency is known as *arc consistency* (AC).

Definition 3.2.2 *An arc (X, Y) of a constraint graph is* arc consistent *if, and only if, for every value x in the current domain $D(X)$ satisfying the constraints on X, there is some value y in the domain $D(Y)$ such that the assignment $\{(X, x), (Y, y)\}$ is permitted by the binary constraint between X and Y. Note that the concept of arc-consistency is directional, i.e., if an arc (X, Y) is consistent, then it does not automatically mean that (Y, X) is also consistent. A CSP is* arc consistent *if and only if every arc in its constraint graph is arc consistent.*

An arc (X, Y) can be made consistent by simply deleting those values from the domain $D(X)$ for which there does not exist a corresponding value in the domain $D(Y)$ such that the binary constraint between X and Y is satisfied. This procedure does not eliminate any solution of the original CSP.

To make a CSP arc consistent, i.e., to make every arc of the corresponding constraint graph arc consistent, it is not sufficient to execute the consistency procedure described above for each arc just once. One execution of the consistency procedure may reduce the domain of some variable X; then each previously revised arc (Y, X) has to be revised again, because some of the members of the domain $D(Y)$ may no longer be compatible with any remaining members of the pruned domain $D(X)$. The easiest way to establish arc consistency is to apply the consistecy procedure to all arcs repeatedly till no domain of any variable changes. There are several well-known algorithms to achieve arc consistency:

1. **AC-1** [Mac77]. This algorithm is not very efficient since it repeats the consistency algorithm on all arcs if there has been any domain change during the previous iteration.

2. **AC-3** [Mac77]. This algorithm is a variation of AC-1 where the consistency test is repeated only for those arcs that are possibly affected by a previous revision.

3. **AC-4** [MH86]. This algorithm works with individual pairs of values, using *support sets* for each value. A value is *supported* if there exists a compatible value in the domain of every other variable. When a value x is removed from $D(X)$, it is not always necessary to examine all the binary constraints (X, Y). More precisely, we can ignore those values

in the domain $D(Y)$ whose support does not rely only on x. In other words, we can ignore the cases where every value in the domain $D(Y)$ is compatible with some value in the domain $D(X)$ other than x. In order to always know the support sets for all variables, the AC-4 algorithm must maintain a complex data structure.

Arc consistency does not guarantee the existence of a solution for arc consistent CSP instances. Figure 3.1 depicts a CSP instance where all constraints are arc consistent but the problem has no solution. Therefore, arc consistency is not enough to eliminate the need for search methods such as backtracking.

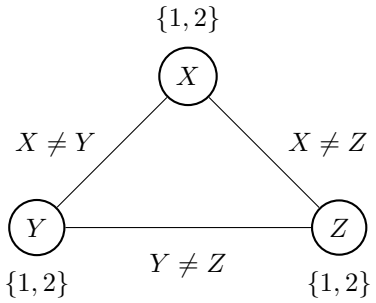

Figure 3.1: Arc consistent CSP without solution.

3.2.3 Other Consistency Techniques

There are stronger consistency techniques that are more powerful than arc consistency with respect to pruning, but at the cost of having higher computational complexity. It is a crucial issue to find the appropriate compromise between the amount of search space pruned and the time spent in pruning. Among others, there is *path consistency*, which considers triples of variables instead of tuples, and the k-consistency generalization, that considers k-tuples.

3.3 Constraint Propagation

In this section we show the result of joining some of the concepts explained in the two previous sections: systematic search (backtracking and backjumping) and consistency techniques (arc consistency).

The idea is to use a simple backtracking algorithm that incrementally instantiates variables, extending a partial assignment that specifies consistent values for some of the variables, towards a complete assignment. In order to reduce the search space, some consistency techniques are performed after assigning a value to a variable, which is known as *propagation*. Depending on which consistency technique is performed, we get different constraint propagation algorithms. The most commonly used are explained in the following subsections.

Note that, in general, consistency techniques use polynomial-time algorithms, in contrast to backtracking, which is non-polynomial. Therefore, they are performed as soon as possible, only performing search when there is no possibility to propagate.

3.3.1 Forward Checking

Forward checking is the easiest way to prevent future conflicts. Instead of performing arc consistency between instantiated variables, forward checking performs arc consistency between pairs of a not yet instantiated variable and an instantiated one. Therefore, it maintains the invariant that for every uninstantiated variable there exists at least one value in its domain which is compatible with the values of the already instantiated variables.

The forward checking algorithm is based on the following idea. When a value is assigned to a variable, any value in the domain of a non instantiated variable which conflicts with this assignment is (temporarily) removed from the domain. If the domain of a future variable becomes empty, the current partial assignment is inconsistent. Consequently, forward checking prunes branches of the search tree that will lead to a failure earlier than with chronological backtracking.

Figure 3.2 depicts an example of forward checking applied to the 4-Queens problem.[1] In the example we use Q on cells where the search has placed a queen and X for the forbidden cells due to propagation.

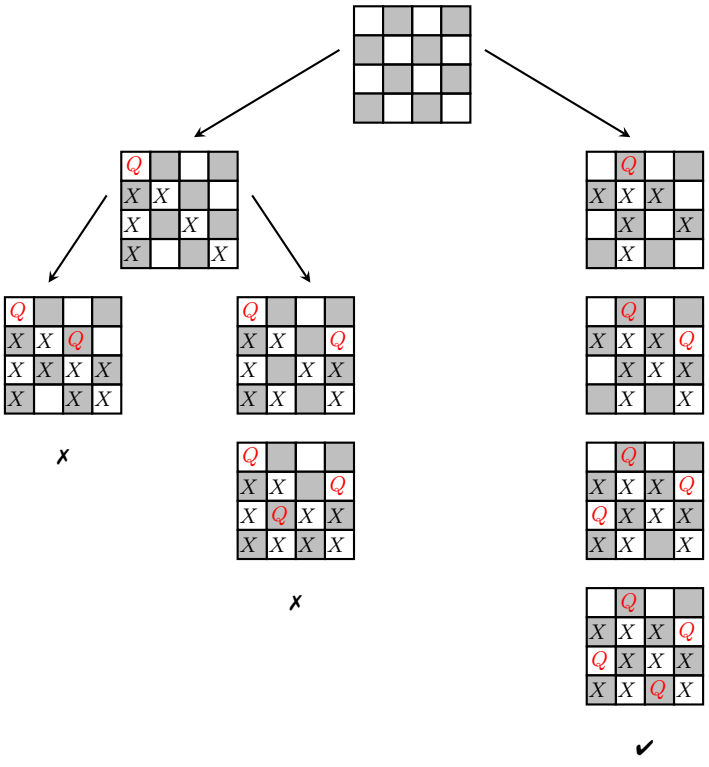

Figure 3.2: Example of Forward Checking in the 4-Queens problem. We use Q on cells where the search has placed a queen and X for the forbidden cells due to propagation.

[1]The n-Queens problem consists in placing n chess queens on an $n \times n$ chessboard so that no two queens threaten each other.

3.3.2 Look Ahead

The idea of the look ahead algorithm is to go one step further than forward checking, extending the consistency check to pairs of variables which both are not instantiated, i.e., in addition to check the consistency of the arcs of the assigned variable, it also checks the consistency of the arcs of the unassigned variables.

There are two versions of the look ahead algorithm, the first one is known as partial look ahead, it uses directional arc consistency (arc consistency only into the arcs (X_i, X_j) where $i < j$ for a given total order on the variables $\mathcal{X}$) to choose the arcs to check. The second one is known as full look ahead, it uses non-directional arc consistency to choose the arcs to check. Both of them are likely to prune more branches of the search than forward checking, but with a higher computational cost, being the full look ahead the one that prunes more but also with more work than partial look ahead.

Figure 3.3 depicts an example of full look ahead applied to the 4-Queens problem. In the example we use Q on cells where the search has placed a queen and X for the forbidden cells due to propagation.

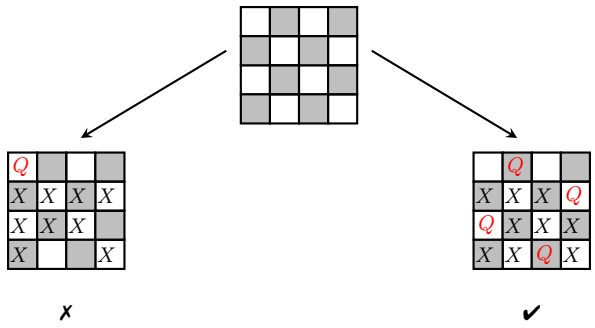

Figure 3.3: Example of Look Ahead in the 4-Queens problem. We use Q on cells where the search has placed a queen and X for the forbidden cells due to propagation.

3.3.3 Propagators

In the previous subsections we explained the classic propagation algorithms. We can define a more abstract notion of propagator [Bes06], so that it can be used by distinct algorithms.

Definition 3.3.1 *A* propagator f *is a monotonically decreasing function from domains to domains. That is,* $f(\mathcal{D}) \sqsubseteq \mathcal{D}$, *and* $f(\mathcal{D}_1) \sqsubseteq f(\mathcal{D}_2)$ *whenever* $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$.
If $\mathcal{D}_1 = \{D_1(X_1), \ldots, D_1(X_n)\}$ *and* $\mathcal{D}_2 = \{D_2(X_1), \ldots, D_2(X_n)\}$, *the operator* $\sqsubseteq$ *means that domain* D_1 *is stronger than domain* D_2, *i.e.,* $\forall i \in \{1 \ldots n\}$, $D_1(X_i) \subseteq D_2(X_i)$.

When a propagator f *is associated to a constraint* $C = \langle S, R \rangle$, *it is noted* f_C *and acts as follows: if* $\mathcal{D} = \{D(X_1), \ldots, D(X_n)\}$ *then* $f_C(\mathcal{D}) = \{D'(X_1), \ldots, D'(X_n)\}$, *where* $D'(X_j) = D(X_j)\ \forall X_j \notin S$ *and* $D'(X_j) \subseteq D(X_j)\ \forall X_j \in S$.

Typically we are interested in "correct" propagators, i.e., propagators that preserve consistent assignments. More formally, a propagator f is correct if for all consistent assignments v of a CSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, v is also a consistent assignment of $\langle \mathcal{X}, f(\mathcal{D}), \mathcal{C} \rangle$.

Example 7 *For the CSP* $\langle\{X,Y,Z\}, \{D(X)=\{1,2,3,4\}, D(Y)=\{2,3\}, D(Z)=\{2,3,4\}\}, \{C_1=(X=Y), C_2=(Y\leq Z)\}\rangle$ *we can provide the two following correct propagators:*

$$f_{C_1}(\mathcal{D})=\mathcal{D}' \quad where \quad \begin{cases} D'(X)=D(X)\cap D(Y) \\ D'(Y)=D(Y)\cap D(X) \\ D'(Z)=D(Z) \end{cases}$$

$$f_{C_2}(\mathcal{D})=\mathcal{D}' \quad where \quad \begin{cases} D'(X)=D(X) \\ D'(Y)=\{d\in D(Y) \mid d\leq max(D(Z))\} \\ D'(Z)=\{d\in D(Z) \mid d\geq min(D(Y))\} \end{cases}$$

Suppose that a search algorithm assigns the value 3 *to* X*. This means that* $D(X)=\{3\}$*. Then, the propagator* f_{C_1} *can be activated and set* $D(Y)=\{3\}$*. This, in turns, activates the propagator* f_{C_2} *reducing the domain of* Z *to* $D(Z)=\{3,4\}$*. Propagators are activated until a fix point is reached.*

Consistency algorithms (arc consistency, path consistency, ...) can use propagators to maintain consistency. Depending on their filtering power the propagator can guarantee arc consistency or other types of consistency. It turns out that consistency algorithms can also be considered as propagators.

3.4 Solving methods for Global Constraints

Although Global Constraints (GC) can be replaced by their decomposition, in general they are not replaced, because GCs simplifies the modelling process (gaining expressiveness) and it is possible to deduce more information taking into account all the constraints together. Thanks to this information it is possible to design specific and more powerful consistency algorithms, which sometimes can be borrowed from other disciplines like graph theory, flow theory, matching theory, linear programming, and finite automaton.

In other words, we represent a set of constraints with a GC in order to allow the CSP solver to exploit the structure of these constraints. For example, Régin [Rég94] proposed an arc consistency algorithm for the *alldifferent* constraint. It uses a bipartite graph to represent the relation between variables and their possible values: at one side of the graph we have nodes representing the variables, at the other side of the graph we have nodes representing the values that the variables can take, and one arc from variables to their possible values. Then it applies an algorithm based on matching theory over this bipartite graph to filter the variable values. This way, following the example in Figure 3.1, if the binary inequalities are replaced by the global constraint *alldifferent*, the solver can deduce that this CSP has no solution.

There have been a lot of research about GCs, to such an extent that has been required to make a catalogue to compile all the existing GCs. This catalogue can be found online at `http://sofdem.github.io/gccat/` and it has about 423 GC at present. Some of the most well-known GCs are: alldifferent, cumulative, global cardinality, cycle, element, sum, etc.

3.5 Constraint Optimization

In many real-life applications, we are not interested in finding any solution but an optimal one. This can be done solving the already defined Constraint Optimization Problem (COP),

which is a CSP where we want to find an optimal solution given some objective function on the variables.

In order to find the optimal solution, we potentially need to explore all the solutions of the CSP and compare the value of the objective function for each of them. Therefore, techniques for finding or generating all solutions are more relevant to COP than techniques for finding a single solution. In general, as in the case of CSPs, COPs are computationally intractable (NP-hard).

The most widely used technique for solving COPs is branch-and-bound (B&B). This algorithm consist in exploring a tree where each node represents a label and the branching from the root to a node represents a partial assignment. In the case of minimization, during the search, the algorithm keeps the cost of the best solution found so far, which is an upper bound u of the optimal solution. At each node, the algorithm computes a lower bound l of the best solution in the subtree below. If l is higher than or equal to u, the algorithm prunes the subtree below, because the current best solution cannot be improved by extending the current assignment. In this algorithm, the objective function f maps every solution (complete labeling of variables satisfying all the constraints) to a numerical value. The task is to find a solution that is optimal regarding the objective function, i.e., it minimizes or maximizes respectively the objective function.

There are other methods used to solve COPs, for instance, Russian Doll Search [VLS96, MS01], which we don't explain because they are out of scope of this thesis.

3.5.1 Weighted CSP

The classical way to solve WCSPs is reformulating them into COPs where the objective function to minimize is the sum of the violated soft constraints. There are other proposals based on semirings [BMR97] or based on arc consistency [Coo03, LS04].

3.6 Solving CSPs by Reformulation

A different way to solve CSPs is by reformulating them into other formalisms. In this section we present the most common formalisms used in reformulation: SAT, MILP and SMT.

3.6.1 Reformulation into SAT

It consists on encoding CSPs into Boolean satisfiability problems (SAT). Recall from definition 2.1.1 that a CSP is defined as a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X} = \{X_1, \ldots, X_n\}$ is a set of variables, $\mathcal{D} = \{D(X_1), \ldots, D(X_n)\}$ is a set of domains containing the values the variables may take, and $\mathcal{C} = \{C_1, \ldots, C_m\}$ is a set of constraints. First of all we have to choose how we want to encode the variables and their domains, and then translate the constraints taking into account this encoding in such a way that they retain their meaning.

Next we describe several well-known CSP to SAT encodings, where we assume that the domain of all variables is $\{1 \ldots n\}$ for the sake of simplicity.

Variable Encodings

Due to the fact that the domain size of the CSP variables can be greater than 2 and Boolean variables can only be *false* or *true*, in general, we will need several Boolean variables to

represent a CSP variable. There are several SAT encodings for variables. Next we present three of them, which are the most common used:

- **Normal encoding.** This is the most intuitive encoding because for each value $i \in D(X)$, where X is a CSP variable, we introduce a Boolean variable x_i which will be *true* only iff $X = i$. Since a CSP variable only takes one value, we also need to encode that exactly one of the Boolean variables $\{x_1, \ldots, x_n\}$ is *true*. This can be done using the **Exactly-One** constraint (described below).

  In the literature this encoding is also known as *standard encoding* [AM04] and *sparse encoding*, name given at [Hoo99] to refer the encoding used in [dK89].

- **Log encoding [IM94].** In this encoding we introduce as many variables as *bits* are needed to represent the values in the domain of the CSP variable, i.e., to encode the CSP variable X we introduce m variables: $x_1, \ldots, x_m$, where $m = \lceil log_2 |D(X)| \rceil$. The polarity of these variables will determine which value is taking variable X. For instance, for a variable X with domain $D(X) = \{1, 2, 3, 4\}$ we need 2 bits, where $\neg x_1 \wedge \neg x_2$ encodes $X = 1$, $x_1 \wedge \neg x_2$ encodes $X = 2$, $\neg x_1 \wedge x_2$ encodes $X = 3$ and $x_1 \wedge x_2$ encodes $X = 3$.

  Note that in this encoding the combination of variables $x_1, \ldots, x_m$ is representing only one value for variable X at the same time (depending on the polarity of each variable). Therefore, we don't need the Exactly-One constraint. But, on the other hand, when the number of values in $D(X)$ is not a power of 2, we will need extra clauses, called *prohibited-value clauses*, to forbid the non possible values of the bits. This can be seen in Example 8.

  In the literature this encoding is also known as *compact encoding* [Hoo99]. Sometimes, when only log variables are used, it is also known as *full log encoding*, while *log encoding* is used when log variables are combined with standard variables (encoding proposed in [FP01]). The combination between log and standard variables is done by the following linking clauses:

$$x_i \leftrightarrow (l_1^i \wedge \ldots \wedge l_{\lceil log_2 |D(X)| \rceil}^i)$$

  where x_i are standard variables representing $X = i$, and l_j^i are literals corresponding to the j bits representing the assignment $X = i$. This way, we can use the standard variables in the formula while we are avoiding to use the Exactly-One constraint.

  Following the example of a variable X with domain $D(X) = \{1, 2, 3, 4\}$ we link the log and standard variables:

$$\begin{array}{l} (x_1 \leftrightarrow \neg b_1 \wedge \neg b_2) \\ (x_2 \leftrightarrow \neg b_1 \wedge b_2) \\ (x_3 \leftrightarrow b_1 \wedge \neg b_2) \\ (x_4 \leftrightarrow b_1 \wedge b_2) \end{array}$$

- **Regular encoding [AM04, BHM01].** In this encoding, for each value $i \in D(X)$, we introduce a Boolean variable $x_i^{\geq}$ (called regular variable), which will be *true* iff $X \geq i$. In addition we add the following auxiliary clauses to ensure that if $x_i^{\geq}$ is *true* then $x_{i-1}^{\geq}$ also has to be *true*:

$$\bigwedge_{i=1}^{i<n} x_{i+1}^{\geq} \rightarrow x_i^{\geq}$$

Then when we have to encode $X = i$ in the formula we use $x_i^{\geq} \wedge \neg x_{i+1}^{\geq}$, except the cases of $i = l$ and $i = u$, where l and u are the lower and upper bounds of X, that we will use $\neg x_{l+1}^{\geq}$ and $x_u^{\geq}$ respectively.

A similar idea of this encoding appears in several places in the literature [BB03, CB94, TTKB09, GN04] also known as *order encoding* and *ladder encoding*. What is different between them are the auxiliary clauses and how they use the encoded variables in the formula.

As well as in the case of the log encoding, sometimes in the literature, the regular encoding is called *full regular encoding* when only regular variables are used, and *regular encoding* when regular variables are combined with standard variables (encoding proposed in [AM04]). The combination between standard and regular variables is done by the following linking clauses:

$$\bigwedge_{i=1}^{i<n} x_i \leftrightarrow (x_i^{\geq} \wedge \neg x_{i+1}^{\geq})$$

In [AM04] is proposed another variant of this encoding called *half regular encoding*. It also links regular and standard variables by the previous linking clauses, but it uses the regular or standard variable encoding in the formula depending on the polarity of the variable assignment ($X = i$ or $\neg X = i$), in order to keep a formula as smaller as possible.

Example 8 *The direct full log encoding for the CSP* $\langle \{X, Y\}, \{D(X) = \{0, 1, 2\}, \{D(Y) = \{1, 2, 3\}, \{X \neq Y\}\rangle$ *goes as follows:*

value domain (i)	b_1	b_0	*literals for* $X = i$	
$X = 0$	0	0	$\neg x_1$	$\neg x_0$
$X = 1$	0	1	$\neg x_1$	x_0
$X = 2$	1	0	x_1	$\neg x_0$

value domain (i)	b_1	b_0	*literals for* $Y = i$	
$Y = 1$	0	0	$\neg y_1$	$\neg y_0$
$Y = 2$	0	1	$\neg y_1$	y_0
$Y = 3$	1	0	y_1	$\neg y_0$

conflict values	*conflict clauses*
$X = 1, Y = 1$	$x_1 \vee \neg x_0 \vee y_1 \vee y_0$
$X = 2, Y = 2$	$\neg x_1 \vee x_0 \vee y_1 \vee \neg y_0$

prohibited-value clauses
$\neg x_1 \vee \neg x_0$
$\neg y_1 \vee \neg y_0$

In the first row of tables we can observe the sets of literals associated with the possible assignments of X (left) and Y (right). In the second row of tables there are the conflict clauses (left) and the prohibited-value clauses (right).

Exactly-One encoding

In the normal encoding we need an Exactly-One constraint which is commonly expressed as the conjunction of the ALO (At-Least-One) constraint and the AMO (At-Most-One) constraint. The ALO constraint states that at least one of the variables is *true* while the AMO constraint states that at most one of the variables is *true*.

There exist different encodings of the ALO and AMO constraints. But in general, the most common encoding for the ALO constraint is using a n-ary clause:

$$x_1 \vee \ldots \vee x_n$$

For the AMO there are the following common encodings:

- **The pair-wise encoding.** This encoding is formed by the following $n * (n-1)/2$ binary clauses:

$$\bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} \neg x_i \vee \neg x_j$$

  In the literature it is also known as naive encoding.

- **The Binary AMO**, introduced in [FP01], associates with each x_i a unique bit string $s_i \in \{1,0\}^{\lceil log_2 n \rceil}$ corresponding to the binary representation of i. The binary AMO is:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=i}^{\lceil log_2 n \rceil} \neg x_i \vee \psi(i,j)$$

  where $\psi(i,j)$ is a clause with j literals, $j = \lceil log_2 n \rceil$, and each literal l_j will be b_j ($\neg b_j$) if the j_{th} bit of s_i is 1 (0). Note that we need $\lceil log_2 n \rceil$ new variables: $b_1, \ldots, b_{\lceil log_2 n \rceil}$.

  This encoding introduces $\lceil log_2 n \rceil$ extra variables and uses $n * \lceil log_2 n \rceil$ clauses.

- **The Sequential counter AMO**, introduced in [Sin05], is based on a sequential counter circuit, that consists in sequentially counting the number of x_i that are *true*. The Sequential counter AMO is:

$$(\neg x_1 \vee s_1)$$

$$\bigwedge_{i=2}^{i<n} ((\neg x_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg x_i \vee \neg s_{i-1}))$$

$$(\neg x_n \vee \neg s_{n-1})$$

  where s_i, $1 \leq i \leq n-1$, are auxiliary variables.

  This encoding requires $3n-4$ binary clauses and $n-1$ auxiliary variables.

Constraint Encodings

Next we describe the most common used SAT encodings for constraints:

- **The Direct Encoding** is probably the most common encoding from CSP into SAT. In the direct encoding for each binary constraint with scope $\{X, Y\}$ there is a binary clause for every nogood. Such clauses are called conflict clauses.

  Below we provide three examples of the direct encoding, in Example 9 variables are represented using the standard encoding, in Example 8 the variables are represented using the full log encoding, and in Example 10 the variables are represented using the full regular encoding.

  Example 9 *The conflict clauses in the direct encoding when variables are encoded using the standard encoding (from now on* standard direct encoding*), for the CSP defined by* $\langle\{X, Y\}, \{D(X) = \{1, 2, 3\}, D(Y) = \{1, 2, 3\}\}, \{X \leq Y\}\rangle$ *are the following:*

  $$(\neg x_2 \vee \neg y_1) \quad (\neg x_3 \vee \neg y_1) \quad (\neg x_3 \vee \neg y_2)$$

  Example 10 *The direct encoding for the CSP defined by* $\langle\{X, Y\}, \{D(X) = \{1, 2, 3\}, D(Y) = \{1, 2, 3\}\}, \{(X = 2 \wedge \neg Y = 2)\}\rangle$ *in the full regular, regular and half regular encodings is the following:*

  $$\begin{array}{ll} \textit{Full regular:} & (x_2^{\geq} \wedge \neg x_3^{\geq}) \wedge \neg(y_2^{\geq} \wedge \neg y_3^{\geq}) \\ \textit{Regular:} & x_2 \wedge \neg y_2 \\ \textit{Half regular:} & x_2 \wedge \neg(y_2^{\geq} \wedge \neg y_3^{\geq}) \end{array}$$

- **The Support Encoding** encodes into clauses the support for each possible value of a variable across a constraint. The support for a value i of a CSP variable X across a binary constraint with scope $\{X, Y\}$ is the set of values of the variable Y which allow $X = i$.

  Example 11 *The standard support clauses for the CSP of Example 9 are:*

  $$(\neg x_2 \vee y_2 \vee y_3) \quad (\neg x_3 \vee y_3) \quad (\neg y_1 \vee x_1) \quad (\neg y_2 \vee x_1 \vee x_2)$$

  The support clause for $X = 1$ *is missing because it is subsumed by* $(y_1 \vee y_2 \vee y_3)$*, and the support clause for* $Y = 3$ *is missing because it is subsumed by* $(x_1 \vee x_2 \vee x_3)$*.*

3.6.2 Reformulation into MILP

In order to encode a CSP using MILP (defined in Section 2.3) we can *directly* translate the arithmetic constraints (in case to be non-linear we have to linearise them). But several types of constraints cannot directly be translated into arithmetic expressions, for instance the ones containing logical operators (such as $\vee$, $\wedge$, $\rightarrow$, ...). To be able to express these kind of constraints we need to reify the literals of the clauses, which means to reflect the satisfaction of each literal into a pseudo-Boolean variable. Then, e.g., $a \vee b$ can be translated into $reif_a + reif_b \geq 1$, where $reif_a$ and $reif_b$ are the pseudo-Boolean variables corresponding to the Boolean variables a and b, respectively.

Note that reification of other kind of constraints could a bit harder. For instance, in Example 12 we show the reification of an inequality using the BigM method, as described in [Wil99].

Example 12 *Assume we want to reify a constraint* $\sum_j a_j x_j \leqslant b$*, being* δ *the corresponding reification variable. The relation between the constraint and* δ *can be modelled as follows:*

- $\delta = 1 \rightarrow \sum_j a_j x_j \leqslant b$ *is translated into* $\sum_j a_j x_j + M\delta \leqslant M + b$*, where* M *is an upper bound for the expression* $\sum_j a_j x_j - b$*.*
- $\delta = 0 \rightarrow \sum_j a_j x_j \not\leqslant b$*, i.e.,* $\delta = 0 \rightarrow \sum_j a_j x_j > b$*, which is translated into* $\sum_j a_j x_j - m\delta \geqslant b$*, where* m *is a lower bound for the expression* $\sum_j a_j x_j - b$*.*

The CSP now encoded into MILP can now be solved using a MILP solver, which, in general, uses the simplex algorithm to solve the linear program (LP) relaxation:

$$min\{c^T x : Ax \leq b, l \leq x \leq u, x \in \mathbb{R}^n\}$$

Notice that it is the same problem but with the difference that the integer requirement on the x variables indexed by I has been dropped. The main reason for this relaxation is that thanks to this, the relaxed problem turns to be polynomial.

Finally, to ensure that the variables indexed by I are integer, MILP solvers integrate a variant of the branch-and-bound and the cutting plane algorithms [Gom58] of the general branch-and-cut scheme [PR91].

3.6.3 Reformulation into SMT

Recall that an SMT formula can be seen as a propositional formula extended with expressions of one or more background theories. Therefore, in order to encode CSPs to SMT, first of all, we have to choose which theories we want to use. Since CSPs are usually composed of Boolean and arithmetic constraints, it is quite common to use the linear integer arithmetic (LIA). This way, we can (almost) directly translate all the constraints, because we can define Boolean and integer variables, use logical operators (like disjunction, conjunction, implies, etc.) and use arithmetic expressions.

More details on how to encode CSPs to SMT can be found in Chapters 5 and 7 since this is one of the contributions of this thesis.

3.7 Other CSP Solving Methods

In recent years there has been a proliferation of alternative methods to traditional search trying, with varying success, to solve CSPs. These new approaches include exact methods, methods that do not guarantee the best solution and even methods that do not guarantee any solution. In this section we present the more relevant ones.

3.7.1 Lazy Clause Generation

Lazy Clause Generation (LCG) is a hybrid technique that combines a SAT solver with finite domain propagators. The original architecture for the LCG [OSC07], which can be seen in

Figure 3.4, is very similar to SMT, because the search is controlled by a SAT solver and the FD propagators are treated as clause generators for the SAT solver. However, the new architecture for LCG [FS09], which can be seen in Figure 3.5, is completely opposite to SMT, because the search is controlled by a FD solver and uses the SAT solver as an oracle to control the possible values of the variable domains. Since the new architecture for LCG is the state-of-the-art, we describe it below (taken from [FS09]).

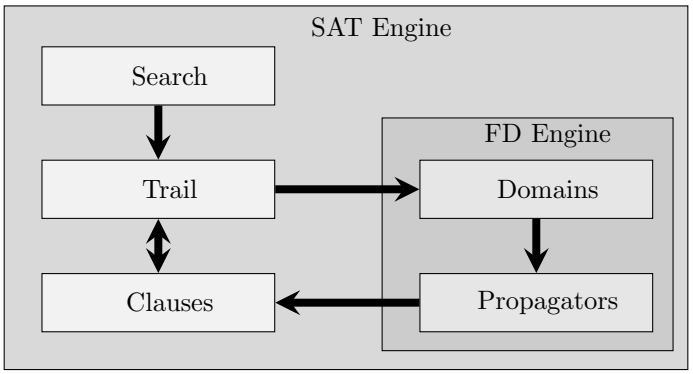

Figure 3.4: Original architecture of LCG

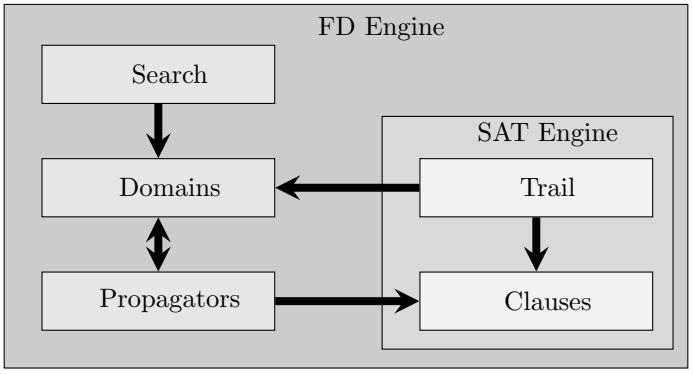

Figure 3.5: New architecture of LCG

In the new LCG architecture, the search is controlled by a finite domain solver. When there is a change in the domain of a variable, propagators are woken and placed in a priority queue. The SAT solver unit propagation engine acts as a global propagator. Whenever a literal is set or clauses are posted to the SAT solver, then this propagator is scheduled for execution at the highest priority.

When a propagator f is executed updating a variable domain ($f(\mathcal{D}) \neq \mathcal{D}$) or causing a failure ($\mathcal{D}$ is empty) it posts an explanation clause to the SAT solver that explains the domain reduction or the failure.

The SAT propagator when executed computes a unit fixpoint, i.e., performs unit propagation to reach a unit propagation fixpoint. Then each of the literals fixed by unit propagation causes the corresponding domain changes to be made in the domain, which may wake up other propagators. The cycle of propagation continues until a fixpoint is reached.

The LCG solver associates each integer variable with a set of Boolean variables. Changes in these Boolean variables will be reflected in the domains of the integer variables. There are two possible ways of doing this

- **Array Encoding**. In this encoding there exist two arrays of Boolean variables:
  - An array of inequality (regular) literals $x_i^{\leq}, i \in [l, u-1]$, which are generated eagerly with the restrictions $\bigwedge_{i=1}^{i<n-1} \neg x_i^{\leq} \vee x_{i+1}^{\leq}$.
  - An array of equality literals $x_i, i \in [l, u]$, which are generated lazily. When an equality literal x_i is necessary (occurs in a propagator or in a model), the following clauses are posted:

$$\neg x_i \vee x_i^{\leq}$$
$$\neg x_i \vee \neg x_{i-1}^{\leq}$$
$$x_{i-1}^{\leq} \vee \neg x_i^{\leq} \vee x_i$$

- **List Encoding**. In this encoding the inequality and the equality literals are generated lazily when they are required for propagation or explanation.

  When an inequality literal $x_i^{\leq}$ is introduced:

  - The closest existing bounds for x_i are determined:

$$l = \max\{j \mid x_j^{\leq} \text{ exists}, j < i\}$$
$$u = \min\{j \mid x_j^{\leq} \text{ exists}, i < j\}$$

  - The following new domain clauses are posted:

$$\neg x_l^{\leq} \vee x_i^{\leq}$$
$$\neg x_i^{\leq} \vee x_u^{\leq}$$

  When an equality literal x_i is introduced, the introduction of the literals $x_i^{\leq}$ and $x_{i-1}^{\leq}$ is required. Then the solver proceeds as in the case of array encoding.

By default the solver uses array encoding for variables with "small"($<$ 10000) initial domains and list encoding for larger domains.

Example 13 *Consider* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ *where* $\mathcal{X} = \{X, Y, Z\}, D(X) = D(Y) = D(Z) = \{1, 2, 3\}$ *and* $\mathcal{C} = \{C_1 = (X = Y), C_2 = (Y < Z)\}$.
The propagators are:

$$f_{C_1}(\mathcal{D}) = \mathcal{D}' \quad \textit{where} \quad \begin{cases} D'(X) = D(X) \cap D(Y) \\ D'(Y) = D(Y) \cap D(X) \\ D'(Z) = D(Z) \end{cases}$$

$$f_{C_2}(\mathcal{D}) = \mathcal{D}' \quad \textit{where} \quad \begin{cases} D'(X) = D(X) \\ D'(Y) = \{d \in D(Y) \mid d < max(D(Z))\} \\ D'(Z) = \{d \in D(Z) \mid d > min(D(Y))\} \end{cases}$$

The initial clauses for the array encoding would be:

- *Variables:*
$$\{x_1^{\leq}, x_2^{\leq}, y_1^{\leq}, y_2^{\leq}, z_1^{\leq}, z_2^{\leq}\}$$
- *Clauses:*
$$\{\neg x_1^{\leq} \vee x_2^{\leq}, \neg y_1^{\leq} \vee y_2^{\leq}, \neg z_1^{\leq} \vee z_2^{\leq}\}$$

The process of the LCG solver starts when the search algorithm tests a label, for example X with value 2. In this moment it has to generate the x_2 equality variable and the clauses $\neg x_2 \vee x_2^{\leq}$, $\neg x_2 \vee \neg x_1^{\leq}$, $x_1^{\leq} \vee \neg x_2^{\leq} \vee x_2$. Then it activates the f_{C_1} propagator, which generates the clause $\neg x_2 \vee y_2$. As y_2 is not present in the Boolean formula, it has to generate this equality variable and the associated clauses. This process continues until a fixed point is reached. Then the search algorithm tests a new label or does backjumping.

In [AS12] is proposed a variant for LCG solvers called *Lazy Decomposition (LD)*. The idea comes from that in some problems it is better to decompose (reformulate) the propagator into SAT and in other problems it is better to use the propagator to generate nogoods for the SAT solver. Therefore, it is only needed to modify the propagators to detect when a variable is active (counting the number of times it generates a nogood), and replace it by its decomposition (or a part of it).

LCG is a highly competitive approach in general, and its last implementation is one of the most competitive of state-of-the-art CSP solvers.

3.7.2 Local search

This technique consists on moving from solution to solution in the space of candidate solutions (the search space) by applying local changes, until a solution deemed optimal is found or a time bound is elapsed. A local search algorithm starts from a solution and then iteratively moves to a neighbour solution. This is only possible if a neighbourhood relation is defined on the search space. As an example, the neighbourhood of a vertex cover is another vertex cover only differing by one node. Despite the inherent incompleteness, these methods have shown great ability to solve difficult problems.

One of the local search methods most used today is tabu search [Glo89, Glo90].

Central to tabu search is a tabu list, which keeps a list of moves that may not be performed. Thus, when generating a list of candidates from a current solution, some neighbouring solutions cannot be added to the list. The tabu list serves to ensure that the moves in the list are not reversed thus preventing previous moves from being repeated. Criteria for moves entering the tabu list can be defined in many ways, and similar criteria exist for moves to be off the tabu list.

3.7.3 Genetic algorithms

Genetic algorithms [BFM97]. This approach is based on an analogy with the evolution theory. In these algorithms, a state corresponds to a total assignment; one derives new states by recombining two parent states using a mating function that produces a new state corresponding to a cross-over of the parent states. The parent states are selected from a pool of states called population. The new states are subjected to small random changes (mutations). This approach does not guarantee that a solution is found although it exists.

Chapter 4

SAT and Satisfiability Modulo Theories (SMT)

In this chapter we introduce the SMT paradigm. To do so, we first explain in Section 4.1 how modern SAT solvers work, since SAT solvers are the base of SMT solvers, and then in Section 4.2 we define SMT and describe how SMT solvers work. Since we are also interested in Weighted MaxSMT, we explain the more relevant (Weighted) MaxSAT algorithms.

This chapter is partially based on [BHvMW09].

4.1 SAT

Since the SAT problem (defined in 2.2) is NP-complete, only algorithms with exponential worst-case complexity are known for it. In spite of this, efficient and scalable algorithms for SAT were developed over the last decade and have contributed to dramatic advances in automatically solve problem instances involving hundreds of thousands of variables and millions of clauses. The advances in these algorithms have had a good impact on MaxSAT (defined in 2.2.1) algorithms.

In this section we explain the more relevant algorithms used for solving SAT and Weighted MaxSAT problems.

4.1.1 Satisfiability Algorithms

There are two types of algorithms used for solving SAT: complete and incomplete. Complete algorithms perform a search through the space of all possible truth assignments, in a systematic manner, to that prove either a given formula is satisfiable (the algorithm finds a satisfying truth assignment) or unsatisfiable (the algorithm completely explores the search space without finding any satisfying truth assignment). By contrast, incomplete algorithms (e.g. local search algorithms) usually do not completely explore the search space, and a given truth assignment can be considered more than once.

Next we describe the Conflict-Driven Clause-Learning algorithm, which is the base algorithm used in state-of-the-art complete SAT solvers. We do not discuss the local search algorithms, since this topic is out of the scope of this thesis. Some basic concepts on local search have been described in Section 3.7.

Conflict-Driven Clause-Learning (CDCL)

The CDCL algorithm is based on the Davis-Putman-Loveland-Logemann algorithm (DPLL) [DP60, DLL62]. The DPLL is a complete backtracking-based search algorithm that works as follows. It chooses a literal, assigns a truth value to it and simplifies the formula by removing from the formula all clauses that become true under that assignment and removing the literals that become false from the remaining clauses. Then it recursively checks if the simplified formula is satisfiable. If the recursive check returns satisfiable it means that the original formula is satisfiable; otherwise, it assigns the opposite truth value to the chosen literal, simplifies the formula and recursively checks the satisfiability of the new simplified formula. If this second check returns satisfiable it means that the original formula is satisfiable and unsatisfiable otherwise. This is known as the *splitting rule*, as it splits the problem into two simpler sub-problems. Before apply the splitting rule DPLL algorithms usually apply other simplification techniques like *unit propagation* and *pure literal rule*, both of them used in Algorithm 1 and explained next.

The main difference in the CDCL algorithm is that the search is conflict-driven, that is, the original DPLL algorithm is improved with learning and non-chronological backtracking to facilitate the pruning of the search space. Algorithm 1 shows the pseudo-code of the CDCL algorithm.

Algorithm 1 Conflict-Driven Clause-Learning algorithm

Input: ϕ : CNF formula
Output: Satisfiability of ϕ

```
dl ← 0 {Decision level}
ν ← ∅ {Current assignment}
UnitPropagation(φ,ν)
PureLiteralRule(φ,ν)
if □ ∈ φ then
  return false
end if;
while not AllVariablesAssigned(φ,ν) do
  (x, v) ← PickBranchingVariable(φ, ν)
  dl ← dl + 1
  ν ← ν ∪ {(x, v)}
  UnitPropagation(φ,ν)
  PureLiteralRule(φ,ν)
  if □ ∈ φ then
    β ← ConflictAnalysisAndLearning(φ,ν)
    if β = 0 then
      return false
    else
      NonChronologicalBacktrack(φ,ν,β)
      dl ← β
    end if
  end if
end while
return true
```

Next we define the functions used in Algorithm 1:

- **UnitPropagation** is the iterated application of the Unit Clause (UC) rule (also referred to as the one-literal rule) until an empty clause is derived or there are no unit clauses left. If a clause is unit, then its single literal must be *true* under any satisfying assignment. If $\{l\}$ is a unit clause of a CNF formula ϕ, UC consists in deleting all the clauses in ϕ with literal l, and removing all the occurrences of literal $\neg l$ in the remaining clauses. If the empty clause is found, then a conflict indication is returned.

- **PureLiteralRule** (also referred to as monotone literal rule). A literal is pure if its complementary literal does not occur in the CNF formula. The satisfiability of a CNF formula is unaffected by satisfying its pure literals. Therefore, all clauses containing a pure literal can be removed.

- **PickBranchingVariable** selects a variable to assign, and its respective value. The variable selection heuristic is decisive for finding as quick as possible a solution with the DPLL-based procedures [Sil99]. A bad heuristic can lead to explore the whole search space, whereas a good heuristic allows us to cut several regions, and even not to traverse more than a single branch in the best case. The variable is selected and used to split the CNF formula into two subproblems. There are a lot of heuristic methods for select the variable to assign [DPBC93, Pre93, Fre95, JW90].

- **ConflictAnalysisAndLearning** consists in analyzing the most recent conflict, and learning a new clause from the conflict. During the conflict analysis process, the information about the current conflict can be stored using redundant clauses [BS94, BGS99]. These redundant clauses do not change the satisfiability of the original formula, but they help to prune parts of the search space with similar conflicts. This technique is called clause learning or conflict driven clause learning.

- **NonChronologicalBacktrack** backtracks to the decision level computed by ConflictAnalysisAndLearning. The NonChronologicalBacktrack procedure detects the reason of the conflict and often backtracks to a lower decision level than the previous.

- **AllVariablesAssigned** tests if all variables have been assigned, in which case the algorithm terminates indicating a satisfiable result.

The performance of the DPLL-based procedures critically depends upon the care taken in the implementation. DPLL-based solvers spend much of their time applying unit propagation [Zha97, LMS05], and this has motivated the definition of several proposals to reduce the cost of applying unit propagation. The most efficient in the state-of-the-art is the unit propagation algorithm based on the called 2-literal watching scheme, first used in Chaff [MMZ$^+$01].

Finally, in order to avoid getting stuck in hard portions of the search space, most modern SAT solvers periodically do restarts with the hope of exploring easier successful branches, but remembering what has been learned. To avoid incompleteness, it suffices to perform restarts at increasing periods of time (typically counting up to a certain number of conflicts). But many other types of restart policies have been studied, such as arithmetic o geometric series over the number of conflicts, or especial series such as the Luby series or Inner-Outer Geometric series [Hua07, RS08].

4.1.2 (Weighted) MaxSAT Algorithms

In recent years several methods have been described to solve (Weighted) MaxSAT (defined in 2.2.1).

A first approach could consist in reifying the soft clauses into Boolean variables, i.e., reflect the satisfaction of each soft clause into Boolean variable. These Boolean variables can be considered as pseudo-Boolean, i.e., $\{0, 1\}$ integer variables, corresponding 0 to *false* and 1 to *true*. The objective function in the case of MaxSAT is the sum of these pseudo-Boolean variables and in the case of Weighted MaxSAT the sum of products of the pseudo-Boolean variable by the corresponding clause weight. In both cases we must minimize the objective function.

Another possibility widely used in recent times (winning in several categories of the last SAT competitions) is to achieve the optimal solution using unsatisfiability cores (UNSAT cores). We look in some more detail to this kind of algorithms since the use of these methods for SMT is one of the thesis contributions.

UNSAT Core Based Algorithms

Definition 4.1.1 *An* unsatisfiable core *is an unsatisfiable subset of clauses of the original CNF.*

There are several algorithms for computing smaller unsatisfiable cores of propositional formulas [ZM03, DHN06, ZLS06].

Currently there are a large number of UNSAT core based algorithms and numerous variants of each. Next we describe the more relevant ones related to this thesis:

- **PM1** was proposed in [FM06] for the Partial MaxSAT problem. This algorithm consists in iteratively calling a SAT solver on a working formula ϕ (see Algorithm 2).

  First of all, the SAT solver will check if the formula is satisfiable or not, and in case to be unsatisfiable, it will return an unsatisfiable core (ϕ_c). At this point the algorithm will create as many new variables (BV), called blocking variables, as soft clauses appearing in the returned core. Then, the new working formula ϕ will consist of the previous ϕ where a blocking variable is added to each clause of the returned core, plus a cardinality constraint to ensure that exactly one of the new blocking variables should be *true*: $CNF(\sum_{b \in BV} ite(b; 1; 0) = 1)$. Finally, it will increase the counter of falsified clauses (*cost*) by one. This procedure will be applied until the SAT solver finds a satisfiable formula.

  Example 14 *Let us consider a Partial MaxSAT instance ϕ having the following clauses: $\{(x), (y), (z), [\neg x \vee \neg y], [x \vee \neg z], [y \vee \neg z]\}$. We show the application of the PM1 algorithm on ϕ in Figure 4.1.*

  Hard clauses are represented between square brackets, and soft clauses are represented between round brackets.

- **WPM1** was proposed in [ABL09] for the Weighted MaxSAT. This is the weighted version of the PM1 (see Algorithm 3).

Algorithm 2 PM1 Algorithm

Input: ϕ : CNF formula
Output: Cost of ϕ

```
cost ← 0
while true do
  (st, φ_c) ← SAT_ALGORITHM(φ)
  if st = SAT then
    return cost
  else
    BV ← ∅
    for all C ∈ φ_c do
      if isSoft(C) then
        b ← New_variable()
        φ ← φ \ {C} ∪ {(C ∨ b)}
        BV ← BV ∪ {b}
      end if
    end for
    if BV = ∅ then
      return UNSAT
    end if
    φ ← φ ∪ CNF(∑_{b∈BV} ite(b; 1; 0) = 1)
    cost ← cost + 1
  end if
end while
```

1st. iter. Clauses	2nd. Iter. Clauses	3rd. iter. Clauses
$\underline{x}$	$\underline{x \vee b_1}$	$x \vee b_1 \vee b_3$
y	$\underline{y}$	$y \vee b_4$
$\underline{z}$	$\underline{z \vee b_2}$	$z \vee b_2 \vee b_5$
$\underline{\neg x \vee \neg y}$	$\underline{\neg x \vee \neg y}$	$\neg x \vee \neg y$
$x \vee \neg z$	$\underline{x \vee \neg z}$	$x \vee \neg z$
$\underline{y \vee \neg z}$	$y \vee \neg z$	$y \vee \neg z$
	$\underline{CNF(b_1 + b_2 = 1)}$	$CNF(b_1 + b_2 = 1)$
		$CNF(b_3 + b_4 + b_5 = 1)$
$cost = 0$	$cost = 1$	$cost = 2$

Figure 4.1: Application of the PM1 algorithm on the Partial MaxSAT formula $\{(x), (y), (z), [\neg x \vee \neg y], [x \vee \neg z], [y \vee \neg z]\}$. UNSAT cores of each stage are underlined.

In this algorithm, when the SAT solver returns an unsatisfiable core (ϕ_c), we calculate the minimum weight of the soft clauses of ϕ_c (w_{min} in the algorithm). Then, we transform the working formula ϕ in the following way: we duplicate the core, having on one of the copies, the clauses with weight $w_i - w_{min}$ (the original weight minus the minimum weight); and on the other copy, we add the blocking variables (BV) and the minimum weight to the clauses. Note that when $w_i - w_{min} = 0$ the clause $(C_i, w_i - w_{min})$ is not added. Finally we add the cardinality constraint on the blocking variables, and we add

w_{min} to the cost.

Algorithm 3 WPM1 Algorithm

Input: $\phi = \{(C_1, w_1), \ldots, (C_n, w_n)\}$: CNF formula
Output: Cost of ϕ

```
cost ← 0
while true do
  (st, φ_c) ← SAT_ALGORITHM({C_i | (C_i, w_i) ∈ φ})
  if st = SAT then
    return cost
  else
    BV ← ∅
    w_min ← min{w_i | C_i ∈ φ_c ∧ isSoft(C_i)}
    for all C_i ∈ φ_c do
      if isSoft(C_i) then
        b ← New_variable()
        φ ← φ \ {(C_i, w_i)} ∪ {(C_i, w_i − w_min)} ∪ {(C_i ∨ b, w_min)}
        {when w_i − w_min = 0 the clause (C_i, w_i − w_min) is not added}
        BV ← BV ∪ {b}
      end if
    end for
    if BV = ∅ then
      return UNSAT
    end if
    φ ← φ ∪ {(C, ∞)|C ∈ CNF(∑_{b∈BV} ite(b; 1; 0) = 1)}
    cost ← cost + w_min
  end if
end while
```

Example 15 *Let us consider a Weighted MaxSAT instance ϕ having the following clauses: $\{(x,1),(y,2),(z,3),(\neg x \vee \neg y,\infty),(x \vee \neg z,\infty),(y \vee \neg z,\infty)\}$. We show the application of the WPM1 algorithm on ϕ in Figure 4.2.*

- **PM2** was also proposed in [ABL09]. It is an improvement of PM1 where only one blocking variable is used for each soft clause.

4.2 SMT

Satisfiability Modulo Theories (SMT) (defined in 2.4) is an extension of SAT where the propositional symbols of the formula can be elements of a background theory instead of boolean variables, for instance a linear integer inequality like $3x + 2y \geq 3z$.

In this section we first explain the two approaches used in SMT solvers. Then we introduce the most common theories and logics used in SMT. Finally we briefly explain the existing work in (Weighted) MaxSMT.

1st. iter.		2nd. iter.		3rd. iter.		4th. iter.	
Clauses	W.	Clauses	W.	Clauses	W.	Clauses	W.
$\underline{x}$	1						
y	2	$\underline{y}$	2				
$\underline{z}$	3	$\underline{z}$	2				
		$x \vee b_1$	1	$\underline{x \vee b_1}$	1		
		$z \vee b_2$	1	$z \vee b_2$	1	$z \vee b_2$	1
				$\underline{y \vee b_3}$	2	$y \vee b_3$	1
				$\underline{z \vee b_4}$	2	$z \vee b_4$	1
						$x \vee b_1 \vee b_5$	1
						$y \vee b_3 \vee b_6$	1
						$z \vee b_4 \vee b_7$	1
$\underline{\neg x \vee \neg y}$	∞	$\underline{\neg x \vee \neg y}$	∞	$\underline{\neg x \vee \neg y}$	∞	$\neg x \vee \neg y$	∞
$x \vee \neg z$	∞	$\underline{x \vee \neg z}$	∞	$\underline{x \vee \neg z}$	∞	$x \vee \neg z$	∞
$\underline{y \vee \neg z}$	∞	$y \vee \neg z$	∞	$\underline{y \vee \neg z}$	∞	$y \vee \neg z$	∞
		$CNF(b_1 + b_2 = 1)$	∞	$\underline{CNF(b_1 + b_2 = 1)}$	∞	$CNF(b_1 + b_2 = 1)$	∞
			∞	$\underline{CNF(b_3 + b_4 = 1)}$	∞	$CNF(b_3 + b_4 = 1)$	∞
						$CNF(b_5 + b_6 + b_7 = 1)$	∞
$cost = 0$		$cost = 1$		$cost = 3$		$cost = 4$	

Figure 4.2: Application of the WPM1 algorithm on the Weighted CNF $\{(x, 1), (y, 2), (z, 3), (\neg x \vee \neg y, \infty), (x \vee \neg z, \infty), (y \vee \neg z, \infty)\}$. UNSAT cores of each stage are underlined.

4.2.1 Eager and Lazy SMT Approaches

There are two types of procedures for solving SMT, the so-called eager and the lazy approaches. In the eager approach, the input formula is fully translated into a propositional CNF formula, in a single step and preserving satisfiability, which is then checked whether is satisfiable or not by a SAT solver. Sophisticated ad-hoc translations have been developed for several theories, but still, on many practical problems either the translation process or the SAT solver run out of time or memory [dMR04].

Currently most successful SMT solvers are based on a lazy approach, the integration of a SAT solver and a T-solver, that is, a decision procedure for the given theory T. In this approach, while the SAT solver is in charge of the Boolean component of reasoning, the T-solver deals with sets of atomic constraints (literals) in T. The basic idea is to let the T-solver analyze the partial truth assignment that the SAT solver is building, and warn about conflicts with theory T (T-inconsistency). This way, we are hopefully getting the best of both worlds: in particular, the efficiency of the SAT solver for the Boolean reasoning and the efficiency of special-purpose algorithms inside the T-solver for the theory reasoning. This approach is usually orders of magnitude faster than the eager approach. It is called lazy because the theory information is only used from time to time, when checking the consistency of the truth assignment with theory T.

Algorithm 4 shows a simplified version of an enumeration-based T-satisfiability procedure (from [BCF$^+$06]) where T-consistency is only checked for total Boolean assignments. We refer the reader to [Seb07] for a survey on the lazy SMT approach.

The algorithm enumerates the Boolean models of the propositional abstraction of the SMT formula φ and checks for their satisfiability in the theory T.

- The function *Atoms* takes a quantifier-free SMT formula φ and returns the set of atoms which occur in φ, where an atom is either a propositional variable or an expression of

Algorithm 4 Bool+T Algorithm

Input: φ : SMT formula
Output: Satisfiability of φ

```
A^p ← T2B(Atoms(φ));
φ^p ← T2B(φ);
while Bool-satisfiable(φ^p) do
  μ^p ← pick_total_assignment(A^p, φ^p);
  μ ← B2T(μ^p);
  (ρ, π) ← T-satisfiable(μ);
  if ρ = sat then
    return sat;
  else
    φ^p ← φ^p ∧ ¬T2B(π);
  end if;
end while
return unsat;
```

theory T.

- The function $T2B$ maps propositional variables to themselves, and ground atoms into fresh propositional variables, and is homomorphic with respect to Boolean operators and set inclusion. φ^p is initialized to be the propositional abstraction of φ using $T2B$.
- The function $B2T$ is the inverse of $T2B$.
- μ^p denotes a propositional assignment as a set (conjunction) of propositional literals.
- The function *pick_total_assignment* returns a total assignment to the propositional variables in φ^p. In particular, it assigns a truth value to all variables in A^p.
- The function *T-satisfiable* checks if a set of conjuncts μ is *T-satisfiable*, i.e., if there is a model for $T \cup \mu$, returning (`sat`,$\emptyset$) in the positive case and (`unsat`,π) otherwise, being $\pi \subseteq \mu$ a *T-unsatisfiable* set (the theory conflict set). Note that the negation of the propositional abstraction of π is added to φ^p in case of `unsat` (learning).

We illustrate Algorithm 4 with Example 16.

Example 16 *Consider the following SMT formula, expressed as a set of clauses, where T is assumed to be the theory of linear integer arithmetic:*

$$\begin{aligned}\varphi = \{&\neg(x > 0) \vee a \vee b,\\ &\neg a \vee \neg b,\\ &\neg(x + 1 < 0) \vee a,\\ &\neg b \vee \neg(y = 1)\}\end{aligned}$$

Then $\{x > 0, a, b, x + 1 < 0, y = 1\}$ is its set of atoms and

$$A^p = \{p_{(x>0)}, a, b, p_{(x+1<0)}, p_{(y=1)}\}$$

is the Booleanization of this set, where $p_{(x>0)}, p_{(x+1<0)}$ *and* $p_{(y=1)}$ *are three fresh propositional variables corresponding to the arithmetic atoms* $x > 0, x + 1 < 0$ *and* $y = 1$*, respectively. The propositional abstraction of* φ *is then the following Boolean formula:*

$$\begin{aligned}\varphi^p = \{&\neg p_{(x>0)} \vee a \vee b,\\ &\neg a \vee \neg b,\\ &\neg p_{(x+1<0)} \vee a,\\ &\neg b \vee \neg p_{(y=1)}\}\end{aligned}$$

It is not hard to see that φ^p *is satisfiable. Suppose that pick_total_assignment(*A^p*,*φ^p*) returns us the following Boolean model for* φ^p*:*

$$\mu^p = \{p_{(x>0)}, a, \neg b, p_{(x+1<0)}, \neg p_{(y=1)}\}$$

Now we need to check the T*-satisfiability of* $B2T(\mu^p)$*. Since we are interested in checking the consistency of the current Boolean assignment with theory* T*, here we only need to take into account the literals corresponding to the theory, i.e., we have to check the* T*-satisfiability of* $\{x > 0, x + 1 < 0, \neg(y = 1)\}$*. This is obviously* T*-unsatisfiable, so we get a subset of* T*-inconsistent literals from the* T*-solver, e.g.,* $\pi = \{x > 0, x + 1 < 0\}$*, and we extend* φ^p *with the learned clause, namely* $\neg p_{(x>0)} \vee \neg p_{(x+1<0)}$*. Then the search starts again.*

In practice, the enumeration of Boolean models is carried out by means of efficient implementations of the DPLL algorithm [ZM02], where the partial assignments μ^p are incrementally built. These systems benefit of the spectacular progress in performance from SAT solvers in the last decade, achieved thanks to better implementation techniques and conceptual enhancements, which have been partially described in Section 4.1.1. Adaptations of SAT techniques to the SMT framework have been described in [SS06].

An important improvement consists on checking the T-satisfiability of partial assignments, in order to anticipate possible conflicts. Theory deduction can be used to reduce the search space by explicitly returning truth values for unassigned literals, as well as constructing and learning implications. The deduction capability is a very important aspect of theory solvers, since getting short explanations (conflict sets) from the theory solver is essential in order to keep the learned lemmas as short as possible. Apart from saving memory space, shorter lemmas will allow for more pruning in general.

In the approach presented so far, the T-solver provides information only after a T-inconsistent partial assignment has been generated. In this sense, the T-solver is used only to validate the search a posteriori, not to guide it a priori. In order to overcome this limitation, the T-solver can also be used to detect literals l occurring in φ such that $M \models_T l$, where M is a partial assignment of φ. This is called theory propagation.

Finally, as it happens in SAT solving, most SMT solvers do restart periodically in order to try to explore easier successful branches.

4.2.2 Theories and Logics

The Satisfiability Modulo Theories Library (SMT-LIB) [BST10a] has the goal of establishing a common standard for the specification of benchmarks and of background theories, as well as to establish a library of benchmarks for SMT. The Satisfiability Modulo Theories Competition (SMT-COMP) is an associated yearly competition for SMT solvers. Among the logics considered in the SMT-LIB there are:

- QF_UF: the quantifier-free fragment of first order logic with equality and no restrictions on the signature (hence the name UF for Uninterpreted Functions). The theory of *Equality and Uninterpreted Functions* (EUF, or simply UF) is also known as the empty theory, as far as we are concerned with first order logic with equality (i.e., with equality built-in). If Γ is a set of equalities and Δ is a set of disequalities, then the satisfiability of $\Gamma \cup \Delta$ in QF_UF can be determined as follows [NO80]:

  - Let τ be the set of terms appearing in $\Gamma \cup \Delta$.
  - Let $\sim$ be the equivalence relation on τ induced by Γ i.e., its closure under reflexivity, symmetry and transitivity.
  - Let $\sim^*$ be the congruence closure of $\sim$, obtained by closing $\sim$ with respect to the congruence axiom $\overline{s} = \overline{t} \rightarrow f(\overline{s}) = f(\overline{t})$, where $\overline{s}$ and $\overline{s}$ are vectors of symbols.
  - $\Gamma \cup \Delta$ is satisfiable iff for each $s \neq t \in \Delta, s \not\sim^* t$.

  It is possible (and sometimes preferable) to eliminate all uninterpreted function symbols by means of Ackermann's reduction [Ack54]. In Ackermann's reduction, each application $f(a)$ is replaced by a variable f_a, and for each pair of applications $f(a)$, $f(b)$ the formula $a = b \rightarrow f_a = f_b$ is added, i.e., the single theory axiom $x = y \rightarrow f(x) = f(y)$ of the theory becomes instantiated as necessary.

- *Linear Arithmetic* over the integers (QF_LIA) or the reals (QF_LRA). Closed quantifier-free formulas with Boolean combinations of inequations between linear polynomials over integer (real) variables, e.g., $(3x + 4y \geq 7) \rightarrow (z = 3)$ where x, y and z are integer variables. These inequalities can be placed in a normal form $c_0 + \sum_{i=1}^{n} c_i * x_i \leq 0$, where each c_i is a rational constant and the variables x_i are integer (real) variables. The most common approaches for solving linear arithmetic (real and integer variables) are based on the Simplex method. A description of a QF_LIA and a QF_LRA solver can be found in [DdM06a].

- *Difference Logic* over the integers (QF_IDL) or the reals (QF_RDL). Fragment of linear arithmetic in which arithmetic atoms are restricted to have the form $x - y \bowtie k$, where x and y are numeric (integer or real) variables, k is a numeric (integer or real) constant and $\bowtie \in \{=, <, >, \leq, \geq\}$. In the usual solving method, first of all, the atoms are rewritten in terms of $\leq$. Then, the resulting atoms can be represented as a weighted directed graph with variables as vertices and edges from x to y labeled with k for every atom $x - y \leq k$. A formula is unsatisfiable iff there exists a path $x_1 \xrightarrow{k_1} x_2 \ldots x_n \xrightarrow{k_n} x_1$ such that $k_1 + k_2 + \cdots + k_n < 0$. A description of a QF_RDL solver can be found in [NO05].

- *Non-linear Arithmetic* over the integers (QF_NIA) or the reals (QF_NRA). Quantifier free integer or real arithmetic with no linearity restrictions, i.e., with clauses like $(3xy > 2 + z^2) \vee (3xy = 9)$ where x, y and z are variables. A possible technique to check the satisfiability of these formulas is to transform the problem into a linear approximation [BLO+12].

- *Arrays* (QF_AX). Closed quantifier-free formulas over the theory of arrays with extensionality. The signature of this theory consists of two interpreted function symbols: *read*, used to retrieve the element stored at a certain index of the array, and *write*,

used to modify an array by updating the element stored at a certain index. The axioms of the theory of arrays are:

$$\begin{array}{l} \forall a : Array, \forall i, j : Index, \forall x : Value \\ \quad i = j \quad \Rightarrow \quad read(write(a, i, x), j) = x \\ \quad i \neq j \quad \Rightarrow \quad read(write(a, i, x), j) = read(a, j) \end{array}$$

Finally, the extensionality axiom states that two arrays that coincide on all indices are indeed equal:

$$\begin{array}{l} \forall a, b : Array \\ \quad (\forall i : Index \;\; read(a, i) = read(b, i)) \quad \Rightarrow \quad a = b \end{array}$$

A possible approach to decide the satisfiability of ground literals in this theory is to transfer the atoms to the Equality and Uninterpreted Functions theory. Other approaches are based on a careful analysis of the problem that allows to infer, for each array, which are the relevant indices and which values are stored at these indices of the array [SBDL01, BNO+08a].

- *Bit vectors* (QF_BV). Closed quantifier-free formulas over the theory of fixed-size bit vectors. Bit vectors are normally used for representing memory contents. Common operations are: extraction of a sequence of bits, concatenation, arithmetic operations ($+$, $-$, $*$, ...), bit-wise operations (and, or, not, ...), etc. State-of-the-art methods for checking the satisfiability of a given bit vector formula are based on reduction to SAT (bit-blasting). Each bit vector is encoded into a set of Boolean variables and the operators are encoded into logical circuits [BKO+07].

- *Other theories*. In the literature we can find some other theories of interest not considered in the SMT-LIB, such as the Alldifferent theory [BM10] and the teory of costs [CFG+10].

The expressivity of each of these logics has its corresponding computational price. Checking consistency of a set of IDL constraints has polynomial time complexity whereas checking consistency of a set of LIA constraints is NP-complete.

Combination of Theories

Many natural SMT problems contain atoms from multiple theories. When dealing with two or more theories, a standard approach is to handle the integration of the different theories by performing some sort of search on the equalities between their shared (or *interface*) variables. First of all, formulas are purified by replacing terms with fresh variables, so that each literal only contains symbols belonging to one theory. For example,

$$a(1) = x + 2$$

is translated into

$$\begin{array}{l} a(v_1) = v_2 \\ \quad v_1 = 1 \\ \quad v_2 = x + 2 \end{array}$$

where the first literal belongs to UF, and the last two to LIA. Variables v_1 and v_2 are then called *interface variables*, as they appear in literals belonging to different theories. An *interface equality* is an equality between two interface variables. All theory combination schemata, e.g., Nelson-Oppen [NO79], Shostak [Sho84], or Delayed Theory Combination (DTC) [BBC$^+$06], rely to some point on checking equality between interface variables, in order to ensure mutual consistency between theories. This may imply to assign a truth value to all the interface equalities. Since the number of interface equalities is given by $|\mathcal{V}| \cdot (|\mathcal{V}| - 1)/2$, where $|\mathcal{V}|$ is the number of interface variables, the search space may be enlarged in a quadratic factor in the number of interface variables.

In the case of combining UF with another theory T, an alternative approach is to eliminate the uninterpreted function symbols by means of Ackermann's reduction [Ack54], and then solving the resulting SMT problem only with theory T. However, this has the same disadvantage as theory combination since the number of additional literals is quadratic in the size of the input and, in fact, as shown in [BCF$^+$06], there is no clear winner between DTC and Ackermannization.

4.2.3 (Weighted) MaxSMT Algorithms

In contrast to (Weighted) MaxSAT, not too much work has been done in (Weighted) MaxSMT (defined in 2.4.1) mostly because SMT is a newer field compared to SAT. A few years ago only two SMT solvers, Z3 and Yices1, can deal with (Weighted) MaxSMT. While the first one only supports MaxSAT by means of an implementation of the PM1 algorithm, the second one supports Weighted MaxSMT by means of an implementation of a non exact and incomplete algorithm. Currently, there is a bit more work done. For instance, using the theory of costs [CFG$^+$10] as a background theory of the SMT solver, combining a MaxSAT solver and an SMT solver to deal with Weighted MaxSMT [CGSS13b]or an extension of Z3 for optimization and Weighted MaxSMT called νZ [BPF15].

Chapter 5

A comparison of CP and SMT solvers on standard CSP benchmarks

There exists a lot of languages for modelling CSPs, but one of the most used, which could be considered as an standard, is MiniZinc [Nic07]. MiniZinc is a high-level declarative language for modelling CSPs, which is reformulated into a lower-level solver input language called FlatZinc.

In this chapter we introduce `fzn2smt`, a tool for translating FlatZinc instances of CSPs to the standard SMT-LIB language (version 1.2). We provide extensive performance comparisons between state-of-the-art SMT solvers and most of the available FlatZinc solvers on standard FlatZinc problems.

We want to remark that `fzn2smt` has been developed in parallel with `WSimply`, but the idea behind both tools is a bit different. The `fzn2smt` tool has been developed with the purpose of comparing the performance of SMT solvers with respect to other CP solvers. To be fair the systems must read the same models, which is done using the FlatZinc language. In addition, we want to show and demonstrate that SMT solvers are competitive with state-of-the-art CP solvers. In contrast, the idea behind `WSimply`, which shares a similar purpose, is to have a complete control of the reformulation from a high-level language into SMT, which allows us to easily extend the language, for instance by introducing meta-constraints (as we will see in next chapter), but it also allows us to easily integrate new SMT solving algorithms, for instance to perform optimization with SMT, e.g. with weighted clauses.

This chapter is based on results from [BPSV12] and it is organised as follows. We will first introduce the MiniZinc and FlatZinc languages, then we will present the architecture of `fzn2smt`, the reformulation from FlatZinc into SMT, experimental results and analysis of the performance of `fzn2smt` with state-of-the-art CP solvers and finally, we present some conclusions.

5.1 MiniZinc and FlatZinc

MiniZinc [Nic07] aims to be a standard language for specifying CSPs (with or without optimization) over Booleans, integers and real numbers. It is a mostly declarative constraint

modelling language, although it provides some facilities such as *annotations* for specifying, e.g., search strategies, that can be taken into account by the underlying solver. One of the most appealing features of MiniZinc is that it can be easily mapped onto different existing solvers, by previously compiling its model and data files into FlatZinc instances. FlatZinc is a low-level solver input language, for which there exist front-ends for several solvers, such as Gecode [SLT10], ECL^{i}PS^e [AW07], SICStus Prolog [Sic10], JaCoP [Jac10] and SCIP [Sci10], apart from a number of solvers developed by the G12 research team.

Example 17 *MiniZinc instance of the well-known Job-shop problem, where we want to schedule n jobs of different durations, using m identical machines, running only one job per machine at the same time and minimizing the total length of the schedule. In this toy instance we use 2 jobs and 2 machines. We will use this example later on to illustrate the translation process of* `fzn2smt`*.*

Data file:

```
size = 2; d = [| 2,5 | 3,4 |];
```

Model file:

```
int: size; % size of problem, number of jobs and machines
array [1..size,1..size] of int: d; % task durations
int: total = sum(i,j in 1..size) (d[i,j]); % total duration
array [1..size,1..size] of var 0..total: s; % start times
var 0..total: end; % total end time

predicate no_overlap(var int:s1, int:d1, var int:s2, int:d2) =
    s1 + d1 <= s2 \/ s2 + d2 <= s1;

constraint
  forall(i in 1..size) (
    forall(j in 1..size-1) (s[i,j] + d[i,j] <= s[i,j+1]) /\
    s[i,size] + d[i,size] <= end /\
    forall(j,k in 1..size where j < k) (
      no_overlap(s[j,i], d[j,i], s[k,i], d[k,i])
  )
);
solve minimize end;
```

FlatZinc instances are simply a list of variable declarations and flat constraints (without foralls, exists and list comprehensions), plus (possibly) a variable to optimize. The MiniZinc to FlatZinc translation has two parts (see [Nic07] for details):

- *Flattening*, where several reductions (built-ins evaluation, list comprehensions unrolling, fixed array accesses replacement, etc.) are applied until a fix-point is reached.
- *Post-flattening*, where several normalisation is done. For example, for Boolean expressions that are not top-level conjunctions, each sub-expression is replaced by a new Boolean variable, and constraints equating these new variables with the sub-expressions they replaced are added (and similarly for non-linear numeric expressions).

Example 18 *This is the FlatZinc instance resulting from translating the MiniZinc instance of Example 17.*

```
var bool: BOOL____1;
var bool: BOOL____2;
var bool: BOOL____3 = true;
var bool: BOOL____4;
var bool: BOOL____5;
var bool: BOOL____6 = true;
array [1..4] of int: d = [2, 5, 3, 4];
var 5..14: end;
array [1..4] of var 0..14: s;
constraint array_bool_or([BOOL____1, BOOL____2], BOOL____3);
constraint array_bool_or([BOOL____4, BOOL____5], BOOL____6);
constraint int_lin_le([-1, 1], [end, s[2]], -5);
constraint int_lin_le([-1, 1], [end, s[4]], -4);
constraint int_lin_le([1, -1], [s[1], s[2]], -2);
constraint int_lin_le([1, -1], [s[3], s[4]], -3);
constraint int_lin_le_reif([1, -1], [s[1], s[3]], -2, BOOL____1);
constraint int_lin_le_reif([-1, 1], [s[1], s[3]], -3, BOOL____2);
constraint int_lin_le_reif([1, -1], [s[2], s[4]], -5, BOOL____4);
constraint int_lin_le_reif([-1, 1], [s[2], s[4]], -4, BOOL____5);
solve minimize end;
```

Note that arithmetic expressions have been encoded as linear constraints with the `int_lin_le` *constraint.*

The G12 MiniZinc distribution [G1210] is accompanied with a comprehensive set of benchmarks. Moreover, in the MiniZinc challenge [SBF10], which is run every year since 2008 with the aim of comparing different solving technologies on common benchmarks, all entrants are encouraged to submit two models each with a suite of instances to be considered for inclusion in the challenge.

5.2 Architecture of `fzn2smt`

Our tool is diagrammatically represented in Fig. 5.1, through the process of compiling and solving FlatZinc instances. Shaded boxes (connected by dashed lines) denote inputs and outputs, rounded corner boxes denote actions and diamond boxes denote decisions.

The input of the compiler is a FlatZinc instance which we assume to come from the translation of a MiniZinc one. Hence we are dealing with "safe" FlatZinc instances, e.g., we don't care about out of bounds array accesses. We are also assuming that all global constraints have been reformulated into FlatZinc constraints with the default encoding provided by the MiniZinc distribution.

The input FlatZinc instance is translated into SMT (in the standard SMT-LIB language v1.2) and fed into an SMT solver. As a by-product, `fzn2smt` generates the corresponding SMT instance as an output file. Due to the large number of existing SMT solvers, each one supporting different combinations of theories, the user can choose which solver to use

(default being Yices 2). The background theory is determined by `fzn2smt` depending on the constraints of the instance.

The FlatZinc language has three solving options: `solve satisfy`, `solve minimize` *obj* and `solve maximize` *obj*, where *obj* is either the name of a variable v or a subscripted array variable $v[i]$, where i is an integer literal. Since optimization is supported neither in the SMT-LIB language nor by most SMT solvers, we have naively implemented it by means of iterative calls successively restricting the domain of the variable to be optimized (as explained in detail in section 5.3.3). Notice from the diagram of Fig. 5.1 that when, after restricting the domain, the SMT solver finds that the problem is not satisfiable anymore, the last previously saved (and hence optimal) solution is recovered. Moreover, since there is no standard output format supported by SMT solvers (using the SMT-LIB language v1.2),[1] we need a specialized *recovery module* for each solver in order to translate its output to the FlatZinc output format. Currently, `fzn2smt` can recover the output from Yices 1.0.28 [DdM06b], Yices 2 Prototype, Barcelogic 1.3 [BNO+08b], Z3.2 [dMB08] and MathSat 5 [BCF+08] SMT solvers.

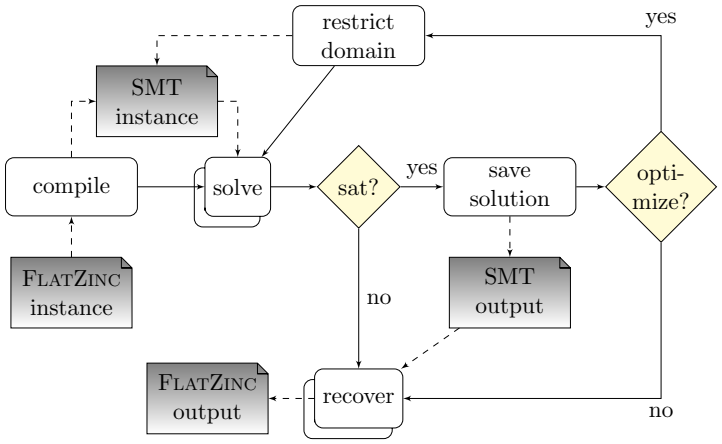

Figure 5.1: The compiling and solving process of `fzn2smt` using linear search.

5.3 Reformulation

Since we are translating FlatZinc instances into SMT, we have to keep in mind two important considerations: on the one hand we have a much richer language than plain SAT, thanks to the theories, and this will allow for more direct translations. On the other hand, in order to take advantage of the SMT solving mechanisms, the more logical structure the SMT formula has, the better. In particular, it will be better to introduce clauses instead of expressing disjunctions arithmetically.

A FlatZinc file consists of

1. a list of constant and (finite domain) variable declarations,

2. a list of flat constraints, and

[1] There are even solvers that only return `sat`, `unsat` or `unknown`. A proposal of a standard format for solutions has been proposed in the SMT-LIB Standard v2.0 [BST10b].

3. a solve goal.

Here we describe the translation of these three basic ingredients.

5.3.1 Constant and Variable Declarations

FlatZinc has two categories of data: constants (also called parameters) and variables (typically with an associated finite domain). Data can be of any of three scalar types: Booleans, integers and floats, or of any of two compound types: sets of integers and one-dimensional $(1..n)$-indexed arrays (multi-dimensional arrays are flattened to arrays of one dimension in the translation from MiniZinc to FlatZinc). Scalar type domains are usually specified by a range or a list of possible values. Our translation of FlatZinc constants and variables goes as follows:

- Scalar type constant names are always replaced by their corresponding value.
- Scalar type variable declarations are translated into their equivalent variable declaration in SMT, plus some constraints on the possible values of the SMT variable in order to fix the domain. For example, `var 0..14: x` is translated into the SMT declaration of the integer variable x plus the constraints $x \geq 0$, $x \leq 14$, whereas `var {1,3,7}: x` is translated into the SMT declaration of the integer variable x plus the constraint $x = 1 \vee x = 3 \vee x = 7$.

  Although SMT solvers are able to deal with arbitrarily large integers (as well as with arbitrary precision real numbers), for unrestricted domain integer variables we assume the G12 FlatZinc interpreter default domain range of `-10000000..10000000`, i.e., we add the constraints $x \geq -10000000$, $x \leq 10000000$ for every unrestricted integer variable x. This way, the results obtained by our system are consistent with the ones obtained by other tools.
- The domain of a FlatZinc set of integers is specified either by a range or by a list of integers. For this reason, we simply use an occurrence representation by introducing a Boolean variable for every possible element, which indicates whether the element belongs to the set or not. This allows for a simple translation into SMT of most of the constraints involving sets (see the **Constraints** section below).

  However, in order to be able to translate the set cardinality constraint, which implies counting the number of elements in a set, a 0/1 partner variable is needed for each introduced Boolean variable. For example, given the declaration `var set of {2,5,6}:s`, we introduce three Boolean variables s_2, s_5 and s_6, three corresponding integer variables s_{i_2}, s_{i_5} and s_{i_6}, the constraints restricting the domain of the integer variables

$$0 \leq s_{i_2},\ s_{i_2} \leq 1$$
$$0 \leq s_{i_5},\ s_{i_5} \leq 1$$
$$0 \leq s_{i_6},\ s_{i_6} \leq 1$$

  and the constraints linking the Boolean variables with their integer counterpart

$$s_2 \rightarrow s_{i_2} = 1,\ \neg s_2 \rightarrow s_{i_2} = 0$$
$$s_5 \rightarrow s_{i_5} = 1,\ \neg s_5 \rightarrow s_{i_5} = 0$$
$$s_6 \rightarrow s_{i_6} = 1,\ \neg s_6 \rightarrow s_{i_6} = 0.$$

Hence, the number of SMT variables increases linearly with the size of the domain of the set. Note also that all introduced clauses are either unary or binary, and hence facilitate Boolean unit propagation.

For the case of constant sets no variables are introduced at all. Instead, the element values of the constant set are directly used in the operations involving it, in order to obtain a simpler encoding.

- For the translation of arrays, we provide two options (which can be chosen by a command line option):[2]

  - Using uninterpreted functions: each array is translated into an uninterpreted function of the same name. For example, `array[1..3] of var 1..5:a` is translated into $a : int \mapsto int$. The domain of the elements of the array is constrained as in the scalar case, that is, $1 \leq a(1)$, $a(1) \leq 5$, $1 \leq a(2)$, $a(2) \leq 5$, $1 \leq a(3)$, $a(3) \leq 5$.
  - Decomposing the array into as many base type variables as array elements. For example, the previous array `a` would be decomposed into three integer variables a_1, a_2 and a_3, with the domain constrained analogously to before.

  In the case of constant arrays, equality is used (instead of two inequalities restricting the domain) to state the value of each element. If, moreover, an access to a constant array uses a constant index, we can simply replace that access with its corresponding value. And, if this is the case for all the accesses to an array, then there is no need for introducing an uninterpreted function or a set of base type variables to represent the array.

  Regarding the two possible encodings, the use of uninterpreted functions seems to be more natural, and allows for more compact encodings of array constraints. For example, to express that some element of the previous array a is equal to 1, we simply write

  $$\begin{gathered} 1 \leq i,\, i \leq 3 \\ a(i) = 1 \end{gathered}$$

  where i is an integer variable, whereas in the decomposition approach the same statement should be expressed as

  $$\begin{gathered} 1 \leq i,\, i \leq 3 \\ i = 1 \rightarrow a_1 = 1 \\ i = 2 \rightarrow a_2 = 1 \\ i = 3 \rightarrow a_3 = 1. \end{gathered}$$

  On the other hand, we have ruled out using the SMT theory of arrays. This theory involves *read* and *write* operations and, hence, is intended to be used for modelling state change of imperative programs with arrays. But, since it makes no sense thinking of *write* operations on arrays in the setting of CP, it suffices to translate every expression of the form $read(a, i)$ into $a(i)$, where a is an uninterpreted function. Moreover, deciding

[2] Due to our encoding of the operations on sets, arrays of sets are always decomposed into a number of sets.

satisfiability of sets of atomic constraints involving uninterpreted functions is far cheaper than using the arrays theory.

However, the uninterpreted functions approach still has the drawback of using more than one theory, namely, uninterpreted functions (UF) and linear integer arithmetic (LIA), and suffers from a non-negligible computational overhead due to theory combination. In Section 5.4 a performance comparison of the two possible encodings for arrays is given.

5.3.2 Constraints

The second and main block of a FlatZinc file is the set of constraints that a solution must satisfy. The arguments of these flat constraints can be literal values, constant or variable names, or subscripted array constants or variables $v[i]$ where i in an integer literal. A literal value can be either a value of scalar type, an explicit set (e.g., `{2,3,5}`) or an explicit array `[`$y_1, \ldots, y_k$`]`, where each array element y_i is either a non-array literal, the name of a non-array constant or variable, or a subscripted array constant or variable $v[i]$, where i in an integer literal (e.g., `[x,a[1],3]`).

We perform the following translation of constraint arguments prior to translating the constraints. Constant names are replaced by their value. Scalar type variables are replaced by their SMT counterpart. Finally, array accesses are translated depending on the chosen array treatment (see the previous section): when using uninterpreted functions, an array access $v[i]$ is translated into $v(i)$, where v is an uninterpreted function; when decomposing the array into base type variables, $v[i]$ is translated into v_i (the corresponding variable for that position of the array). We remark that, in all array accesses $v[i]$, i can be assumed to be an integer literal, i.e., i cannot be a variable, since all variable subscripted array expressions are replaced by `array_element` constraints during the translation from MiniZinc to FlatZinc. Moreover, we don't need to perform array bounds checks, because this is already done by the MiniZinc to FlatZinc compiler.

In the following we describe the translation of the constraints, that we have categorized into *Boolean constraints*, *Integer constraints*, *Float constraints*, *Array constraints* and *Set constraints*.

- *Boolean constraints* are built with the common binary Boolean operators (*and*, *or*, *implies*, ...) and the relational operators ($<$, $\leq$, $=$, ...) over Booleans. All of them have their counterpart in the SMT-LIB language, and hence have a direct translation.

  There is also the `bool2int(a,n)` constraint, which maps a Boolean variable into a 0/1 integer. We translate it into $(a \rightarrow n = 1) \wedge (\neg a \rightarrow n = 0)$.

- *Integer constraints* are built with the common relational constraints over integer expressions (hence they are straightforwardly translated into SMT). They also include some named constraints. Here we give the translation of some representative ones.

  The constraint

  $$\texttt{int\_lin\_eq}([\texttt{c}_1, \ldots, \texttt{c}_\texttt{n}], [\texttt{v}_1, \ldots, \texttt{v}_\texttt{n}], \texttt{r})$$

  where $\texttt{c}_1, \ldots, \texttt{c}_\texttt{n}$ are integer constants and $\texttt{v}_1, \ldots, \texttt{v}_\texttt{n}$ are integer variables or constants, means, and is translated as

  $$\sum_{i \in 1..n} c_i v_i = r.$$

The minimum constraint `int_min(a,b,c)`, meaning $min(a, b) = c$, is translated as

$$(a > b \rightarrow c = b) \wedge (a \leq b \rightarrow c = a).$$

The absolute value constraint `int_abs(a,b)`, meaning $|a| = b$, is translated as

$$(a = b \vee -a = b) \wedge b \geq 0.$$

The constraint `int_times(a,b,c)`, that states $a \cdot b = c$, can be translated into a set of linear arithmetic constraints under certain circumstances: if either a or b are (uninstantiated) finite domain variables, we linearize this constraint by conditionally instantiating the variable with the smallest domain, e.g.,

$$\bigwedge_{i \in \mathcal{D}om(a)} (i = a \rightarrow i \cdot b = c).$$

In fact, we do it better (i.e., we do not necessarily expand the expression for all the values of $\mathcal{D}om(a)$) by considering the domain bounds of b and c, and narrowing accordingly the domain of a.

Since it is better to use the simplest logic at hand, we use linear integer arithmetic for the translation if possible, and non-linear integer arithmetic otherwise. Hence, only in the case that a and b are unrestricted domain variables we translate the previous constraint as $a \cdot b = c$ and label the SMT instance to require QF_NIA (*non-linear integer arithmetic logic*).

- *Float constraints* are essentially the same as the integer ones, but involving float data. Hence, the translation goes in the same way as for the integers, except that the inferred logic is QF_LRA (*linear real arithmetic*).

- *Array constraints.* The main constraint dealing with arrays is `element`, which restricts an element of the array to be equal to some variable.

  As an example, `array_var_int_element(i, a, e)` states $i \in 1..n \wedge a[i] = e$, where n is the size of a. The translation varies depending on the representation chosen for arrays (see the previous section):

  - In the uninterpreted functions approach, the translation is

    $$1 \leq i \wedge i \leq n \wedge a(i) = e,$$

    where a is the uninterpreted function symbol representing the array `a`.
  - In the decomposition approach, the translation is

    $$1 \leq i \wedge i \leq n \wedge \left(\bigwedge_{j \in 1..n} i = j \rightarrow a_j = e \right).$$

  Constraints such as `array_bool_and`, `array_bool_or` or `bool_clause`, dealing with arrays of Booleans, are straightforwardly translated into SMT.

- *Set constraints.* These are the usual constraints over sets. We give the translation of some of them.

  The constraint `set_card(s,k)`, stating $|s| = k$, is translated by using the 0/1 variables introduced for each element (see the previous section) as $\sum_{j \in \mathcal{D}om(s)} s_{i_j} = k$.

  The constraint `set_in(e,s)`, stating $e \in s$, is translated depending on whether e is instantiated or not. If e is instantiated then `set_in(e,s)` is translated as s_e if e is in the domain of s (recall that we are introducing a Boolean variable s_e for each element e in the domain of s), and as *false* otherwise. If `e` is not instantiated, then we translate the constraint as

  $$\bigvee_{j \in \mathcal{D}om(s)} (e = j) \wedge s_j.$$

  For constraints involving more than one set, one of the difficulties in their translation is that the involved sets can have distinct domains. For example, the constraint `set_eq(a,b)`, stating $a = b$, is translated as

  $$\left(\bigwedge_{j \in \mathcal{D}om(a) \cap \mathcal{D}om(b)} a_j = b_j \right) \wedge \left(\bigwedge_{j \in \mathcal{D}om(a) \setminus \mathcal{D}om(b)} \neg a_j \right) \wedge \left(\bigwedge_{j \in \mathcal{D}om(b) \setminus \mathcal{D}om(a)} \neg b_j \right).$$

  And the constraint `set_diff(a,b,c)`, which states $a \setminus b = c$, is translated as

  $$\left(\bigwedge_{j \in (\mathcal{D}om(a) \setminus \mathcal{D}om(b)) \cap \mathcal{D}om(c)} a_j = c_j \right) \wedge \left(\bigwedge_{j \in (\mathcal{D}om(a) \setminus \mathcal{D}om(b)) \setminus \mathcal{D}om(c)} \neg a_j \right) \wedge \left(\bigwedge_{j \in \mathcal{D}om(c) \setminus \mathcal{D}om(a)} \neg c_j \right) \wedge \left(\bigwedge_{j \in \mathcal{D}om(a) \cap \mathcal{D}om(b) \cap \mathcal{D}om(c)} \neg c_j \right) \wedge \left(\bigwedge_{j \in (\mathcal{D}om(a) \cap \mathcal{D}om(b)) \setminus \mathcal{D}om(c)} a_j \rightarrow b_j \right).$$

  Although the translation of the set constraints seem to be convoluted, note that we are mainly introducing unit and binary clauses. We remark that when the sets are already instantiated at compilation time, some simplifications are actually made. Note also that the size of the SMT formula increases linearly on the size of the domains of the sets.

Finally, let us mention that almost all FlatZinc constraints have a reified counterpart. For example, in addition to the constraint `int_le(a,b)`, stating $a \leq b$, there is a constraint `int_le_reif(a,b,r)`, stating $a \leq b \leftrightarrow r$, where a and b are integer variables and r is a Boolean variable. In all these cases, given the translation of a constraint, the translation of its reified version into SMT is direct.

5.3.3 Solve Goal

A FlatZinc file must end with a solve goal, which can be of one of the following forms: `solve satisfy`, for checking satisfiability and providing a solution if possible, or `solve minimize` *obj* or `solve maximize` *obj*, for looking for a solution that minimizes or maximizes, respectively, the value of *obj* (the objective function), where *obj* is either a variable v or a subscripted array variable $v[i]$, where i is an integer literal. Although search annotations can be used in the solve goal, they are currently ignored in our tool.

Algorithm 5 Minimization

Input: $SMT\_inst : SMTinstance$;
$var : int$; // variable to minimize
$inf, sup : int$; // bounds of the variable to minimize
$t : int$; // threshold for the hybrid strategy
$bs : \{linear, binary, hybrid\}$; // bounding strategy
Output: $\langle int,\ sat|unsat\rangle$
$\langle sol, status\rangle$:= SMT_solve(bound(SMT_inst, var, inf, sup));
if $status$ = unsat **then**
  return $\langle\infty, unsat\rangle$;
else
  $sup := sol - 1$;
  while $inf \leq sup$ **do**
    if $(sup - inf \leq t \ \wedge \ bs = hybrid)$ **then**
      bs := linear;
    end if
    if bs = linear **then**
      $med := sup$;
    else
      $med := \lceil (inf + sup)/2 \rceil$;
    end if
    $\langle new\_sol, status\rangle$:= SMT_solve(bound(SMT_inst, var, inf, med));
    if $status$ = unsat **then**
      if bs = linear **then**
        return $\langle sol, sat\rangle$;
      else
        $inf := med + 1$;
      end if
    else
      $sol := new\_sol$;
      $sup := sol - 1$;
    end if
  end while
  return $\langle sol, sat\rangle$;
end if

When the `satisfy` option is used, we just need to feed the selected SMT solver with the SMT file resulting from the translation explained in the previous sections (see Example 19 to

see a complete SMT-LIB instance generated by `fzn2smt`). Thereafter the recovery module will translate the output of the SMT solver to the FlatZinc format (as explained in section 5.2).

We have implemented an ad hoc search procedure in order to minimize or maximize the objective function. This procedure successively calls the SMT solver with different problem instances, by restricting the domain of the variable to be optimized with the addition of constraints. We have implemented three possible bounding strategies: linear, dichotomic and hybrid. The linear bounding strategy approaches the optimum from the satisfiable side, i.e., bounding the objective function to be less than the upper bound.[3] While the dichotomic strategy simply consists of binary search optimization. Finally, the hybrid strategy makes a preliminary approach to the optimum by means of binary search and, when a (user definable) threshold on the possible domain of the variable is reached, it turns into the linear approach, again from the satisfiable side. Both the bounding strategy and the threshold for the hybrid case can be specified by the user from the command line. Algorithm 5 describes our optimization procedure, taking into account all bounding strategies, for the minimization case.

The `SMT_solve` function consists of a call to the SMT solver with an SMT instance. The `bound`(SMT_inst, var, inf, sup) function returns the SMT instance resulting from adding the bounds $var \geq inf$ and $var \leq sup$ to the SMT instance SMT_inst.

We are not keeping learnt clauses from one iteration of the SMT solver to the next since we are using the SMT solver as a black box. We have also tested an implementation of these optimization algorithms using the Yices API, which allows keeping the learnt clauses, without obtaining significantly better results.

Example 19 *Continuing Example 18, next we show the SMT-LIB instance produced by* `fzn2smt`*. For the minimization process, we should add the appropriate bounds to the objective variable* `end`*.*

```
(benchmark jobshopp.fzn.smt
  :source { Generated by fzn2smt }
  :logic QF_IDL
  :extrapreds ((BOOL____4) (BOOL____2) (BOOL____1) (BOOL____5))
  :extrafuns ((s_1_ Int) (s_2_ Int) (s_3_ Int) (s_4_ Int) (end Int))
  :formula (and
    (>= end 5)
    (<= end 14)
    (>= s_1_ 0)
    (<= s_1_ 14)
    (>= s_2_ 0)
    (<= s_2_ 14)
    (>= s_3_ 0)
    (<= s_3_ 14)
    (>= s_4_ 0)
    (<= s_4_ 14)
    (= (or  BOOL____1 BOOL____2) true)
```

[3] This allows us to eventually jump several values when we find a new solution. On the contrary, approaching from the unsatisfiable side is only possible by modifying the value of the variable to optimize in one unit at each step.

```
    (= (or  BOOL____4 BOOL____5) true)
    (<= (+ (~ end) s_2_) (~ 5))
    (<= (+ (~ end) s_4_) (~ 4))
    (<= (+ s_1_ (~ s_2_)) (~ 2))
    (<= (+ s_3_ (~ s_4_)) (~ 3))
    (= (<= (+ s_1_ (~ s_3_)) (~ 2)) BOOL____1)
    (= (<= (+ (~ s_1_) s_3_) (~ 3)) BOOL____2)
    (= (<= (+ s_2_ (~ s_4_)) (~ 5)) BOOL____4)
    (= (<= (+ (~ s_2_) s_4_) (~ 4)) BOOL____5)
  )
)
```

5.4 Experimental results

In this section we compare the performance of `fzn2smt` and that of several existing FlatZinc solvers on FlatZinc instances, and provide some possible explanations for the `fzn2smt` behaviour. We first compare several SMT solvers within `fzn2smt` and, afterwards, use the one with the best results to compare against other existing FlatZinc solvers.

We perform the comparisons on the benchmarks of the three MiniZinc challenge competitions (2008, 2009 and 2010),consisting of a total of 294 instances from 32 problems. These benchmarks consist of a mixture of puzzles, planning, scheduling and graph problems. Half of the problems are optimization problems, whilst the other half are satisfiability ones.

We present several tables that, for each solver and problem, report the accumulated time for the solved instances and the number of solved instances (within parenthesis). The times are the sum of the translation time, when needed (e.g., `fzn2smt` translates from FlatZinc to the SMT-LIB format), plus the solving time. We indicate in boldface the cases with more solved instances, breaking ties by total time. The experiments have been executed on an Intel Core i5 CPU at 2.66 GHz, with 6GB of RAM, running 64-bit openSUSE 11.2 (kernel 2.6.31), with a time limit of 15 minutes per instance (the same as in the competition).

5.4.1 Comparison of SMT solvers within `fzn2smt`

Here we compare the performance of several SMT solvers which are SMT-LIB 1.2 compliant, working in cooperation with `fzn2smt` v2.0.1, on the MiniZinc challenge benchmarks. We have selected the solvers that historically have had good performance in the QF_LIA division of the annual SMT competition. These are Barcelogic 1.3 [BNO+08b], MathSAT 5 (successor of MathSAT 4 [BCF+08]), Yices 1.0.28 [DdM06b], Yices 2 Prototype (a prototype version of Yices 2), and Z3.2 [dMB08]. Linux binaries of most of these solvers can be downloaded from `http://www.smtcomp.org`.

In the executions of this subsection we have used the default options for `fzn2smt`: array expansion (see Subsection 5.3.1 and binary search for optimization (see Subsection 5.3.3).

In Table 5.1 we can observe that Yices 1.0.28 and Yices 2 are able to solve, respectively, 213 and 212 (out of 294) instances. We consider performance of Yices 2 the best of those considered because it had the best performance on 19 of the problems, far more than any of the other solvers.

Table 5.1: Accumulated time for the solved instances and the number of solved instances (within parenthesis) of state-of-the-art SMT solvers in cooperation with `fzn2smt` on the MiniZinc challenge benchmarks. Type 's' stands for satisfaction and 'o' for optimization. # stands for the number of instances.

Problem	Type	#	Barcelogic 1.3		MathSAT 5		Yices 1.0.28		Yices 2 proto		Z3.2	
debruijn-binary	s	11	0.60	(1)	0.56	(1)	**0.47**	**(1)**	0.50	(1)	0.57	(1)
nmseq	s	10	0.00	(0)	568.35	(2)	778. 07	(5)	118.74	(2)	**173.84**	**(5)**
pentominoes	s	7	0.00	(0)	291.60	(1)	**50.47**	**(2)**	168.47	(2)	314.48	(1)
quasigroup7	s	10	34.16	(2)	191.83	(5)	54.94	(5)	**20.03**	**(5)**	691.32	(4)
radiation	o	9	342.34	(1)	2202.92	(9)	**373.24**	**(9)**	1047.03	(9)	2473.27	(9)
rcpsp	o	10	0.00	(0)	315.46	(2)	920.08	(8)	**993.74**	**(9)**	1791.79	(8)
search-stress	s	3	0.84	(2)	1.05	(2)	0.86	(2)	**0.74**	**(2)**	0.84	(2)
shortest-path	o	10	**109.35**	**(9)**	484.73	(8)	658.58	(9)	1419.67	(7)	790.54	(8)
slow-convergence	s	10	405.25	(7)	426.79	(7)	269.66	(7)	**247.06**	**(7)**	291.99	(7)
trucking	o	10	254.11	(5)	31.70	(4)	**9. 50**	**(5)**	47.77	(5)	1084.97	(4)
black-hole	s	10	511.13	(1)	29.24	(1)	**3857.51**	**(9)**	892.46	(8)	765.09	(1)
fillomino	s	10	93.87	(10)	30.28	(10)	20.48	(10)	**19.99**	**(10)**	21.13	(10)
nonogram	s	10	0.00	(0)	0.00	(0)	**1656.54**	**(10)**	1546.56	(7)	0.00	(0)
open-stacks	o	10	1772.39	(5)	0.00	(0)	702.09	(6)	**707.25**	**(7)**	776.61	(6)
p1f	o	10	875.54	(8)	**86.90**	**(9)**	167.84	(9)	126.01	(9)	184.22	(9)
prop-stress	s	10	315.80	(7)	330.31	(7)	**266.11**	**(7)**	274.07	(7)	289.23	(7)
rect-packing	s	10	559.50	(5)	679.62	(10)	**104.82**	**(10)**	106.66	(10)	122.28	(10)
roster-model	o	10	98.41	(10)	51.03	(10)	53.89	(10)	**50.38**	**(10)**	56.04	(10)
search-stress2	s	10	23.19	(10)	14.63	(10)	9.43	(10)	**7.90**	**(10)**	10.55	(10)
still-life	o	4	30.64	(3)	132.51	(4)	128.71	(4)	**62.18**	**(4)**	173.82	(4)
vrp	o	10	0.00	(0)	0.00	(0)	0.00	(0)	0.00	(0)	0.00	(0)
costas-array	s	5	0.00	(0)	0.00	(0)	675.23	(1)	**664.13**	**(2)**	628.83	(1)
depot-placement	o	15	2480.74	(5)	2496.69	(10)	613.53	(15)	**295.78**	**(15)**	2073.93	(15)
filter	o	10	24.02	(6)	38.01	(6)	24.28	(6)	**17.88**	**(6)**	22.42	(6)
ghoulomb	o	10	0.00	(0)	0.00	(0)	75.87	(1)	**2545.02**	**(6)**	508.19	(2)
gridColoring	o	5	4.82	(2)	202.18	(3)	857.74	(3)	**38.15**	**(3)**	51.94	(3)
rcpsp-max	o	10	85.04	(1)	238.34	(2)	533.87	(4)	475.14	(4)	**383.73**	**(4)**
solbat	s	15	0.00	(0)	1721.73	(15)	341.04	(15)	**141.48**	**(15)**	1339.75	(15)
sugiyama2	o	5	79.38	(5)	21.31	(5)	10.26	(5)	**8.64**	**(5)**	9.56	(5)
wwtp-random	s	5	61.08	(5)	29.72	(5)	17.71	(5)	**15.21**	**(5)**	16.56	(5)
wwtp-real	s	5	107.57	(5)	39.77	(5)	17.97	(5)	**14.47**	**(5)**	17.09	(5)
bacp	o	15	1753.26	(14)	232.22	(15)	75.16	(15)	**64.17**	**(15)**	97.51	(15)
Total		294	10023	(129)	10889	(168)	**13325**	**(213)**	12137	(212)	15162	(192)

Array encodings

In Table 5.2 we compare the performance of using array decomposition versus uninterpreted functions as array encodings. We only give the results for Yices 2 proto, which is the solver with the best performance in the executions of Table 5.1 (where array decomposition was used as default option).

Table 5.2: Accumulated time for the solved instances and the number of solved instances (within parenthesis) of Yices 2 solver using array decomposition vs uninterpreted functions (UF). Type 's' stands for satisfaction and 'o' for optimization. # stands for the number of instances.

Problem	Type	#	Decomposition		UF	
debruijn-binary	s	11	**0.50**	**(1)**	0.74	(1)
nmseq	s	10	118.74	(2)	**83.51**	**(2)**
pentominoes	s	7	**168.47**	**(2)**	197.22	(1)
quasigroup7	s	10	**20.03**	**(5)**	336.59	(5)
radiation	o	9	**1047.03**	**(9)**	2360.85	(9)
rcpsp	o	10	**993.74**	**(9)**	695.84	(8)
search-stress	s	3	**0.74**	**(2)**	0.84	(2)
shortest-path	o	10	**1419.67**	**(7)**	1068.03	(4)
slow-convergence	s	10	**247.06**	**(7)**	522.42	(3)
trucking	o	10	47.77	(5)	**39.67**	**(5)**
black-hole	s	10	**892.46**	**(8)**	0.00	(0)
fillomino	s	10	19.99	(10)	**19.21**	**(10)**
nonogram	s	10	**1546.56**	**(7)**	0.00	(0)
open-stacks	o	10	**707.25**	**(7)**	1729.37	(5)
p1f	o	10	**126.01**	**(9)**	25.49	(8)
prop-stress	s	10	274.07	(7)	**262.95**	**(7)**
rect-packing	s	10	**106.66**	**(10)**	112.10	(10)
roster-model	o	10	**50.38**	**(10)**	51.01	(10)
search-stress2	s	10	**7.90**	**(10)**	9.74	(10)
still-life	o	4	**62.18**	**(4)**	97.39	(4)
vrp	o	10	0.00	(0)	0.00	(0)
costasArray	s	5	**664.13**	**(2)**	151.60	(1)
depot-placement	o	15	**295.78**	**(15)**	2651.71	(10)
filter	o	10	**17.88**	**(6)**	23.12	(6)
ghoulomb	o	10	**2545.02**	**(6)**	707.74	(2)
gridColoring	o	5	**38.15**	**(3)**	335.10	(3)
rcpsp-max	o	10	**475.14**	**(4)**	498.89	(4)
solbat	s	15	**141.48**	**(15)**	567.69	(15)
sugiyama2	o	5	**8.64**	**(5)**	9.04	(5)
wwtp-random	s	5	**15.21**	**(5)**	34.67	(5)
wwtp-real	s	5	**14.47**	**(5)**	28.82	(5)
bacp	o	15	**64.17**	**(15)**	77.85	(15)
Total		294	**12137**	**(212)**	12699	(175)

As shown in Table 5.2, the decomposition approach clearly outperforms the uninterpreted functions approach on FlatZinc instances from the MiniZinc distribution. We have also tested other SMT solvers than Yices, obtaining similar results. This apparently strange behaviour is better understood when looking at how SMT solvers deal with uninterpreted functions (see Section 4.2.2) and, in particular, how this behaves on the instances generated by our tool.

It is worth noting that current SMT solvers have been designed mainly to deal with verification problems, where there are few input parameters and almost all variable values are undefined. In such problems, uninterpreted functions are typically used to abstract pieces of code and, hence, their arguments are variables (or expressions using variables). Moreover, the number of such abstractions is limited. This makes Ackermannization feasible in practice. On the contrary, in the instances we are considering, we have the opposite situation: a lot of parameters and a few decision variables. In particular, most arrays are parameters containing data. For example, in a scheduling problem, a FlatZinc array containing durations of tasks, such as

```
array[1..100] of int:  d = [2,5,...,4];
```

could be expressed using an SMT uninterpreted function as follows:

$$\begin{gathered} d(1) = 2 \\ d(2) = 5 \\ \dots \\ d(100) = 4. \end{gathered}$$

Similarly, for an undefined array containing, e.g., starting times, such as

```
array[1..100] of var 0..3600:  s;
```

we could use an uninterpreted function, and state its domain as follows:

$$\begin{gathered} 0 \leq s(1),\ s(1) \leq 3600 \\ 0 \leq s(2),\ s(2) \leq 3600 \\ \dots \\ 0 \leq s(100),\ s(100) \leq 3600. \end{gathered}$$

In any case, lots of distinct uninterpreted function applications appear, and Ackermannization results in a quadratic number of formulas like $1 = 2 \rightarrow f_1 = f_2$, which are trivially true since the antecedent is false. Although difficult to determine because we are using each SMT solver as a black box, we conjecture that this is not checked in the Ackermannization process since, as said before, uninterpreted functions are expected to have variables in the arguments.

Finally, although the decomposition approach exhibits better performance than the uninterpreted functions approach, we have decided to maintain both options in our system. The main reason is that the uninterpreted functions approach allows for more compact and natural representations and, hopefully, can lead to better results in the future if Ackermannization is adapted accordingly.

Bounding Strategy

In this section, we test the performance of Yices 2 with the different bounding strategies described in Subsection 5.3.3 for optimization problems. For the hybrid strategy we have used the default threshold of 10 units for switching from the binary to the linear approximation strategy. Experiments with larger threshold have not yielded better results.

Table 5.3: Accumulated time for the solved instances and the number of solved instances (within parenthesis) of Yices 2 solver using different optimization search strategies. Type 's' stands for satisfaction and 'o' for optimization. # stands for the number of instances.

Problem	#	Binary		Hybrid		Linear	
radiation	9	**1047.03**	**(9)**	1304.59	(9)	2008.77	(9)
rcpsp	10	**993.74**	**(9)**	710.77	(8)	1180.11	(5)
shortest-path	10	1419.67	(7)	1381.92	(7)	**1034.22**	**(8)**
trucking	10	47.77	(5)	41.86	(5)	**34.66**	**(5)**
open-stacks	10	707.25	(7)	**650.27**	**(7)**	691.46	(7)
p1f	10	126.01	(9)	**125.44**	**(9)**	188.14	(9)
roster-model	10	50.38	(10)	50.29	(10)	**50.16**	**(10)**
still-life	4	**62.18**	**(4)**	118.54	(4)	119.39	(4)
vrp	10	0.00	(0)	0.00	(0)	0.00	(0)
depot-placement	15	295.78	(15)	**248.19**	**(15)**	263.61	(15)
filter	10	17.88	(6)	**17.37**	**(6)**	18.34	(6)
ghoulomb	10	**2545.02**	**(6)**	2825.16	(6)	1255. 42	(3)
gridColoring	5	38.15	(3)	**17.43**	**(3)**	17.81	(3)
rcpsp-max	10	475.14	(4)	**460.08**	**(4)**	1035.02	(4)
sugiyama2	5	8.64	(5)	**8.37**	**(5)**	9.25	(5)
bacp	15	64.17	(15)	**58.03**	**(15)**	64.98	(15)
Total	153	**7898**	**(113)**	8018	(113)	7971	(108)

Table 5.3 shows that the binary and hybrid strategies perform better than the linear one in general. Both the binary and hybrid strategies are able to solve the same number of instances, but the first one spends less time globally. Nevertheless, the hybrid strategy is faster than the binary in most of the problems. And, interestingly, the linear strategy is better in three problems.

We want to remark that the linear strategy approaches the optimum from the satisfiable side. We also tried the linear strategy approaching from the unsatisfiable side but the results where a bit worse globally. This is probably due to the fact that the latter can only make approximation steps of size one whilst the former can make bigger steps when a solution is found.

5.4.2 Comparison of `fzn2smt` with other FlatZinc Solvers

In this section we compare the performance of `fzn2smt` (using Yices 2) and the following available FlatZinc solvers: Gecode, G12 and G12 lazy_fd (the solvers distributed with MiniZinc) and FznTini (a SAT based solver).

Let us remark that `fzn2smt` with Yices 2 obtained (*ex aequo* with Gecode) the golden medal in the *par* division and the silver medal in the *free* division of the MiniZinc challenge 2010, and the silver medal in the same divisions of the MiniZinc challenge 2011. It is also

Table 5.4: Accumulated time for the solved instances and the number of solved instances (within parenthesis) of `fzn2smt` and some available FlatZinc solvers. Type 's' stands for satisfaction and 'o' for optimization. # stands for the number of instances.

n	Problem	Type	#	Gecode		FznTini		G12		G12 lazy_fd		`fzn2smt`	
1	debruijn-binary	s	11	**4.46**	**(6)**	0.06	(1)	31.30	(6)	0.00	(0)	0.50	(1)
2	nmseq	s	10	**535.62**	**(8)**	2.64	(1)	927.42	(7)	0.00	(0)	118.74	(2)
3	pentominoes	s	7	**601.82**	**(7)**	89.68	(1)	848.17	(4)	466.57	(5)	168.47	(2)
4	quasigroup7	s	10	**278.73**	**(6)**	773.28	(4)	1.72	(5)	2.85	(5)	20.30	(5)
5	radiation	o	9	1112.14	(9)	4260.37	(7)	1302.79	(9)	**3.60**	**(9)**	1047.03	(9)
6	rcpsp	o	10	12.07	(5)	0.00	(0)	97.27	(5)	82.60	(8)	**993.74**	**(9)**
7	search-stress	s	3	11.30	(2)	391.71	(3)	14.66	(2)	**0.40**	**(3)**	0.74	(2)
8	shortest-path	o	10	442.49	(10)	0.00	(0)	4.27	(4)	**127.77**	**(10)**	1419.67	(7)
9	slow-convergence	s	10	**8.41**	**(10)**	62.62	(4)	95.42	(10)	154.83	(10)	247.06	(7)
10	trucking	o	10	**1.01**	**(5)**	593.23	(4)	4.12	(5)	159.84	(5)	47.77	(5)
11	black-hole	s	10	69.68	(7)	0.00	(0)	2423.48	(6)	97.69	(7)	**892.46**	**(8)**
12	fillomino	s	10	118.62	(10)	4.36	(10)	332.56	(10)	**2.59**	**(10)**	19.99	(10)
13	nonogram	s	10	1353.63	(8)	48.13	(7)	336.93	(2)	**1533.07**	**(9)**	1546.56	(7)
14	open-stacks	o	10	**169.74**	**(8)**	1325.98	(4)	299.55	(8)	0.00	(0)	707.25	(7)
15	p1f	o	10	2.57	(8)	315.65	(9)	2.40	(8)	0.00	(0)	**126.01**	**(9)**
16	prop-stress	s	10	600.80	(4)	223.52	(2)	883.08	(3)	**221.95**	**(9)**	274.07	(7)
17	rect-packing	s	10	134.36	(6)	569.57	(3)	339.80	(6)	165.58	(6)	**106.66**	**(10)**
18	roster-model	o	10	**1.04**	**(10)**	0.00	(0)	4.46	(10)	18.80	(7)	50.38	(10)
19	search-stress2	s	10	296.16	(9)	9.10	(10)	381.82	(8)	**0.08**	**(10)**	7.90	(10)
20	still-life	o	4	1.01	(3)	35.50	(3)	2.56	(3)	18.55	(3)	**62.18**	**(4)**
21	vrp	o	10	0.00	(0)	0.00	(0)	0.00	(0)	0.00	(0)	0.00	(0)
22	costas-array	s	5	**943.75**	**(4)**	405.28	(2)	423.09	(3)	145.54	(2)	664.13	(2)
23	depot-placement	o	15	1205.75	(12)	1820.57	(8)	522.81	(8)	613.51	(12)	**295.78**	**(15)**
24	filter	o	10	30.95	(1)	62.16	(7)	278.72	(1)	**2.65**	**(7)**	17.88	(6)
25	ghoulomb	o	10	0.00	(0)	0.00	(0)	246.67	(1)	**2512.92**	**(8)**	2545.02	(6)
26	gridColoring	o	5	0.48	(1)	152.71	(3)	0.51	(1)	0.10	(1)	**38.15**	**(3)**
27	rcpsp-max	o	10	42.11	(2)	0.00	(0)	30.00	(1)	596.41	(4)	**475.14**	**(4)**
28	solbat	s	15	746.31	(10)	1300.54	(14)	1540.71	(10)	311.92	(11)	**141.48**	**(15)**
29	sugiyama2	o	5	308.31	(5)	37.00	(5)	510.72	(5)	520.88	(5)	**8.64**	**(5)**
30	wwtp-random	s	5	0.03	(1)	322.21	(3)	2.79	(2)	0.00	(0)	**15.21**	**(5)**
31	wwtp-real	s	5	0.08	(3)	1239.45	(4)	0.31	(3)	71.83	(4)	**14.47**	**(5)**
32	bacp	o	15	847.78	(10)	1170.15	(5)	976.35	(10)	**28.54**	**(15)**	64.17	(15)
	Total		294	9881	(190)	15215	(124)	12866	(166)	7859	(185)	**12137**	**(212)**

fair to notice that the solver with the best performance in the MiniZinc challenges 2010 and 2011 (in all categories) was Chuffed, implemented by the MiniZinc team and not eligible for prizes.[4]

Table 5.4 shows the results of this comparison without using solver specific global constraints, which means that global constraints are decomposed into conjunctions of simpler constraints. However, search strategy annotations are enabled in all experiments and, while `fzn2smt` ignores them, the other systems can profit from these annotations.

We can observe that `fzn2smt` is the solver which is able to solve the largest number of instances, closely followed by G12 lazy_fd and Gecode. Looking at the problems separately, `fzn2smt` offers better performance in 12 cases, followed by G12 lazy_fd in 10 cases and Gecode in 9 cases.

We remark that G12 lazy_fd does not support instances with unbounded integer variables: the ones of debruijn-binary, nmseq, open-stacks, p1f, wwtp-random. We have tried to solve these instances by bounding those variables with the MiniZinc standard default limits, i.e., by setting `var -10000000..10000000 : x;` for every unrestricted integer variable x (similarly

[4]See http://www.g12.csse.unimelb.edu.au/minizinc/challenge2011/results2011.html for details.

as we have done for `fzn2smt`), but G12 lazy_fd runs out of memory. This is probably due to its use of Boolean encodings for the domains of the integer variables, as these encodings imply introducing a new Boolean variable for each element of the domain (see [OSC09]).

The plot of Figure 5.2 shows the elapsed times, in logarithmic scale, for the solved instances of Table 5.4. The instances have been ordered by its execution time in each system. The overall best system (in terms of number of solved instances) is `fzn2smt`. However `fzn2smt` is the worst system (in terms of execution time) within the first 50 solved instances and, moreover, Gecode is better along the 160 first instances, closely followed by G12 lazy_fd. It must be taken into account that the `fzn2smt` compiler is written in Java, and it generates an SMT file for each decision problem that is fed into the chosen SMT solver. Hence this can cause an overhead in the runtime that can be more sensible for the easier instances. Finally, note also that `fzn2smt` scales very well from the 50 to the 150 first solved instances. This exhibits the SMT solvers robustness.

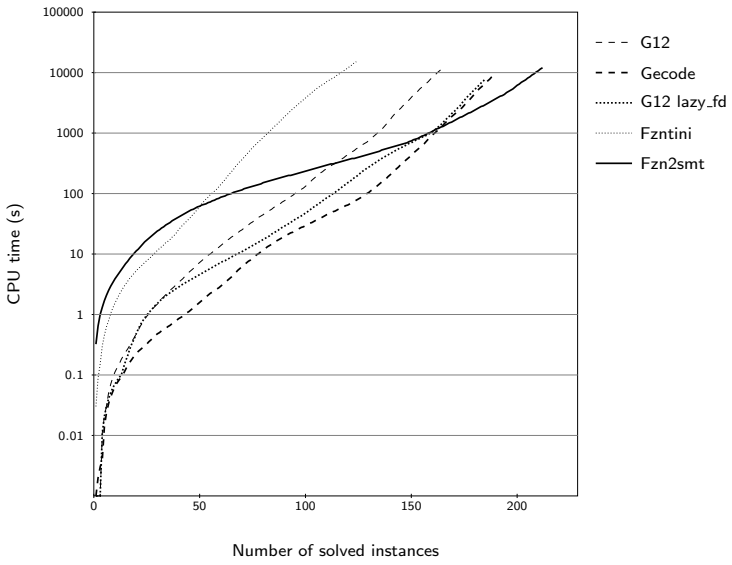

Figure 5.2: Number of solved instances and elapsed times (referred to Table 5.4).

5.4.3 FlatZinc Solvers with Global Constraints

Some FlatZinc solvers provide specific algorithms for certain global constraints (such as `alldifferent`, `cumulative`, etc.). Thus, the user can choose not to decompose some global constraints during the translation from MiniZinc to FlatZinc, in order to profit from specific algorithms provided by the solvers.

Table 5.5 shows the performance of two FlatZinc solvers with support for global constraints compared to the performance of themselves, and that of `fzn2smt`, without using that support, on problems where global constraints do occur. Note that `fzn2smt` does not provide any specific support for global constraints. We have not included the results for G12 lazy_fd with global constraints, since it exhibited very similar performance.

Again we can see that `fzn2smt` overall offers a bit better performance than Gecode and G12, even when they are using global constraints. This is even more significant if we take into

Table 5.5: Accumulated time for the solved instances and the number of solved instances (within parenthesis) of `fzn2smt` and some available FlatZinc solvers with global constraints. +gc stands for using global constraints, and −gc stands for not using global constraints. Type 's' stands for satisfaction and 'o' for optimization. # stands for the number of instances.

			Gecode				G12				`fzn2smt`	
Problem	Type	#	+gc		−gc		+gc		−gc			
debruijn-binary	s	11	**4.14**	**(7)**	4.46	(6)	35.93	(7)	31.30	(6)	0.50	(1)
pentominoes	s	7	**65.82**	**(7)**	601.82	(7)	847.11	(4)	848.17	(4)	168.47	(2)
quasigroup7	s	10	**250.49**	**(6)**	278.73	(6)	1.55	(5)	1.72	(5)	20.03	(5)
rcpsp	o	10	10.56	(5)	12.07	(5)	0.51	(4)	97. 27	(5)	**993.74**	**(9)**
black-hole	s	10	20.88	(7)	69.68	(7)	2405.38	(6)	2423.48	(6)	**892.46**	**(8)**
nonogram	s	10	**493.61**	**(8)**	1353.63	(8)	351.35	(2)	336.93	(2)	1546.56	(7)
open-stacks	o	10	**168.56**	**(8)**	169.74	(8)	283.52	(8)	299.55	(8)	707.25	(7)
p1f	o	10	**730.60**	**(10)**	2.57	(8)	1.87	(8)	2.04	(8)	126.01	(9)
rect-packing	s	10	132.44	(6)	134.36	(6)	7.71	(5)	339.80	(6)	**106.66**	**(10)**
roster-model	o	10	**0.88**	**(10)**	1.04	(10)	4. 49	(10)	4.46	(10)	50.38	(10)
costasArray	s	5	**615.51**	**(4)**	943.75	(4)	411.82	(3)	423.09	(3)	664.13	(2)
depot-placement	o	15	1035.72	(12)	1205.75	(12)	519.64	(8)	522.81	(8)	**295.78**	**(15)**
filter	o	10	31.15	(1)	30.95	(1)	280.57	(1)	278. 72	(1)	**17.88**	**(6)**
ghoulomb	o	10	**1044.25**	**(10)**	0.00	(0)	598.54	(3)	246.67	(1)	2545.02	(6)
rcpsp-max	o	10	5.04	(2)	42.11	(2)	116.40	(2)	30. 00	(1)	**475.14**	**(4)**
sugiyama2	o	5	310.94	(5)	308.31	(5)	510.84	(5)	510. 72	(5)	**8.64**	**(5)**
bacp	o	15	848.88	(10)	847.78	(10)	979.85	(10)	976.35	(10)	**64.17**	**(15)**
Total		168	5769	(118)	6006	(105)	7357	(91)	7373	(89)	**8682**	**(121)**

account that most of these problems are optimization ones, and we have naively implemented a search procedure to supply the lack of support for optimization of SMT solvers (see Subsection 5.3.3). However, Gecode is best in 9 problems, whereas `fzn2smt` is best only in 8. We believe that unit propagation and conflict-driven lemma learning at Boolean level, partially compensate for the lack of specialized algorithms for global constraints in SMT solvers.

5.4.4 Impact of the Boolean component of the instances in the performance of `fzn2smt`

In this section we statistically compare the performance of `fzn2smt` with the best of the other available FlatZinc solvers, which is Gecode. We compare the number of solved instances by Gecode and `fzn2smt`, taking into account their Boolean component. In particular, we consider the number of Boolean variables and the number of non-unary clauses of the SMT instances resulting from the translation of each FlatZinc instance. We first look at this relation graphically (figures 5.3 and 5.4) and propose the following hypothesis: the more Boolean component the problem has, the better the performance of `fzn2smt` is with respect to that of Gecode. This hypothesis seems quite reasonable, because having a greater Boolean component, the SMT solver can better profit from built-in techniques such as unit propagation, learning and backjumping. We provide statistical tests to support this hypothesis.

First of all, we define the *normalized difference of solved instances* of each problem

$$dif = \frac{\#\texttt{fzn2smt}\ solved\ instances - \#Gecode\ solved\ instances}{\#instances}.$$

This difference ranges from −1 to 1, where −1 means that Gecode has solved all the instances and `fzn2smt` none, and 1 means the inverse.

We define the *Boolean variables ratio* r_v of each problem as the average of the number of Boolean variables divided by the number of variables of each SMT instance.

Similarly, we define the *disjunctions ratio* r_d of each problem as the average of the number of non-unary clauses divided by the number of constraints of each SMT instance.

In figure 5.3 we plot the differences with respect to the Boolean variables ratio, and in figure 5.4 with respect to the disjunctions ratio. These figures show that, the more Boolean variables and disjunctions the problem has, the better performance `fzn2smt` has, compared to Gecode. In particular, when the Boolean variables ratio r_v is above 0.2, `fzn2smt` is able to solve more instances than Gecode (i.e., the difference is positive). Only in two of those problems Gecode is able to solve more instances than `fzn2smt`, namely in **nmseq** (problem #2) and **open-stacks** (problem #14). In these problems, Boolean variables are mainly used in `bool2int()` constraints, hence these variables provide little Boolean structure and the SMT solver cannot profit from their contribution. When considering the disjunctions ratio, `fzn2smt` outperforms Gecode only when r_d is above 0.4. An exception to this fact is again on **nmseq**, where most disjunctions come from `bool2int()`.

Note that `fzn2smt` is able to solve more instances than Gecode in **propagation stress** (problem #16), which has neither Boolean variables nor disjunctions. This is probably due to the fact that the linear constraints of the problem can be solved by the *Integer Difference Logic* (IDL), a subset of LIA which is very efficiently implemented by Yices [DdM06a].

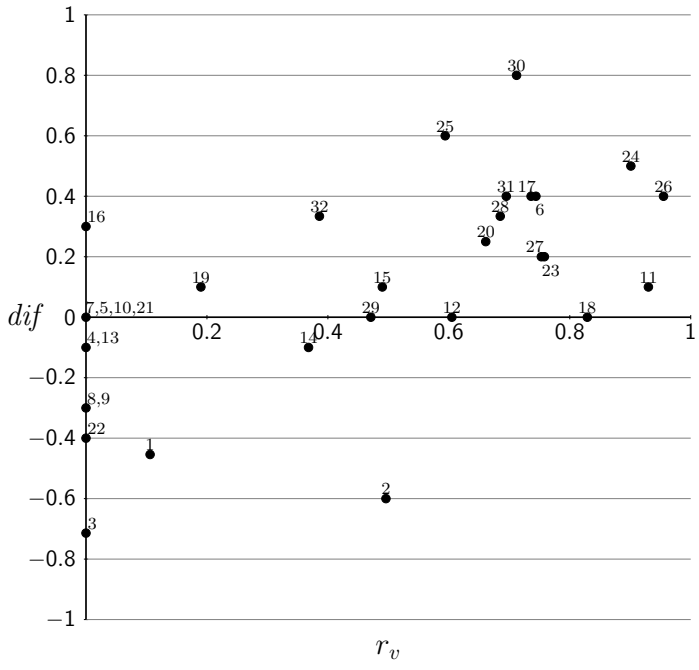

Figure 5.3: Normalized difference of solved instances between `fzn2smt` and Gecode with respect to the ratio of Boolean variables. The numbers next to the points denote instance numbers (see Table 5.4).

We use a paired t-test in order to show that our method (`fzn2smt` with Yices) solves significantly more instances than Gecode. For each problem $i \in 1..n$, being n the number of considered problems, we take X_i as the normalized number of instances solved by `fzn2smt`

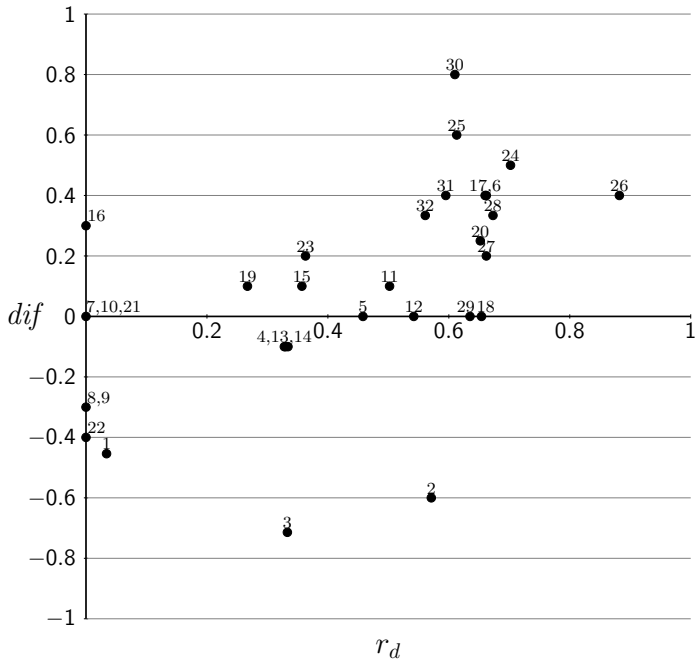

Figure 5.4: Normalized difference of solved instances between `fzn2smt` and Gecode with respect to the ratio of disjunctions. The numbers next to the points denote instance numbers (see Table 5.4).

and Y_i as the normalized number of instances solved by Gecode, and define $D_i = X_i - Y_i$ with null hypothesis

$$H_0 : \mu_D = \mu_x - \mu_y = 0$$

i.e., $H_0 : \mu_x = \mu_y$. Then we calculate the t-value as

$$t = \frac{\overline{D}_n}{S_D^*/\sqrt{n}}$$

where $\overline{D}_n$ is the sample mean of the D_i and S_D^* is the sample standard deviation of the D_i. This statistic follows a Student's-t distribution with $n - 1$ degrees of freedom.

Table 5.6 shows that in the general case with all 32 problems there is no significant difference between the means of the two solvers (the probability of the null hypothesis is $p = 0.2354$). Therefore we cannot say that `fzn2smt` is statistically better than Gecode. But, already for problems with Boolean variables ratio $r_v \geq 0.1$ we observe a significant difference (i.e., with $p < 0.05$ in all tests) in favor of `fzn2smt`. This confirms our hypothesis: the higher the Boolean variables ratio is, the better the performance of `fzn2smt` is with respect to that of Gecode. We also note that if we use the disjunctions ratio r_d for comparing the means, the results are similar: with $r_d >= 0.4$ the difference of means is significant in favor of `fzn2smt`.

Finally, Table 5.7 shows that the difference of means of `fzn2smt` and G12 lazy_fd is less significant. We have not taken into account the problems not supported by G12 lazy_fd due to unbounded integer variables. These results suggest that the two approaches work similarly well for the same kind of problems.

Table 5.6: Paired t-test, with probability p of the null hypothesis, for the difference in mean of the number of solved instances by `fzn2smt` and Gecode, for problems with different ratios r_v and r_d.

r_v	#problems	p
≥ 0.0	32	**0.2354**
≥ 0.1	21	**0.0150**
≥ 0.2	19	**0.0043**
≥ 0.3	19	**0.0043**
≥ 0.4	17	**0.0057**
≥ 0.5	14	**0.0001**
≥ 0.6	13	**0.0003**

r_d	#problems	p
≥ 0.0	32	**0.2354**
≥ 0.1	24	**0.0472**
≥ 0.2	24	**0.0472**
≥ 0.3	23	**0.0540**
≥ 0.4	17	**0.0060**
≥ 0.5	16	**0.0056**
≥ 0.6	11	**0.0006**

Table 5.7: Paired t-test, with probability p of the null hypothesis, for the difference in mean of the number of solved instances by `fzn2smt` and G12 lazy_fd, for problems with different ratios r_v and r_d.

r_v	#problems	p
≥ 0.0	27	**0.8929**
≥ 0.1	16	**0.0157**
≥ 0.2	15	**0.0152**
≥ 0.3	15	**0.0152**
≥ 0.4	14	**0.0147**
≥ 0.5	13	**0.0140**
≥ 0.6	12	**0.0028**

r_d	#problems	p
≥ 0.0	27	**0.8929**
≥ 0.1	20	**0.1853**
≥ 0.2	20	**0.1853**
≥ 0.3	19	**0.1856**
≥ 0.4	15	**0.0279**
≥ 0.5	14	**0.0274**
≥ 0.6	10	**0.0633**

5.5 Conclusions

In this chapter we have presented `fzn2smt`, a tool for translating FlatZinc instances to the standard SMT-LIB v1.2 language, and solving them through a pluggable SMT solver used as a black box. Our tool is able to solve not only decision problems, but optimization ones. In fact, surprisingly good results have been obtained on many optimization problems although our tool uses simple optimization algorithms: successive calls to the decision procedure, performing either linear, binary or hybrid search. In Chapter 8 we present a more refined optimization algorithm based on incrementality.

The `fzn2smt` system with Yices 2 has obtained remarkably good results in the MiniZinc challenges 2010, 2011 and 2012. In contrast, it has obtained poor results in the MiniZinc challenge 2013. Related to this let us mention that, although not eligible for prizes, the true winner of all the MiniZinc challenges (in almost all categories) is Chuffed, a lazy clause generation solver developed by the MiniZinc group. Recall from Section 3.7.1 that lazy clause generation is a combination of a CP solver with a SAT solver and hence, it is apparent that more cross-fertilization between the CP and the SAT and SMT communities is necessary in order to tackle real-world problems properly.

We have performed exhaustive experimentation on nearly 300 MiniZinc instances using four distinct FlatZinc solvers, and using `fzn2smt` with five distinct SMT solvers (using various translation options), lasting more than 12 days of CPU. The good results obtained

by `fzn2smt` on the MiniZinc benchmarks suggest that the SMT technology can be effectively used for solving CSPs in a broad sense. We think that `fzn2smt` can help getting a picture of the suitability of SMT solvers for solving CSPs, as well as to compare the performance of state-of-the-art SMT solvers outside the SMT competition.

Table 5.4 evidences that in scheduling problems (rcpsp, depot-placement, rcpsp-max, wwtp and bacp) the performance of `fzn2smt` with Yices 2 is much better than that of other FlatZinc solvers. In fact, in Chapter 5 we propose a system to solve the RCPSP using SMT with very good results. These kind of problems have a rich Boolean component (Boolean variables and disjunctions). We have proposed the hypothesis that the more Boolean component the problem has, the better the performance of `fzn2smt` is with respect to Gecode.[5] This hypothesis seems quite reasonable, because the greater the Boolean component is, the better the SMT solver is supposed to profit from built-in techniques such as unit propagation, learning and backjumping. We have also provided statistical tests that support this hypothesis.

Finally, we think that better results could be obtained by directly translating from a high-level language to SMT. In doing so, more clever translations could be possible avoiding some of the flattening and probably less variables could be generated. For instance, MiniZinc disjunctions of arithmetic constraints are translated into FlatZinc constraints by reifying them with auxiliary Boolean variables. In our approach this step is not needed since it is already done by the SMT solver. Moreover, the FlatZinc formula resulting from the translation of many MiniZinc benchmarks has very simple Boolean structure (the formula is often trivially satisfiable at the Boolean level), and hence it is likely that the SMT solver cannot substantially profit from conflict-driven lemma learning on unsatisfiable instances. In fact, there exist unsatisfiable instances that result in a few or no conflicts at all, and most of the work is hence done by the theory solver. In Chapter 7 we propose a system to solve CSPs by directly translating from a high-level declarative language into SMT.

[5] Gecode is the second best system considering the total number of solved instances.

Chapter 6

An SMT approach to the Resource-Constrained Project Scheduling Problem

In this chapter we consider one of the problems where SMT solvers has shown good results in the previous chapter: the Resource Constrained Project Scheduling Problem (RCPSP).[1] However, instead of using a high-level declarative language for dealing with this problem, we provide an ad hoc tool to ease the incorporation of preprocessing algorithms and to have more precise control of the SMT encodings used.

The RCPSP consists in scheduling a set of non-preemptive activities with predefined durations and demands on each of a set of renewable resources, subject to partial precedence constraints. Normally the goal is to minimize the makespan. This is one of the most general scheduling problems that has been extensively studied in the literature. Some surveys published in the last years include [HRD98, BDM$^+$99, KP01, HL05] and, more recently, [HB10, KALM11]. The RCPSP is NP-hard in the strong sense [BMR88, BLK83]. However, many small instances (with up to 50 activities) are usually tractable within a reasonable time.

Many approaches have been developed to solve the RCPSP: constraint programming (CP) [LM08, Bap09], Boolean satisfiability (SAT) [Hor10], mixed integer linear programming (MILP) [KALM11], branch and bound algorithms (BB) [DPPH00] and others [DV07]. An approach using lazy clause generation has shown very good results [SFSW09, SFSW10].

Next we show that state-of-the-art SMT solvers are a viable engine for solving the RCPSP. We propose some variations of well-known MILP formulations of the RCPSP, which are easily encoded using SMT with the theory of linear integer arithmetic. We have built a tool, named `rcp2smt`, which solves RCPSP instances using the (Weighted Max)SMT solver Yices 1 [DdM06a] through API. Since the RCPSP is an optimization problem and SMT does not directly support it, our tool can deal with optimization using two natural approaches: on the one hand we can simply iteratively encode the RCPSP instance into SMT instances that bound the optimal makespan, and on the other hand we can encode the RCPSP into a Weighted MaxSMT instance where the soft clauses encode the objective function. The Weighted MaxSMT instance can be solved using the Yices default Weighted MaxSMT algorithm or using the WPM1 algorithm based on unsatisfiable cores [ABL09, MSP09].

[1] This problem is denoted as $PS|temp|C_{max}$ in [BDM$^+$99] and $m,1|gpr|C_{max}$ in [HL05].

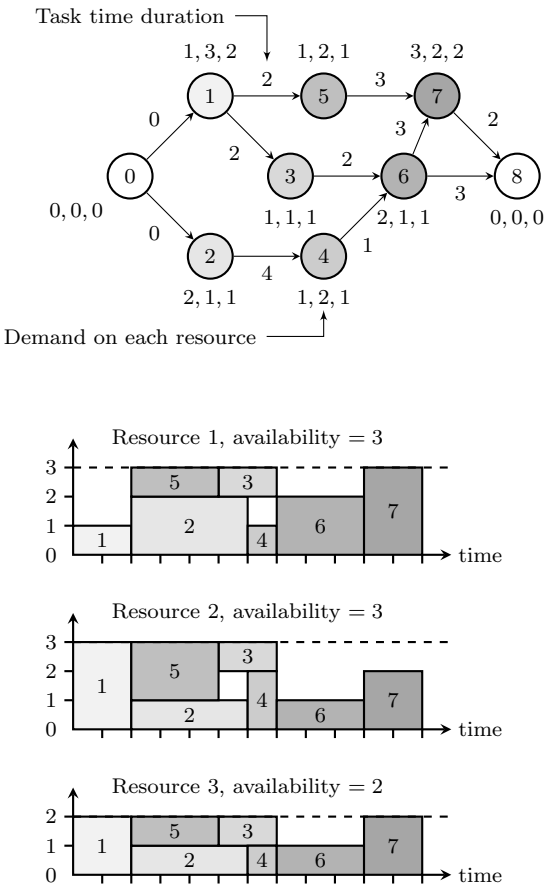

Figure 6.1: An example of RCPSP [LM08]

This chapter is based on results from [ABP+11a] and it is organized as follows. First we briefly introduce the RCPSP. Then we describe `rcp2smt` with special emphasis in the preprocessing and optimization phases. Next we describe our encodings for the RCPSP using the SMT formalism, and provide performance comparisons between ours and others approaches. Finally, we present some conclusions.

6.1 The Resource-Constrained Project Scheduling Problem (RCPSP)

The RCPSP is defined by a tuple (V, p, E, R, B, b) where:

- $V = \{A_0, A_1, \ldots, A_n, A_{n+1}\}$ is a set of activities. Activity A_0 represents by convention the start of the schedule and activity A_{n+1} represents the end of the schedule. The set of non-dummy activities is defined by $A = \{A_1, \ldots, A_n\}$.
- $p \in \mathbb{N}^{n+2}$ is a vector of durations. p_i denotes the duration of activity i, with $p_0 = p_{n+1} = 0$ and $p_i > 0, \forall A_i \in A$.
- E is a set of pairs representing precedence relations, thus $(A_i, A_j) \in E$ means that the execution of activity A_i must precede that of activity A_j, i.e., activity A_j must start after

activity A_i has finished. We assume that we are given a precedence activity-on-node graph $G(V, E)$ that contains no cycles; otherwise the precedence relation is inconsistent. Since the precedence is a transitive binary relation, the existence of a path in G from the node i to node j means that activity i must precede activity j. We assume that E is such that A_0 is a predecessor of all other activities and A_{n+1} is a successor of all other activities.

- $R = \{R_1, \ldots, R_m\}$ is a set of m renewable resources.
- $B \in \mathbb{N}^m$ is a vector of resource availabilities. B_k denotes the availability amount of each resource R_k
- $b \in \mathbb{N}^{(n+2)\times m}$ is a matrix of the demands of the activities for resources. $b_{i,k}$ represents the amount of resource R_k used during the execution of A_i. Note that $b_{0,k} = 0$ and $b_{n+1,k} = 0$, $\forall k \in \{1, \ldots, m\}$.

A schedule is a vector $S = (S_0, S_1, \ldots, S_n, S_{n+1})$ where S_i denotes the start time of each activity $A_i \in V$. We assume that $S_0 = 0$. A solution of the RCPSP problem is a non-preemptive (an activity cannot be interrupted once it is started) schedule S of minimal makespan S_{n+1} subject to the precedence and resource constraints:

$$\text{minimize } S_{n+1} \tag{6.1}$$

subject to:

$$S_j - S_i \geq p_i \qquad \forall (A_i, A_j) \in E \tag{6.2}$$

$$\sum_{A_i \in \mathcal{A}_t} b_{i,k} \leq B_k \qquad \forall R_k \in R, \forall t \in H \tag{6.3}$$

A schedule S is feasible if it satisfies the generalized precedence constraints (6.2) and the resource constraints (6.3) where $\mathcal{A}_t = \{A_i \in A \mid S_i \leq t < S_i + p_i\}$ represents the set of non-dummy activities in process at time t, the set $H = \{0, \ldots, T\}$ is the scheduling horizon, and T (the length of the scheduling horizon) is an upper bound for the makespan.

In the example of Figure 6.1, three resources and seven activities are considered. Each node is labeled with the number of the activity it represents. The durations of the activities are indicated in the on-going arcs, and the resources consumptions are indicated in the labels next to the nodes. The upper part of the picture represents therefore the instance to be solved, while the bottom part gives a feasible solution using Gantt charts. For each resource, the horizontal axis represents the time and the vertical axis represents the consumption.

6.2 RCPSP to SMT

The input to `rcp2smt` is an RCPSP instance in `rcp` or `sch` format,[2] which is preprocessed in order to build more suitable instances for our solving method (e.g., obtaining better bounds, extending the set of precedences, etc.). After the preprocessing phase, `rcp2smt` searches for an optimal solution as described in the Optimization section.

[2]RCPSP formats from PSPLib [KS97].

6.2.1 Preprocessing

The preprocessing phase computes the extended precedence set, a lower and an upper bound for the makespan, time windows for each activity and a matrix of incompatibilities between activities.

Extended precedence set Since a precedence is a transitive relation we can compute the minimum precedence between each pair of activities in E. For this calculation we use the Floyd-Warshall (FW) algorithm $O(n^3)$ on the graph defined by the precedence relation E labeling each arc (A_i, A_j) with the duration p_i. This extended precedence set is named E^* and contains, for all pair of activities A_i and A_j such that A_i precedes A_j, a tuple of the form $(A_i, A_j, l_{i,j})$ where $l_{i,j}$ is the length of the longest path from A_i to A_j. Notice also that, if $(A_i, A_i, l_{i,i}) \in E^*$ for some A_i and $l_{i,i} > 0$, then there is a cycle in the precedence relation and therefore the problem is inconsistent and has no solution.

Lower Bound The lower bound LB is the minimum time we can ensure that the last activity A_{n+1} begins. There are different methods for computing the lower bounds, see [KS99, MMRB98]. We have only implemented two of them:

- $LB1$*: Critical path bound.* This is the most obvious lower bound. To compute this, we ignore the capacity restrictions. Then, the minimal project duration is the length of a critical path in the project network. The critical path is the longest path of the graph of precedences between the initial activity A_0 and the final activityA_{n+1}. For instance, in Figure 6.1 the critical path is $[A_0, A_2, A_4, A_6, A_7, A_8]$ and has length 10. Notice that we can easily know the length of this path if we have already computed the precedence set since we only need to obtain $l_{0,n+1}$ from $(A_0, A_{n+1}, l_{0,n+1}) \in E^*$.

- $LB2$*: Capacity bound.* To compute this bound, we ignore the precedence restrictions. This bound value is the maximum of the total requirement for each resource divided by the capacity of the resource and rounded up to the next integer. The additional cost to compute this lower bound is $O(nm)$ (being n the number of activities and m the number of resources). For instance, in the example of Figure 6.1 the capacity bound of resource 3 is 11.

$$LB2 = \max\{\lceil (\sum_{A_i \in A} b_{i,k} * p_i)/B_k \rceil \mid B_k \in B\}$$

Finally, we set our computed lower bound to $LB = \max\{LB1, LB2\}$.

Upper Bound The upper bound UB is the maximum time we can ensure that the last activity A_{n+1} begins. We have considered the trivial upper bound for the RCPSP problem:

$$UB = \sum_{A_i \in A} p_i$$

For instance, in the example of Figure 6.1 the upper bound is 22.

Time windows We can reduce the domain of each variable S_i (start of activity A_i) that initially is $\{0\,..\,UB - p_i\}$ by computing its time window. The time window of activity A_i is $[ES_i, LS_i]$, being ES_i the earliest start time and LS_i the latest start time. To compute the time window we use the lower and upper bound and the extended precedence set as follows:

For activities $A_i, 0 \leq i \leq n$,

$$\begin{array}{rcll} ES_i & = & l_{0,i} & (A_0, A_i, l_{0,i}) \in E^* \\ LS_i & = & UB - l_{i,n+1} & (A_i, A_{n+1}, l_{i,n+1}) \in E^* \end{array}$$

and for activity A_{n+1},

$$ES_{n+1} = LB \qquad LS_{n+1} = UB$$

For instance, in the example of Figure 6.1, activity A_4 has time window $[4, 16]$.

Notice that, if E^* has been successfully computed, then $l_{0,i} \geq 0$ for all $(A_0, A_i, l_{0,i}) \in E^*$ and $l_{i,n+1} \geq p_i$ for all $(A_i, A_{n+1}, l_{i,n+1}) \in E^*$.

Incompatibility We compute the matrix of Booleans I where each element $I[i, j]$ (noted as $I_{i,j}$) is true, if the activity A_i and the activity A_j cannot overlap in time. The incompatibility can occur for two reasons:

- *Precedence.* There exists a precedence constraint between A_i and A_j, i.e., $(A_i, A_j, l_{i,j}) \in E^*$ or $(A_j, A_i, l_{j,i}) \in E^*$.
- *Resources.* For some resource R_k, the sum of the demands of the two activities A_i and A_j, is greater than the resource capacity B_k: $\exists R_k \in R$ s.t. $b_{i,k} + b_{j,k} > B_k$.

For instance, in the example of Figure 6.1, activity A_4 is incompatible with activities A_0, A_2, A_6, A_7 and A_8 due to the precedences, and with activities A_1, A_5 and A_7 due to resources demands.

This matrix of incompatibilities is symmetric, and the additional cost for its computation is $O(n^2 m)$.

Notice that we could compute incompatibilities due to resource demands between subsets of activities in general (i.e., not restricted to two activities). We have explored this approximation for subsets of size bigger than 2 but the gain of efficiency has been almost zero or even negative, in all encodings and solving methods.

6.2.2 Optimization

There are two approaches we follow to solve RCPSP through SMT solvers. Both approaches share the constraints derived from the preprocessing steps, and the ones from (6.2) and (6.3) modelled as the Encodings section describes. The difference consists on how we treat the objective function. On the one hand, we implement an ad hoc search procedure which calls the SMT solver successively constraining the domain of the variable S_{n+1}, by adding a constraint $S_{n+1} \leq bound$ (where *bound* is a constant and $LB \leq bound \leq UB$) and perform a dichotomic search strategy. Notice that when we get a satisfiable answer, we can eventually refine our upper bound by checking the value of S_{n+1} in the satisfying assignment. Furthermore, since we are using Yices through API, when we get a satisfiable answer we can keep the learned clauses. We will refer to this method as dico.

On the other hand, some SMT solvers, like Yices, can solve Weighted MaxSMT instances. Weighted MaxSMT allows us to represent optimization problems. We just need to transform

the objective function into a set of soft constraints, and keep the rest of the constraints as hard. In order to translate the objective function, we can simply enumerate all its possible values. For example, taking into account the RCPSP in figure 6.1, where $LB = 11$ and $UB = 22$, we add the following set of soft constraints: $\{(11 \leq S_{n+1}, 1), (12 \leq S_{n+1}, 1), \ldots, (22 \leq S_{n+1}, 1)\}$. Each soft constraint is represented by the pair (C, w) where C is a constraint and w the weight (or cost) of falsifying C. We can obtain a more compact encoding by considering the binary representation of S_{n+1}. Following our example, we would add a new hard constraint, $1 \cdot b_1 + 2 \cdot b_2 + 4 \cdot b_3 + 8 \cdot b_4 + LB = S_{n+1}$, where b_i are Boolean variables, and the following set of soft constraints: $\{(\neg b_1, 1), (\neg b_2, 2), (\neg b_3, 4), (\neg b_4, 8)\}$. The resulting instance can be solved by any SMT solver supporting Weighted MaxSMT, like Yices. In our experiments, we refer to the method which uses the non-binary encoding of the objective function as yices.

We have extended the Yices framework by incorporating Weighted Max-SAT algorithms. In particular, we have implemented the algorithms WPM1 and WBO from [ABL09, MSP09] (see Section 4.1.2 for a description of WPM1), which is based on the detection of unsatisfiable cores. The performance of these algorithms heavily depends on the quality of the cores. Therefore, we have extended them with a heuristic that prioritizes those cores which involve constraints with higher weights. In our experiments, we refer to the method which uses the binary encoding of the objective function and this algorithm as core.

6.3 Encodings

SAT modulo linear integer arithmetic allows us to directly express all the constraints of the following encodings, since we can logically combine arithmetic predicates. It is worth noting that in the resulting formulas both Boolean variables and integer variables can occur together. The three encodings we propose are inspired by existing ones, and conveniently adapted to SMT. Moreover, some refinements are introduced considering time windows, incompatibilities, extended precedences. We also add redundant constraints for better propagation.

Since a schedule is a vector $S = (S_0, S_1, \ldots, S_n, S_{n+1})$ where S_i denotes the start time of each activity $A_i \in V$, in all encodings we use a set $\{S_0, S_1, \ldots, S_n, S_{n+1}\}$ of integer variables. By S' we denote the set $\{S_1, \ldots, S_n\}$.

Also, in all encodings the objective function is (6.1), and we add the following constraints:

$$S_0 = 0 \tag{6.4}$$

$$S_i \geq ES_i \qquad \forall A_i \in \{A_1, \ldots, A_{n+1}\} \tag{6.5}$$

$$S_i \leq LS_i \qquad \forall A_i \in \{A_1, \ldots, A_{n+1}\} \tag{6.6}$$

$$S_j - S_i \geq l_{i,j} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{6.7}$$

where (6.5) and (6.6) are simple encodings of the time windows for each activity, and (6.7) encodes the extended precedences.

We also introduce additional constraints for the incompatibilities between activities due to resource capacity constraints:

$$\begin{aligned} S_i + p_i \leq S_j \vee S_j + p_j \leq S_i \\ \forall I_{i,j} \in I \text{ s.t. } I_{i,j} = \mathit{true}, \\ (A_i, A_j, l_{i,j}) \notin E^* \text{ and } (A_j, A_i, l_{j,i}) \notin E^* \end{aligned} \tag{6.8}$$

Notice that the incompatibilities between activities due to precedence constraints are already encoded in (6.7).

6.3.1 Time formulation

The more natural encoding for the RCPSP in SMT is the Time formulation, very similar to the MILP encoding proposed by [PW96] and referred in [KALM11] as Basic discrete-time formulation and in [SFSW09, SFSW10] as Time-resource decomposition. The idea is that, for every time t and resource R_k, the sum of all resource requirements for the activities must be less than or equal to the resource availabilities.

$$ite((S_i \leq t) \wedge \neg(S_i \leq t - p_i); x_{i,t} = 1; x_{i,t} = 0) \quad \forall S_i \in S', \forall t \in \{ES_i, \ldots, LS_i + p_i - 1\} \tag{6.9}$$

$$\sum_{A_i \in A} ite(t \in \{ES_i, \ldots, LS_i + p_i - 1\}; b_{i,r} * x_{i,t}; 0) \leq B_r \quad \forall B_r \in B, \forall t \in H \tag{6.10}$$

where $ite(c, e_1, e_2)$ is an *if-then-else* expression denoting e_1 if c is true and e_2 otherwise. Notice that $t \in \{ES_i, \ldots, LS_i + p_i - 1\}$ can be easily encoded into $ES_i \leq t \wedge LS_i \leq LS_i + p_i - 1$.

We remark that (6.9) imposes that $x_{i,t} = 1$ if the activity A_i is active at time t and $x_{i,t} = 0$ otherwise. We restrict the possible times by using the time windows. Constraints (6.10) encode the resource constraints (6.3) using the $x_{i,t}$ variables.

Constraints (6.9) can be replaced by the equivalent ones:

$$y_{i,t} \leftrightarrow (S_i \leq t) \wedge \neg(S_i \leq t - p_i) \quad \forall S_i \in S', \forall t \in \{ES_i, \ldots, LS_i + p_i - 1\} \tag{6.11}$$

$$ite(y_{i,t}; x_{i,t} = 1; x_{i,t} = 0) \quad \forall S_i \in S', \forall t \in \{ES_i, \ldots, LS_i + p_i - 1\} \tag{6.12}$$

We have observed that for small problem instances this formulation gives better performance results than the previous. However, for big instances (with more than 50 activities) the addition of the new variables $y_{i,t}$ usually makes the problem intractable by state-of-the-art SMT solvers.

In order to improve propagation we have considered the following constraints:

$$(S_i = t) \rightarrow y_{i,t'} \quad \forall S_i \in S', \forall t \in \{ES_i, \ldots, LS_i + p_i - 1\}, \forall t' \in \{t, \ldots, t + p_i - 1\} \tag{6.13}$$

$$(S_i = t) \rightarrow \neg y_{i,t'} \quad \forall S_i \in S', \forall t \in \{ES_i, \ldots, LS_i + p_i - 1\}, \forall t' \in \{ES_i, \ldots, LS_i + p_i - 1\} \setminus \{t, \ldots, t + p_i - 1\} \tag{6.14}$$

In our experiments, these constraints have shown to provide execution speedup only for small instances.

The following redundant constraints help improving the search time in all instances. This is probably due to the special treatment of those kind of constraints in SMT solvers:

$$x_{i,t} \geq 0 \quad \forall S_i \in S', \forall t \in \{ES_i, \ldots, LS_i + p_i - 1\} \tag{6.15}$$

$$x_{i,t} \leq 1 \quad \forall S_i \in S', \forall t \in \{ES_i, \ldots, LS_i + p_i - 1\} \tag{6.16}$$

6.3.2 Task formulation

In this formulation we use variables indexed by activities instead of by time. The key idea is that checking only that there is no overload at the beginning (end) of each activity is sufficient to ensure that there is no overload at every time (for the non-preemptive case). In this formulation, the number of variables and constraints is independent of the length of the scheduling horizon T. This formulation is similar to the Time-resource decomposition of [SFSW09] and it is inspired by the encoding proposed in [OEK99] for temporal and resource reasoning in planning.

$$z^1_{i,j} = \mathit{true} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{6.17}$$

$$z^1_{j,i} = \mathit{false} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{6.18}$$

$$\begin{gathered} z^1_{i,j} = S_i \leq S_j \\ \forall A_i, A_j \in A, \\ (A_i, A_j, l_{i,j}) \notin E^*, (A_j, A_i, l_{j,i}) \notin E^*, i \neq j \end{gathered} \tag{6.19}$$

$$z^2_{i,j} = \mathit{false} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{6.20}$$

$$z^2_{j,i} = \mathit{true} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{6.21}$$

$$\begin{gathered} z^2_{i,j} = S_j < S_i + p_i \\ \forall A_i, A_j \in A, \\ (A_i, A_j, l_{i,j}) \notin E^*, (A_j, A_i, l_{j,i}) \notin E^*, i \neq j \end{gathered} \tag{6.22}$$

$$z_{i,j} = 0 \qquad \forall I_{i,j} \in I \text{ s.t. } I_{i,j} = \mathit{true} \tag{6.23}$$

$$\begin{gathered} \mathit{ite}(z^1_{i,j} \wedge z^2_{i,j}; z_{i,j} = 1; z_{i,j} = 0) \\ \forall I_{i,j} \in I \text{ s.t. } I_{i,j} = \mathit{false}, i \neq j \end{gathered} \tag{6.24}$$

$$\sum_{A_i \in A \setminus \{A_j\}} b_{i,k} * z_{i,j} \leq B_k - b_{j,k} \qquad \forall A_j \in A, \forall B_k \in B \tag{6.25}$$

The Boolean variables $z^1_{i,j}$ are true if activity A_i starts not after A_j and false otherwise. The Boolean variables $z^2_{i,j}$ are true if activity A_j starts before A_i ends and false otherwise. The integer variables $z_{i,j}$ are 1 if activities A_i and A_j overlap and 0 otherwise. The last constraints (6.25) state that, for every activity A_j and resource R_k, the sum of the resource demands $b_{i,k}$ for R_k from the activities A_i that overlap with A_j should not exceed the capacity B_k of R_k less the demand $b_{j,k}$ for R_k from A_j.

Moreover, in order to improve propagation, we have the following redundant constraints encoding anti-symmetry and transitivity of the precedence relation:

$$z^1_{i,j} \vee z^1_{j,i} \qquad \forall A_i, A_j \in A, i \neq j \tag{6.26}$$

$$(z^1_{i,j} \wedge z^1_{j,k}) \rightarrow z^1_{i,k} \qquad \forall A_i, A_j, A_k \in A, i \neq j, j \neq k, i \neq k \tag{6.27}$$

The following constraints have also shown to improve propagation in this encoding:

$$z^1_{i,j} \vee z^2_{i,j} \qquad \forall A_i, A_j \in A, i \neq j \tag{6.28}$$

$$z^1_{i,j} \rightarrow z^2_{j,i} \qquad \forall A_i, A_j \in A, i \neq j \tag{6.29}$$

6.3.3 Flow formulation

This formulation is inspired by the formulations of [KALM11, AMR03] named Flow-based continuous-time formulation.

$$y_{i,j} = \mathit{true} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{6.30}$$

$$y_{j,i} = \mathit{false} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{6.31}$$

$$\begin{gathered} y_{i,j} = S_j \geq S_i + p_i \\ \forall A_i, A_j \in V, \\ (A_i, A_j, l_{i,j}) \notin E^*, (A_j, A_i, l_{j,i}) \notin E^*, i \neq j \end{gathered} \tag{6.32}$$

$$f_{i,j,k} \geq 0 \qquad \forall A_i, A_j \in V, R_k \in R, i \neq j \tag{6.33}$$

In the following constraints $b^e_{q,k}$ denotes $b_{q,k}$ for all $A_q \in A$ and denotes B_k for $q = 0$ and $q = n + 1$.

$$f_{i,j,k} \leq \min(b^e_{i,k}, b^e_{j,k}) \qquad \forall A_i, A_j \in V, R_k \in R, i \neq j \tag{6.34}$$

$$y_{i,j} \vee f_{i,j,k} = 0 \qquad \forall A_i, A_j \in V, R_k \in R, i \neq j \tag{6.35}$$

$$\begin{gathered} \sum_{A_j \in A \cup \{A_{n+1}\}} f_{i,j,k} = b_{i,k} \\ \forall A_i \in A \cup \{A_0\}, \forall R_k \in R, i \neq j \end{gathered} \tag{6.36}$$

$$\begin{gathered} \sum_{A_i \in A \cup \{A_0\}} f_{i,j,k} = b_{j,k} \\ \forall A_j \in A \cup \{A_{n+1}\}, \forall R_k \in R, i \neq j \end{gathered} \tag{6.37}$$

The Boolean variables $y_{i,j}$ are true if activity A_j starts after A_i finishes and false otherwise. The integer variables $f_{i,j,k}$ denote the quantity of resource R_k that is transferred from A_i (when finished) to A_j (at the start of its processing). Constraints (6.33) and (6.34) fix their range of values. Constraints (6.35) denote that if A_j does not start after the completion of A_i then the flow from A_i to A_j should be 0 for all resource R_k. These constraints link the $y_{i,j}$ and $f_{i,j,k}$ variables. Constraints (6.36) and (6.37) state that, for each activity and resource, the sum of the flows transferred and the sum of the flows received by the activity must be equal to its demand for the resource. Notice no flow enters A_0 and no flow exits from A_{n+1}.

Similarly to the Task formulation, we have the following constraints for anti-symmetry and transitivity of the precedences:

$$\neg y_{i,j} \vee \neg y_{j,i} \qquad \forall A_i, A_j \in A, i \neq j \tag{6.38}$$

$$(y_{i,j} \wedge y_{j,k}) \rightarrow y_{i,k} \qquad \forall A_i, A_j, A_k \in A, i \neq j, j \neq k, i \neq k \tag{6.39}$$

The constraints for the incompatibilities (6.8) are reformulated as follows:

$$\begin{gathered} y_{i,j} \vee y_{j,i} \\ \forall I_{i,j} \in I \text{ s.t. } I_{i,j} = \mathit{true}, \\ (A_i, A_j, l_{i,j}) \notin E^* \text{ and } (A_j, A_i, l_{j,i}) \notin E^* \end{gathered} \tag{6.40}$$

		Time			Task			Flow		
set	#	core	dico	yices	core	dico	yices	core	dico	yices
KSD30	480	1.81(100)	1.24(99)	**1.20(100)**	1.19(95)	1.14(96)	0.37(95)	4.42(91)	9.78(92)	11.27(89)
Pack	55	3.46(63)	**2.43(65)**	6.75(65)	37.15(36)	27.71(40)	33.30(41)	143.02(3)	88.35(3)	86.26(5)
KSD15_d	479	7.96(100)	7.18(100)	6.01(100)	0.28(100)	**0.18(100)**	0.30(100)	0.10(99)	0.11(99)	0.26(99)
Pack_d	55	67.45(14)	41.35(12)	177.91(12)	**1.69(43)**	1.25(41)	33.30(40)	35.74(12)	30.05(12)	7.44(12)

Table 6.1: % solved instances in parenthesis, mean time in seconds, cutoff 500 seconds.

set	#	MCS	[KALM11]	lazy_fd	SMT
KSD30	480	7.39(97)	DDT–10.45(82)	**0.49(100)**	yices(Time)–1.20(100)
Pack	55	115.88(25)	**DDT–63.39(76)**	94.40(20)	dico(Time)–2.43(65)
KSD15_d	479	0.07(100)	FCT–12.06(94)	**0.04(100)**	dico(Task)–0.18(100)
Pack_d	55	72.34(38)	OOE–75.58(18)	**51.24(62)**	core(Task)–1.69(43)

Table 6.2: % solved instances in parenthesis, mean time in seconds, cutoff 500 seconds.

6.4 Experimental results

We have run our experiments on machines with the following specs; operating system: Rocks Cluster 5.2 Linux 2.6.18, processor: AMD Opteron 242 Processor 1.5 GHz, memory: 450 MB, cache: 1024 KB, and compiler: GCC 4.1.2.

The benchmarks we considered are those from the experimental investigation conducted in [KALM11]. We focus on 4 families of RCPSP instances: KSD30 (KSD instances from the PSPLib with 30 activities), Pack instances from [CN03], Pack_d and KS15_d [KALM11]. We are mainly interested on the two last benchmarks. As pointed out in [KALM11], KSD and Pack instances, which can typically be found in the process industry, involve relatively short time horizons. Therefore, the authors of [KALM11] generated the Pack_d and KS15_d sets, which involve longer time horizons.

At tables, the numbers in parenthesis indicate the percentage of optimal solutions found, while the others indicate the average execution time of solved instances. The timeout was set to 500 seconds as in [KALM11]. Notice that the processor used in [KALM11] was also of 1.5Ghz.

Table 6.1 presents the results of the comparison among the three formulation techniques we presented in the Encodings section: Time, Task, and Flow. For each formulation, we applied three different solving approaches based on the Yices 1.0.29 SMT solver, which are described in the Optimization section: dico, yices and core. As we can see, for both sets KSD30 and Pack, the Time encoding is the best one while, for those sets involving longer time horizons, KSD15_d and Pack_d, the Task encoding is superior to the rest. We conjecture that Time behaves better when time horizons are smaller, like in the KSD30 and Pack sets, because the number of variables grows significantly with the time horizon. Both Task and Flow formulations are independent of time horizons, however the Flow formulation involves more variables and constraints. Notice that the Task formulation has $O(n^2)$ variables, being n the number of activities, while the Flow formulation has $O(n^2 * m)$ variables, where m is the number of resources. The same complexities apply to the number of constraints.

Table 6.2 compares the best solving method reported by [KALM11] and our best solving

method on each set of benchmarks. In [KALM11] the authors compare several MILP formulations solved with the ILOG Cplex solver against the method of [Lab05] which is based on the detection and resolution of Minimal Critical Sets (MCS) at each node of the search. Among the MILP formulations, we highlight the best performing ones: Disaggregated Discrete-Time (DDT), Flow-based Continuous-Time (FCT) and On/Off Event-based (OOE). Moreover, we compare with the `lazy_fd` solver from [SFSW09].

Overall, the SMT based approach is a very competitive tool. The approach by [KALM11] is the best performing one for the Pack set, closely followed by the Time formulation with the dico method, but it is worse on the other sets, especially on the sets with larger horizons, where there is a difference of at least one order of magnitude. MCS is comparable to the SMT approaches for the easiest sets, but it is also about one order of magnitude worse for the Pack and Pack_d sets. The `lazy_fd` solver exhibits very good performance in all sets of instances apart from the Pack set.[3] Compared to `lazy_fd`, although the SMT approach is able to solve less instances from the Pack_d set, it is able to solve more instances from the Pack set, resulting in a larger number of solved instances globally. Moreover, it has very low mean times in all sets.

6.5 Conclusions

In this chapter we have presented `rcp2smt`, an ad hoc tool to deal with the RCPSP using SMT technology which is competitive with the state-of-the-art RCPSP solving methods. `rcp2smt` can read instances in two input formats (rcp and sch). These instances can be solved using one of the three proposed SMT encodings: Time, Task and Flow. The Time encoding is in general the best encoding, except when the time horizons are longer being better the Task encoding. This is reasonable since in the Time encoding the number of constraints and variables mainly depends on the size of the time horizon, while in the Task and Flow formulations the number of constraints and variables mainly depends on the number of activities.

Another important feature of `rcp2smt` is the preprocessing phase, which significantly helps to solve the problem. For instance, since the Time encoding is very sensitive to the time horizon, computing better bounds and reducing the time window of the activities greatly reduces the number of constraints and variables.

We have implemented some optimization algorithms using SMT. In particular we want to remark that, as far as we known, the WPM1 algorithm presented in [ABP+11a] was the first implementation of a core-based algorithm for SMT. However, experiments don't show which type of algorithm is the best.

Finally, the experimental results have confirmed that this kind of problems, where we have a combination of Boolean and integer constraints, are a good target for SMT.

[3] To improve the performance of `lazy_fd` for this set an specialized propagator for the cumulative resource constraint was developed [SFS13].

Chapter 7

Simply and WSimply

In this chapter we describe WSimply, a system that reformulates from its own high-level (W)CSP language into SMT. The development of a system with its own language has several direct benefits. First of all, we have a complete control of the reformulation from the high-level language into SMT, for instance, allowing us to decide when we want to use auxiliary variables. In addition, we can easily extend the language with new features, for instance by introducing weighted constraints and meta-constraints. Finally, it simplifies the integration optimization with SMT.

This chapter is based on results from [BPSV09, ABP+13, BEPV12, BPSV14] and it is organised as follows. First of all, in Section 7.1, we present an overview of the evolution of the tool, from Simply to WSimply and its extensions. Next, in Section 7.2 we explain the current architecture of the tool. In Section 7.3 we describe the language of the input files of the system. In Section 7.4 and Section 7.5 we describe the reformulations from (W)CSP into SMT and PB/LP respectively. In Section 7.6 we describe the solving module for dealing with COP and WCSPs. In Section 7.7 we present experimental results and comparisons of the different versions of the tool. Finally, conclusions are given in Section 7.8.

7.1 History

The origin of Simply lies in the development of a tool that uses SMT technology for solving CSPs and, at the same time, simplifies the task of generating SMT benchmarks. At the current state of development, the (W)CSPs are solved using the Yices 1 API with the LIA logic, but it also has an option to generate SMT-Lib 1.2 files, so other SMT solvers can be used externally. Its input language is a new high-level declarative language for CSPs, which has the same name as the system. Next we describe the evolution of the system.

In first version of Simply [BPSV09], which was written in Haskell, CSP instances were translated into SMT instances, which were then solved by calling an SMT solver. The language of this version only supported basic logical and arithmetic constraints, a few global constraints, two statements, the *forall* and the *if-then-else*, the first one for generating multiple constraints and the second one to filter them, and finally an special *if-then-else* construct which was evaluated at solving time (based on the special *ite* function of the standard SMT language). In order to solve the SMT instance, this version checked the satisfiability of the instance, and in the case of optimization problems it performed binary search algorithm calling

the SMT solver with the same instance but changing the bounds of the objective function.

Since the results of the prototype were encouraging, we decided to develop the next version of `Simply` using the C++ language and integrating it with the Yices 1 API, which was the solver with best performance solving the CSPs tested with the previous version. In addition this integration would allow us to take advantage of one of the strongest points of SMT technology, which is learning. This is especially important in optimization problems, because it allows `Simply` to keep the learned clauses during the successive calls to the solver. Having this in mind, we decided to extend the tool to deal with Weighted CSPs by allowing to associate a weight to the desired constraints, i.e., by defining soft constraints. Then the WCSP instances can be solved by reformulating the WCSP into a COP, defining an objective function to be minimized which is the aggregation of the soft constraint costs, but also by using modified Weighted MaxSAT algorithms for Weighted MaxSMT, such as WPM1 (presented in Section 4.1.2), which could be very interesting for solving WCSPs. Furthermore, the language was extended with more global constraints, with the *exists* statement (similar to the *forall* but generating a disjunction instead of a conjunction of constraints), allowing the user to define its own constraints, and, finally, with meta-constraints, giving to the user more expressibility and control over the soft constraints. Note that the name of the tool changed from `Simply` to `WSimply` [ABP+13] when it was extended to deal with WCSPs.

Another extension of `WSimply` was to be able to generate files in the pseudo-Boolean formats [BEPV12]: pseudo-Boolean optimization (pbo) and weighted PseudoBoolean (wbo), which then can be solved using specialized PseudoBoolean solvers. Although this extension was only implemented for a subset of the language (almost all as we will see in Section 7.5), also two new meta constraints were introduced.

Finally, a new incremental optimization method was introduced in `WSimply` [BPSV14], which uses Binary Decision Diagrams (BDD) to represent the objective function of the WCSPs, and then this BDD is represented by clauses using a state-of-the-art SAT encoding for BDDs. At the same time, PB reformulation was extended to generate optimization LP files in order to compare `WSimply` with state-of-the-art LP solvers such as CPLEX.

7.2 Architecture of the tool

In this section we describe the architecture of the current version of `WSimply`. Note we do not explain the architecture of previous versions for the sake of simplicity.

We have three possible reformulations in `WSimply` for solving WCSP instances: using SMT, PB or LP. The input of the system is the same for the three reformulations, a WCSP instance with meta-constraints formed by a model file, which defines the problem, and an optional data file, which contains specific data of the instance.

In Figure 7.1 we show the reformulation and solving steps when using SMT. First, `WSimply` reformulates the input instance into the suitable format for the solving procedures. It has four reformulations: (R1) from a WCSP instance with meta-constraints into a WCSP instance (without meta-constraints), (R2) from a WCSP instance into a COP instance, (R3) from a WCSP instance into a WSMT instance and (R4) from a COP instance into a WSMT instance.

Once the problem has been properly reformulated, `WSimply` can apply two different solving procedures: WSMT Solving (S1) or Optimisation SMT Solving (S2).

The architecture of `WSimply` for the reformulations into PB and into LP is very similar to the one shown in Figure 7.1. In the case of the PB reformulation, steps R1, R2, R3, S1 and

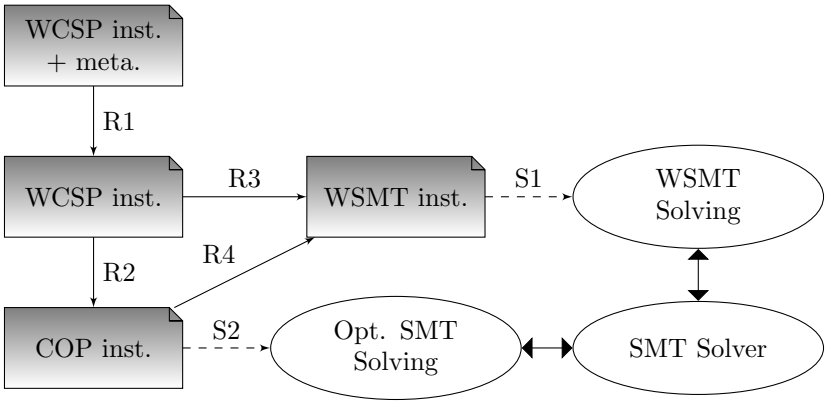

Figure 7.1: Basic architecture and SMT solving process of `WSimply`.

S2 are the same, but, instead of generating SMT instances, generating a PB instance in one of the two possible formats supported by PB solvers: weighted PB (for S1) and optimization PB (for S2). Step R4 does not exist in the PB reformulation. In the case of the LP reformulation, only steps R1, R2 and S2 exist, since LP solvers only support optimization instances.

In the following sections we describe the different reformulations and solving procedures.

7.3 Language description

The input of `WSimply` is a WCSP instance, written in the `WSimply` language (similar to EaCL [MTW+99] and MiniZinc [Nic07]), which is formed by a model file and, optionally, a data file. In this section we first describe the language for the model file and then for the data file.

7.3.1 Model File

In Figure 7.2 we present the complete grammar of `WSimply` model files.

The model file always starts with the keyword `Problem`, followed by the character "`:`" and a string naming the problem. After that several sections are used to model the problem:

- Data definition
- Domains definition
- Variables definition
- Predefinition of Functions and Constraints (optional)
- Constraints posting
- Meta-constraints posting (optional)
- Optimization (optional)

As an example in Figure 7.3 we show the classical n-Queens problem.

```
⟨simply_program⟩   ::=  Program:⟨id⟩ ⟨data⟩ ⟨domains⟩ ⟨variables⟩
                        ⟨user_defined_functions⟩? ⟨constraints⟩
                        ⟨meta_constraints⟩? ⟨optimitzation⟩?
⟨data⟩             ::=  Data ⟨data_def⟩*
⟨data_def⟩         ::=  int ⟨id⟩ = ⟨arithm_exp⟩ ;
                    |   bool ⟨id⟩ = ⟨formula⟩ ;
                    |   (bool | int) ⟨array_id⟩ = ⟨list⟩ ;
⟨domains⟩          ::=  Domains ⟨domain_def⟩*
⟨domain_def⟩       ::=  Dom ⟨id⟩ = ⟨list⟩ ;
⟨variables⟩        ::=  Variables ⟨variable_def⟩*
⟨variable_def⟩     ::=  IntVar ⟨array_id⟩ ( , ⟨array_id⟩)* :: (⟨id⟩ | int) ;
                    |   BoolVar ⟨array_id⟩ ( , ⟨array_id⟩)* ;
⟨array_id⟩         ::=  ⟨id⟩
                    |   ⟨id⟩[ ⟨arithm_exp⟩ ( , ⟨arithm_exp⟩)* ]
⟨user_defined_functions⟩      ::=  Functions ⟨user_defined_functions_def⟩*
⟨user_defined_functions_def⟩  ::=  ⟨def_constraint⟩
                               |   ⟨def_function⟩ ;
⟨def_constraint⟩              ::=  DefConstraint ⟨id⟩ ⟨dfc_params⟩ { ⟨sentence⟩+ }
⟨def_function⟩                ::=  Function (int | bool) ⟨id⟩ ⟨dfc_params⟩ ⟨sentence⟩
⟨dfc_params⟩                  ::=  ( (int | bool) ⟨id⟩ (, (int | bool) ⟨id⟩)* )
⟨constraints⟩      ::=  Constraints ⟨sentence⟩+
⟨sentence⟩         ::=  ⟨constraint⟩ ;
                    |   ⟨soft_constraint⟩ ;
                    |   ⟨list⟩ ;
⟨soft_constraint⟩  ::=  ⟨label_def⟩? ⟨constraint⟩ ⟨weight⟩
⟨label_def⟩        ::=  # ⟨array_id⟩ :
⟨weight⟩           ::=  @ { ⟨weight_value⟩ }
                    |   @ ⟨weight_value⟩
⟨weight_value⟩     ::=  ⟨arithm_exp⟩
                    |   -
⟨constraint⟩       ::=  ⟨formula⟩
                    |   ⟨global_constraint⟩
⟨formula⟩          ::=  Not ⟨formula⟩
                    |   ⟨formula⟩ ⟨bool_operator⟩ ⟨formula⟩
                    |   ⟨arithm_exp⟩ ⟨relational_operator⟩ ⟨arithm_exp⟩
                    |   ⟨user_defined_references⟩
                    |   If_Then_Else (⟨formula⟩) { ⟨sentence⟩+} { ⟨sentence⟩+}
                    |   ⟨statement⟩
                    |   ( ⟨formula⟩ )
                    |   ⟨array_id⟩
                    |   True
                    |   False
⟨user_defined_references⟩  ::=  ⟨id⟩ ⟨dfc_param⟩
⟨relational_operator⟩      ::=  = | <> | < | > | =< | >=
⟨bool_operator⟩            ::=  And | Or | Xor | Iff | Implies
```

```
⟨global_constraint⟩  ::=  AllDifferent( ⟨list⟩ )
                      |   Sum( ⟨list⟩ , ⟨arithm_exp⟩ )
                      |   Count( ⟨list⟩ , (⟨arithm_exp⟩|⟨formula⟩) , ⟨arithm_exp⟩ )
                      |   AtLeast( ⟨list⟩ , ⟨arithm_exp⟩ )
                      |   AtMost( ⟨list⟩ , ⟨arithm_exp⟩ )
                      |   ExactlyOne( ⟨list⟩ )
⟨arithm_exp⟩         ::=  ⟨numeral⟩
                      |   ⟨array_id⟩
                      |   ⟨arithm_exp⟩ ⟨arithm_operator⟩ ⟨arithm_exp⟩
                      |   ( ⟨arithm_exp⟩ )
                      |   Abs( ⟨arithm_exp⟩ )
                      |   ⟨user_defined_references⟩
                      |   length( ⟨arithm_exp⟩ )
                      |   max( ⟨list⟩ )
                      |   min( ⟨list⟩ )
                      |   sum1( ⟨list⟩ )
⟨statement⟩          ::=  ⟨if_then_else⟩
                      |   ⟨forall⟩
                      |   ⟨exists⟩
⟨if_then_else⟩       ::=  If ( ⟨formula⟩ ) Then { ⟨sentence⟩+ }
                      |   If ( ⟨formula⟩ ) Then { ⟨sentence⟩+ } Else { ⟨sentence⟩+ }
⟨forall⟩             ::=  Forall( (⟨id⟩ in ⟨list⟩ )+ ) { ⟨sentence⟩+ }
⟨exists⟩             ::=  Exists( (⟨id⟩ in ⟨list⟩ )+ ) { ⟨sentence⟩+ }
⟨arithm_operator⟩    ::=  + | - | * | Div | Mod
⟨list⟩               ::=  [ ⟨list_element⟩ ( , ⟨list_element⟩)* ]
                      |   [ (⟨arithm_exp⟩|⟨formula⟩) | ⟨var_restrict⟩( , ⟨var_restrict⟩)*]
⟨list_element⟩       ::=  ⟨arithm_exp⟩
                      |   ⟨range⟩
⟨var_restrict⟩       ::=  ⟨id⟩ in ⟨list⟩
                      |   ⟨formula⟩
⟨range⟩              ::=  ⟨arithm_exp⟩ .. ⟨arithm_exp⟩
⟨id⟩                 ::=  non empty sequence of letters, digits and _, not starting with digit
⟨numeral⟩            ::=  non empty sequence of digits
⟨comments⟩           ::=  % anything else until end of line
                      |   \ * anything else * \
⟨meta_constraints⟩       ::=  ⟨meta_constraint_def⟩ ;
⟨meta_constraint_def⟩    ::=  priority( ⟨list⟩ )
                          |   priority( ⟨array_id⟩ , ⟨array_id⟩ , ⟨arithm_exp⟩ )
                          |   samePriority( ⟨list⟩ )
                          |   multiLevel( ⟨list_of_lists⟩ )
                          |   atLeast( ⟨list⟩ , ⟨arithm_exp⟩ )
                          |   maxCost( ⟨list_of_lists⟩ , ⟨arithm_exp⟩ )
                          |   minCost( ⟨list_of_lists⟩ , ⟨arithm_exp⟩ )
                          |   homogeneousAbsoluteWeight( ⟨list_of_lists⟩ , ⟨arithm_exp⟩ )
                          |   homogeneousPercentWeight( ⟨list_of_lists⟩ , ⟨arithm_exp⟩ )
                          |   homogeneousAbsoluteNumber( ⟨list_of_lists⟩ , ⟨arithm_exp⟩ )
                          |   homogeneousPercenteNumber( ⟨list_of_lists⟩ , ⟨arithm_exp⟩ )
⟨optimitzation⟩      ::=  Minimize ⟨id⟩ ;
                      |   Maximize ⟨id⟩ ;
```

Figure 7.2: Grammar of the `WSimply` model file.

```
Problem:queens

    Data
        int n=8;

    Domains
        Dom rows=[1..n];

    Variables
        IntVar q[n]::rows;

    Constraints

        AllDifferent([q[i]|i in [1..n]]);

        Forall(i in [1..n-1]) {
            Forall(j in [i+1..n]) {
                q[i]-q[j]<>j-i;
                q[j]-q[i]<>j-i;
            };
        };
```

Figure 7.3: `WSimply` model of the classical n-Queens problem.

Data definition

The data definition section is where the problem constants are declared, and optionally a value is assigned to them. Otherwise it is supposed that the value will be taken from the data file.

The data definition section starts with the token `Data` and we can define two types of constants: integer and Boolean. In both cases a data definition could be a single value or an indexed list of values. These values can be directly defined or can be arithmetic expressions or formulas which will be evaluated in compilation time. Note that the order in the data definition is relevant, since we only can use data values already defined in the definition of new data.

Domains definition

The domains definition section is where the domains for the integer variable are defined.

The section starts with the token `Domains` and each domain is defined by a list of integer values, which can be created by combinations of integers and ranges separated by commas.

Variables definition

The variables definition section is where variables, single or arrays, are defined.

The section starts with the token `Variables` and we can define two types of variables: integer (`IntVar`) or Boolean (`BoolVar`).

Integer variables must have a domain assigned to them. However, `WSimply` provides a predefined domain, called `int`, that lefts the variable unbounded. It is responsibility of the user to bound it later in the constraints section.

User Functions and Constraints definition

We can also predefine functions and constraints in order to have simpler models.

The functions and constraints definition section starts with the token `Functions`. We can define functions, which will be used to evaluate a value from its parameters in compilation time, and constrains, which will be used to generate constraints from its parameters in compilation time.

For instance, in the queens example we can define the constraint to forbid the diagonal attack as follows:

```
DefConstraint non_diagonal_attack(int t,int i,int j)
{
    t[i]-t[j]<>j-i;
    t[j]-t[i]<>j-i;
}
```

and then in the constraints posting section we can refer to this constraint using its name and parameters.

Constraints posting

The constraints posting section is where the problem is modelled. Constraints posting starts with the token `Constraints` followed by one or more sentences. A sentence is a constraint (hard or soft) or a comprehension list that posts a conjunction of constraints (generated by the list). Later in this section we explain how a constraint is set to be soft. A constraint is a formula or a global constraint.

A formula can be:

- The `True` and `False` constants.
- A Boolean decision variable.
- A reference to a user defined constraint.
- A negation of another formula.
- A formula in parentheses.
- A binary Boolean operation on two formulas: `And`, `Or`, `Xor`, `Iff` and `Implies`.
- A binary relational operation on two arithmetic expressions: `<`, `>`, `=<`, `>=` and `=`.
- An statement evaluated at compilation time: *forall*, *exists* or *if-then-else*.
- An `If_Then_Else`, which is a special case of constraint because it is like the if-then-else statement but evaluated at solving time.

A global constraint can be:

- `Sum(List,Value)`. This constraint enforces equality between `Value` and the sum of all elements of `List`. When the `List` is empty, the `Value` is enforced to be zero.
- `Count(List,Value,N)`. This constraint states equality between `N` and the number of occurrences of `Value` in `List`. When the `List` is empty, `N` is enforced to be zero.
- `AtMost(List,Value,N)`. This constraint states that the number of occurrences of `Value` in `List` must be less or equal than `N`. When `List` is empty, `N` is enforced to be zero.
- `AtLeast(List,Value,N)`. This constraint states that the number of occurrences of `Value` in `List` must be greater or equal than `N`. When `List` is empty, `N` is enforced to be zero.
- `ExactlyOne(List)`. This constraint states that the number of satisfied constraints in `List` must be exactly one.
- `AllDifferent(List)` requires all the elements of `List` to be different.

Note that the elements of `List` for the `Sum` global constraint must be arithmetic expressions, while in the case of the `Count`, `AtMost` and `AtLeast` global constraints they only have to be of the same type as `Value` (arithmetic expressions or formulas).

In the n-Queens model in Figure 7.3, we use the global constraint `AllDifferent`, two statements (two nested `Foralls`) and two basic constraints. In Figure 7.4 we changed the basic constraints by a reference to the `non_diagonal_attack` constraint.

```
...
Constraints

    AllDifferent([q[i]|i in [1..n]]);

    Forall(i in [1..n-1]) {
        Forall(j in [i+1..n]) {
            non_diagonal_attack(q,i,j);
        };
    };
```

Figure 7.4: `WSimply` model of the classical n-Queens problem using a user defined reference.

There are three types of statements in `WSimply`, that are evaluated during compilation time:

- *Forall*: generates a conjunction of sentences
- *Exists*: generates a disjunction of sentences
- *If-then-Else*: needs a condition, and generates the sentences of the `Then` if the condition is evaluated to true (at compilation time) otherwise generates the sentences of the `Else`.

It is important to notice the difference between the *if-then-else* statement and the constraint `If_Then_Else`, which, as we have said, is not evaluated at compilation time. Consider, for instance, the following example:

```
If (i < 4) Then {
    m[i] <> m[i+1];
} Else {
    m[i] <> m[i-1];
    m[i] = m[i-2];
}
```

Condition `i < 4` is evaluated at compilation time. Therefore, the variable `i` cannot be a decision variable, i.e., it must be a constant or a "local" variable, e.g., an index of a `Forall` statement.

There are two types of lists in `WSimply`:

- Extensional lists: by directly enumerating elements and ranges
- Intensional lists: using comprehension lists.

Finally, an arithmetic expression can be:

- A number.
- An integer decision variable.
- A reference to a user defined integer function.
- An arithmetic expression in parentheses.
- A binary arithmetic operation on two arithmetic expressions: `+`, `-`, `*`, `Div` and `Mod`.
- The absolute value of an arithmetic expression (`Abs`).
- The minimum/maximum of a list of arithmetic expressions (`min`/`max`).
- The length of a list (`length`).
- The sum of a list of arithmetic expressions (`sum1`).

We want to remark that the arithmetic expressions `length`, `max` and `min` are evaluated at compilation time only, therefore the elements in $\langle list \rangle$ must be evaluated at compilation time.

Note that the arithmetic expression `sum1` is exactly equal to the operator `+`, but it is useful when we the number of elements to sum is undefined.

Optimization

When the user wants to solve a COP only has to specify an objective variable to be minimized or maximized, e.g., `Minimize` *optimization_variable*`;`.

Soft constraints posting

In the constraints posting we can combine posting of hard constraints with soft constraints.

A soft constraint is a constraint with an associated weight defined using an `@` and the expression for the weight. The weight expression is the cost of falsifying the constraint. Optionally, it is allowed to define a label for the soft constraint in order to reference it in other constraints or in meta-constraints. The labelling is done by adding an `#`, the label and `:` before the soft constraint.

$$\#label : (constraint) \,@\, \{weight\};$$

`WSimply` only supports linear integer expressions int the weight expression. It can either be evaluable at compile time, or contain decision variables. Weight expressions should always evaluate to a non-negative integer, as they amount to a cost. When negative they will be considered as zero. More details on how weights are treated can be found in Section 7.4.3.

An interesting detail worth remarking on is that by allowing the use of decision variables in the cost expression we can encode the degree of violation of a constraint. For instance, considering a soft constraint on the maximum number of shifts that a worker has to work per week, the violation cost could be increased by one unit for each extra shift worked:

$$(\texttt{worked\_shifts} < 6) \,@\, \{\texttt{base\_cost} + \texttt{worked\_shifts} - 5\};$$

Meta-constraints

Inside the model file, we can also define meta-constraints. The meta-constraints section starts with the token `MetaConstraints`, followed by one or more meta-constraint posting. Next we describe the meta-constraints available in `WSimply` grouped in three families: priority, homogeneity and dependence.

1. **Priority.** The user may have some preferences on the soft constraints to violate. For instance, if we have an activity to perform and worker 1 *doesn't want* to perform it while worker 2 *should not* perform it, then it is better to violate the first constraint than the second. It would be useful to free the user of deciding the exact value of the weight of each constraint.

   To this end, we allow the use of undefined weights denoted as "_":

   $$\#label : (constraint) \,@\, \{\_\};$$

   The value of this undefined weight is computed at compile time according to the meta-constraints that refer to the label. This simplifies the modelling of the problem, since the user does not need to compute any concrete weight.

   `WSimply` provides the following meta-constraints related to priority:

   - `samePriority(`*List*`)`, where *List* is a list of labels of weighted constraints. With this meta-constraint we are stating that the constraints referred to in the *List* are soft constraints with the same priority.

- `priority(`*List*`)`, where *List* is a list of labels of weighted constraints. With this meta-constraint we are stating that the constraint corresponding to the i-th label in *List* has higher priority than the constraint corresponding to the $(i+1)$-th label. In other words, it must be more costly to violate the i-th constraint than the $(i+1)$-th.
- `priority(`$label_1$`,`$label_2$`,`n`)`, with $n > 1$, defines how many times it is worse to violate the constraint corresponding to $label_1$ than to violate the constraint corresponding to $label_2$. That is, if $weight_1$ and $weight_2$ denote the weights associated with $label_1$ and $label_2$, respectively, we are stating that $weight_1 \geq weight_2 * n$.
- `multiLevel(`*ListOfLists*`)`, where *ListOfLists* is a list of lists of labels. This meta-constraint states that the cost of falsifying each of the constraints referred to by the labels in the i-th list is greater than the cost of falsifying the constraints referred to by the labels in the rest of the lists (from the $i+1$-th list to the last) all together. For example,

  ```
  multiLevel([[A,B,C],[D,E,F],[G,H,I]]);
  ```

  states that the cost of falsifying each of the constraints (denoted by) `A`, `B` and `C` is greater than the cost of falsifying `D`, `E`, `F`, `G`, `H`, and `I` together and, at the same time, the cost of falsifying each of the constraints `D`, `E` and `F` is greater than the cost of falsifying `G`, `H`, and `I` together.

2. **Homogeneity.** The user may wish there to be some homogeneity in the amount of violation of disjoint groups of constraints. For instance, for the sake of fairness, the number of violated preferences of nurses should be as homogeneous as possible. `WSimply` provides the following meta-constraints related to homogeneity:

   - `maxCost(`*ListOfLists*`,`v`)` where *ListOfLists* is a list of lists of labels of soft constraints and v is a positive integer. This meta-constraint ensures that the maximum cost of the violated constraints of each list in *ListOfLists* is v.
   - `minCost(`*ListOfLists*`,`v`)` where *ListOfLists* is a list of lists of labels of soft constraints and v is a positive integer. This meta-constraint ensures that the minimum cost of the violated constraints of each list in *ListOfLists* is v.
   - `atLeast(`*List*`,`p`)`, where *List* is a list of labels of soft constraints and p is a positive integer in 1..100. This meta-constraint ensures that the percentage of constraints denoted by the labels in *List* that are satisfied is at least p.
   - `homogeneousAbsoluteWeight(`*ListOfLists*`,`v`)`, where *ListOfLists* is a list of lists of labels of soft constraints and v is a positive integer. This meta-constraint ensures that, for each pair of lists in *ListOfLists*, the difference between the cost of the violated constraints in the two lists is at most v. For example, given

     ```
     homogeneousAbsoluteWeight([[A,B,C],[D,E,F,G]],10);
     ```

     if the weights of constraints `A`, `B` and `C` are 5, 10 and 15 respectively, and constraints `A` and `B` are violated and constraint `C` is satisfied, then the cost of the violated constraints in `[D,E,F,G]` must be between 5 and 25.

- `homogeneousAbsoluteNumber(`*ListOfLists*`,`v`)`. Same as above, but where the maximum difference v is between the number of violated constraints.
- `homogeneousPercentWeight(`*ListOfLists*`,`p`)`, where *ListOfLists* is a list of lists of labels of soft constraints and p is a positive integer in 1..100. This meta-constraint is analogous to `homogeneousAbsoluteWeight`, but where the maximum difference p is between the percentage in the cost of the violated constraints (with respect to the cost of all the constraints) in each list.
- `homogeneousPercentNumber(`*ListOfLists*`,`p`)`. Same as above, but where the maximum difference p is between the percentage in the number of violated constraints.

We remark that the `homogeneousAbsoluteWeight` and `homogeneousPercentWeight` meta-constraints are not allowed to refer to any constraint with undefined weight. This is because, constraints with undefined weight are referenced by priority meta-constraints, and their weight is determined at compile time accordingly to those priority meta-constraints, independently from other constraints. Hence, since the meta-constraints `homogeneousAbsoluteWeight` and `homogeneousPercentWeight` also constrain the weight of the referenced constraints, if they were allowed to reference constraints with undefined weights, this could lead to incompleteness. We also remark that these two meta-constraints can not refer to constraints whose weights contain decision variables. Nevertheless, note that we can homogenise constraints using the meta-constraints `homogeneousAbsoluteNumber` and `homogeneousPercentNumber`, without worrying about the weights.

3. **Dependence.** Particular configurations of violations may entail the necessity to satisfy other constraints, that is, if a soft constraint is violated then another soft constraint must not, or a new constraint must be satisfied. For instance, working the first or the last turn of the day is penalized but, if somebody works in the last turn of one day, then he cannot work in the first turn the next day. This could be succinctly stated as follows:

```
#A: not_last_turn_day_1 @ w1;
#B: not_first_turn_day_2 @ w2;
(Not A) Implies B;
```

stating that, if the soft constraint `A` is violated, then the soft constraint `B` becomes hard.

7.3.2 Data file

Inside the data file, we can only assign values to data. The data definition is inside the model file, so in the data file we cannot determine the type or the number of dimensions of the data elements.

In Figure 7.5 we present the grammar of `WSimply` data files.

7.4 WCSP reformulation into SMT

In this section we describe the reformulation steps from a WCSP instance into a SMT instance. First of all we describe the meta-constraints reformulation into hard and soft constraints

$$
\begin{array}{lcl}
\langle data\_file \rangle & ::= & \langle data\_file\_def \rangle^* \\
\langle data\_file\_def \rangle & ::= & \langle id \rangle \texttt{ = } \langle arithm\_exp \rangle \texttt{ ;} \\
& | & \langle id \rangle \texttt{ = } \langle formula \rangle \texttt{ ;} \\
& | & \langle id \rangle \texttt{ = } \langle list \rangle \texttt{ ;}
\end{array}
$$

Figure 7.5: `WSimply` data file grammar.

(reformulation step R1 of Figure 7.1). Then we describe the hard constraints reformulation into SMT. Then we describe the soft-constraints reformulation into a COP (reformulation step R2 of Figure 7.1) and into a WSMT instance (reformulation step R3 of Figure 7.1). And finally we describe the COP reformulation into a WSMT instance (reformulation step R4 of Figure 7.1).

7.4.1 Meta-constraint reformulation

We remove all meta-constraints by reformulating them into hard and soft `WSimply` constraints.

As we have shown in Section 7.3.1, meta-constraints use labels. Therefore, first of all, we introduce a new reification variable for each labelled soft constraint of the form:

$$\#label : (constraint) \,@\, \{expression\};$$

and we replace the constraint by:

$$b_{label} \Leftrightarrow constraint;$$
$$b_{label} \,@\, \{expression\};$$

where b_{label} is the fresh (Boolean) reification variable introduced for this constraint.

In the following we show how `WSimply` reformulates the priority, homogeneity and dependence meta-constraints.

Reformulation of Priority Meta-Constraints

To deal with the *priority* meta-constraints, we create a system of linear inequations on the (probably undefined) weights of the referenced soft constraints. The inequations are of the form $w = w'$, $w > w'$ or $w \geq n \cdot w'$, where w is a variable, w' is either a variable or a non-negative integer constant, and n is a positive integer constant. For example, given

```
#A: (a>b)@{3};  #B:(a>c)@{_};  #C: (a>d)@{_};  #D:(c=2-x)@{_};
priority([A,B,C]);
priority(D,B,2);
```

the following set of inequations is generated:

$$w_A = 3, w_B > 0, w_C > 0, w_D > 0,$$
$$w_A > w_B, w_B > w_C,$$
$$w_D \geq 2 \cdot w_B$$

This set of inequations is fed into an SMT solver[1] which acts as an oracle at compile time, so that a model, i.e., a value for the undefined weights satisfying the inequations, can be

[1] In fact, the set of inequations could be fed into any linear integer arithmetic solver.

found. Following the previous example, the SMT solver would return a model such as, e.g.:

$$w_A = 3, w_B = 2, w_C = 1, w_D = 4$$

This allows the reformulation of the original problem into an equivalent WCSP without undefined weights:

```
#A:(a>b)@{3};  #B:(a>c)@{2};  #C:(a>d)@{1};  #D:(c=2-x)@{4};
```

Hence, with the meta-language, and thanks to this simple use of a solver as an oracle at compile time, we free the user of the tedious task of thinking about concrete weights for encoding priorities.

Since satisfiability test based algorithms (see Section 7.6) usually behave better with low weights, we ask not only for a solution of the inequations system, but for a solution minimising the sum of undefined weights.

In the case of the `multiLevel` meta-constraint, given for example

```
#A:(a>b)@{_};  #B:(a>c)@{_};  #C:(a>d)@{_};  #D:(c=2-x)@{_};
multiLevel([[A,B][C,D]]);
```

the following set of inequations would be generated:

$$\begin{aligned} &w_A > 0, w_B > 0, w_C > 0, w_D > 0, \\ &w_A > (w_C + w_D), \\ &w_B > (w_C + w_D), \end{aligned}$$

and the SMT solver would return a model such as, e.g.:

$$w_A = 3, w_B = 3, w_C = 1, w_D = 1$$

We remark that the weight expressions of the constraints referenced by a priority meta-constraint must either be undefined or evaluable at compile time, i.e., they cannot use any decision variable, since our aim is to compute all undefined weights at compile time. Moreover, if the set of inequations turns out to be unsatisfiable, the user will be warned about this fact during compilation.

Reformulation of Homogeneity Meta-Constraints

We reformulate the *homogeneity* meta-constraints by reifying the referenced constraints and constraining the number of satisfied constraints.

For example, the meta-constraint `atLeast(`$List$`,`p`)` is reformulated into:

```
Count(ListReif, True, n);
n >= val;
```

where `ListReif` is the list of Boolean variables resulting from reifying the constraints referenced in $List$, and `val` is computed at compile time and is equal to $\lceil length(List) * p/100 \rceil$. `Count(`$l$`,`$e$`,`$n$`)` is a `Simply` global constraint that is satisfied if and only if there are exactly n occurrences of the element e in the list l.

The meta-constraint `maxCost(`$ListOfLists$`,`val`)` is reformulated into:

```
Sum([ If_Then_Else (List[1][j]) (0) (weight_label[1][j])
      | j in [1..len[1]] ], vio_wei[1]);
vio_wei[1] =< val;
...
Sum([ If_Then_Else (List[n][j]) (0) (weight_label[n][j])
      | j in [1..len[n]] ], vio_wei[n]);
vio_wei[n] =< val;
```

where `n` is the length of *ListOfLists*, `len[i]` is the length of the `i`-th list in *ListOfLists*, `weight_label[i][j]` is the weight associated with the `j`-th label of the `i`-th list, `vio_wei[i]` denotes the aggregated weight of the violated constraints in the `i`-th list and *val* is the maximum allowed violation cost.

The reformulation of the `minCost(`*List*`,`*min*`)` meta-constraint is almost the same of the previous one, but replacing the `=<` for `>=` in the constraints `vio_wei[i] >= val;`.

The meta-constraint `homogeneousPercentWeight(`*ListOfLists*`,`*p*`)`, is reformulated into:

```
Sum([ weight_label[1][j] | j in [1..len[1] ], total_wei[1]);
Sum([ If_Then_Else (ListOfLists[1][j]) (0) (weight_label[1][j])
      | j in [1..len[1]] ], vio_wei[1]);
(vio_wei[1] * 100 Div total_wei[1]) >= min_homogen;
(vio_wei[1] * 100 Div total_wei[1]) =< max_homogen;

...

Sum([ weight_label[n][j] | j in [1..len[n] ], total_wei[n]);
Sum([ If_Then_Else (ListOfLists[n][j]) (0) (weight_label[n][j])
      | j in [1..len[n]] ], vio_wei[n]);
(vio_wei[n] * 100) Div total_wei[n]) >= min_homogen;
(vio_wei[n] * 100) Div total_wei[n]) =< max_homogen;
(max_homogen - min_homogen) < p;
```

where `len[i]` is the length of the `i`-th list in *ListOfLists*, `weight_label[i][j]` is the weight associated with the `j`-th label of the `i`-th list, `total_wei[i]` is the sum of weights of the labels in the `i`-th list and `n` is the length of *ListOfLists*. Note that according to the `Sum` constraints, `vio_wei[i]` denotes the aggregated weight of the violated constraints in the `i`-th list. Finally, `min_homogen` and `max_homogen` are fresh new variables.

Since we are restricted to linear integer arithmetic, the `total_wei[i]` expressions must be evaluable at compile time. This requires the weights of the constraints referenced by this meta-constraint to be evaluable at compile time.

The reformulation of `homogeneousAbsoluteWeight(`*ListOfLists*`,`*v*`)` is analogous to the previous one, but where, instead of computing the percentage on `vio_wei`[`i`], we can directly state:

```
...
vio_wei[1] >= min_homogen;
vio_wei[1] =< max_homogen;
...
vio_wei[n] >= min_homogen;
```

```
vio_wei[n] =< max_homogen;
(max_homogen - min_homogen) =< v;
```

Technically, our reformulation could allow the `homogeneousAbsoluteWeight` meta-constraint to reference constraints whose weight expression uses decision variables (and hence is not evaluable at compile time) without falling out of linear integer arithmetic. However, this is not the case for the meta-constraint `homogeneousAbsolutePercent`, for which we must be able to compute the aggregated weight of the labels. Thus, for coherence reasons, we have forbidden both constraints to reference constraints with decision variables in their weight, in addition to constraints with undefined (_) weight, as pointed out in Section 7.3.1.

The reformulations of the `homogeneousAbsoluteNumber` and `homogeneousPercentNumber` meta-constraints are similar to the previous ones, but where we count the number of violated constraints instead of summing their weights.

Reformulation of Dependence Meta-Constraints

The *dependence* meta-constraints are straightforwardly reformulated by applying the logical operators between constraints directly supported by `WSimply` on the corresponding reification variables.

7.4.2 Constraints reformulation

There are two steps in the constraints reformulation: how we reformulate the variables and their domains, and how we reformulate the constraints which define the model.

The first step is how we reformulate the `WSimply` variables and their domains into SMT variables. As we have seen in Section 7.3 `WSimply` only supports Boolean and integer variables, which are directly reformulated into SMT variables since `WSimply` uses LIA as background theory. In case of `WSimply` indexed variables (arrays or matrices) we only have to create as many base type SMT variables as elements contained in the indexed variable. Finally, for the integer variables with an assigned domain, we add bounding constraints as we have explained at Section 5.3.1 of the `fzn2smt` tool.

The second step of the constraint reformulation is the hard constraints reformulation, which can be divided in three groups: basic constraints, statements (basically generators and filters), and global constraints. Since `WSimply` uses LIA as background theory in the SMT formulas, the hard constraints reformulation is straightforward for the basic constraints, either arithmetic constraints (`<`, `>`, `=<`, `>=`, `<>` and `=`) and the Boolean constraints (`And`, `Or`, `Iff`, `Implies`, `Not` and `If_Then_Else`), and for the statements (`Forall` $\simeq$ `And`, `Exists` $\simeq$ `Or` and `If-Then-Else`). Note that the reformulation of comprehension list, which is also a constraint generator, is the same as the `Forall` statement. Finally, we only have to explain how to reformulate the global constraints, which is explained below.

Global constraints reformulation

In general, SMT solvers do not implement specific theories for global constraints, and implementing theories for global constraints is out of scope of this thesis (although it could be very interesting for future work). Therefore, in `WSimply`, global constraints are reformulated into conjunctions of basic constraints. Next we describe the reformulation of the global constraints supported by `WSimply`:

- The `AllDifferent(`$List$`)` is reformulated into:

$$\bigwedge \neg(e_i = e_j)$$

  where e_i and e_j are the i-th and j-th elements of $List$, and $i \neq j$.

- The `Sum(`$List$`,`val`)` is reformulated into:

$$\left(\sum e_i\right) = val$$

  where e_i is the i-th element of $List$, and val is an arithmetic expression. Note that val can contain decisional variables.

- The `Count(`$List$`,`ele`,`val`)` is reformulated into:

$$\left(\sum if\_then\_else(e_i = ele)(1)(0)\right) = val$$

  where e_i is the i-th element of $List$, val is the number of appearances of ele in $List$, ele is a formula or an arithmetic expression, and val is an arithmetic expression. Note that either ele or val can contain decisional variables.

- The `AtMost(`$List$`,`ele`,`exp`)` default reformulation is exactly the same as the `Count` reformulation where instead the operator $=$ in the outer condition we have the operator $<=$.

  `WSimply` also supports three other reformulations for the `AtMost`: the two first are adaptations of the pair-wise and sequential encodings (presented in Section 3.6.1), and the last one is based on a SAT encoding for pseudo-Boolean constraints using BDDs, which is explained in Chapter 8. In this last reformulation exp must be evaluable at compilation time.

- The `AtLeast(`$List$`,`ele`,`exp`)` default reformulation is exactly the same as the `AtMost` and the `Count` reformulations where the operator of the outer condition is $>=$.

  `WSimply` also supports the reformulation based on SAT encodings for pseudo-Boolean constraints using BDDs for the `AtLeast`, in this case, we first reformulate the `AtLeast` into an `AtMost` and then the same reformulation as above is applied.

- The `ExactlyOne(`$List$`)` is reformulated into a `AtMost` pair-wise encoding and a disjunction for the `AtLeast`, both presented in Section 3.6.1.

7.4.3 WCSP and COP reformulations

As we will see in Section 7.6, `WSimply` can solve WCSP instances using WSMT or using optimization over SMT. Therefore, in this section we first describe how WCSP instances are reformulated into WSMT instances (reformulation step R2 of Figure 7.6) and COP instances (reformulation step R3 of Figure 7.6), and finally how COP instances can be reformulated into WSMT instances (reformulation step R4 of Figure 7.6).

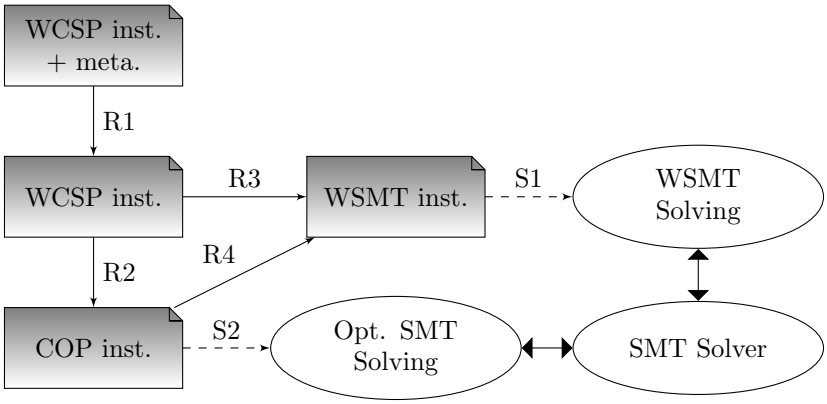

Figure 7.6: Basic architecture and SMT solving process of `WSimply`.

WCSP to COP

In order to transform our WCSP instance into a COP instance, we first replace each soft constraint $C_i \,@\, w_i$ by the following constraints where we introduce a fresh integer variable o_i:

$$\neg(w_i > 0 \ \wedge \ \neg C_i) \quad \rightarrow \quad o_i = 0 \tag{7.1}$$

$$(w_i > 0 \ \wedge \ \neg C_i) \quad \rightarrow \quad o_i = w_i \tag{7.2}$$

If the weight expression, w_i, evaluates to a value less or equal than 0, then the cost of falsifying C_i is 0, otherwise it is w_i. Since we are defining a minimisation problem we could actually replace Equation (7.1) with $o_i \geq 0$.[2]

Secondly, we introduce another fresh integer variable O, which represents the sum of the o_i variables, i.e., the optimisation variable of the COP to be minimised, and the following constraint:

$$O = \sum_{i=1}^{m} o_i \tag{7.3}$$

Finally, we keep the original hard constraints with no modification.

WCSP to WSMT

To the best of our knowledge, existing WSMT solvers only accept WSMT clauses whose weights are constants. Therefore, we need to convert the WCSP instance into a WSMT instance where all the WSMT clauses have a constant weight.

We apply the same strategy as in R2, i.e., we first replace each soft constraint $C_i \,@\, w_i$, where w_i is not a constant (involves variables), with the constraints 7.1 and 7.2 introducing a fresh integer variable o_i. Secondly, we add the following set of soft constraints over each possible value of each o_i variable:

$$\bigcup_{v_j \in V(o_i)} o_i \neq v_j \,@\, v_j \tag{7.4}$$

[2] `WSimply` has a command line parameter to disable this simplification.

where $V(o_i)$ is the set of all possible positive values of o_i. These values are determined by evaluating the expression for all the possible values of the variables, and keeping only the positive results. Note that at this point we do have a WCSP instance where all the soft constraints have a constant weight.

Finally, we replace each soft constraint $C_i \,@\, w_i$, with the WSMT clause (C'_i, w_i) where C'_i is the translation of C_i into SMT as described in the previous section. We also replace each hard constraint with its equivalent hard SMT clause.

COP to WSMT

Taking into account that the optimisation variable O of the COP instance is the integer variable that represents the objective function, we only need to add the following set of WSMT clauses: $\bigcup_{i=1}^{i=W}(O < i, 1)$, where W is the greatest value the objective variable can be evaluated to. A more concise alternative could result from using the binary representation of W, i.e., adding the set of WSMT clauses $\bigcup_{i=0}^{i<\lceil log_2(W+1)\rceil}(\neg b_i, 2^i)$, and the hard clause $(\sum_{i=0}^{i<\lceil log_2(W+1)\rceil} 2^i \cdot b_i = O, \infty)$.

We finally replace all the constraints of the COP instance with the equivalent hard SMT clauses as described in the previous section.

7.5 WCSP reformulation into PB/LP

Since a PB problem can be seen as a particular case of a LP problem where variables are pseudo-Boolean, the reformulation steps from a WCSP instance into a PB instance and into a LP instance are the same. Next we describe the reformulation steps in the same order as the previous section: first the meta-constraint elimination (equivalent to reformulation step R1 of Figure 7.1), then how to reformulate the hard constraints into PB/LP, and, finally, we describe the soft-constraints reformulation into a COP (equivalent to reformulation step R2 of Figure 7.1) for PB/LP and into a weighted PB instance (equivalent to reformulation step R3 of Figure 7.1).

7.5.1 Meta-constraints reformulation

The meta-constraint reformulation step for PB and LP reformulations is very similar to the equivalent step of the SMT reformulation: for each labelled soft constraint we introduce a new reification variable, using the bigM method described in Section 3.6.2, and then we proceed to reformulate the meta-constraints. The reformulation of priority and dependence meta-constraints is exactly the same as the previous section. But the homogeneity meta-constraints reformulation is a bit different and it only supports a subset of the meta-constraints.

Reformulation of Homogeneity Meta-Constraints

First of all we describe the `maxCost` and `minCost` meta-constraint reformulations, which is the same for both PB and LP reformulations.

The meta-constraint `maxCost(`*ListOfLists*`,`*val*`)` is reformulated into n `AtLeast` global constraints:

```
AtLeast([ (ListOfLists[1][j] * weight_label[1][j])
```

Table 7.1: LP (PB) reformulation of the logical constraints.

`Not a`	$i_a = 0$
`a Or b`	$i_a + i_b \geq 1$
`a And b`	$i_a + i_b = 2$
`a Implies b`	$i_b - i_a \geq 0$
`a Iff b`	$i_a - i_b = 0$
`a Xor b`	Not supported
`If_Then_Else(c)(a)(b)*`	$i_a - i_c \geq 0$ and $i_c + i_b \geq 1$

*If_Then_Else(c)(a)(b) is equivalent to (`c Implies a`) and (`c Or b`).

```
          | j in [1..len[1]] ], total_wei[1]-val);
...
AtLeast([ (ListOfLists[n][j] * weight_label[n][j])
          | j in [1..len[n]] ], total_wei[n]-val);
```

where `n` is the length of *ListOfLists*, `len[i]` is the length of the `i`-th list in *ListOfLists*, `weight_label[i][j]` is the weight associated with the `j`-th label of the `i`-th list, `total_wei[i]` is the aggregated weight of the labels in the `i`-th list and *val* is the maximum allowed violation cost. Note that we use `total_wei[1]-val` because *ListOfLists*$[i][j]$ is equal to 1 when constraint j is satisfied.

The reformulation of the `minCost(`*List*`,`*min*`)` meta-constraint is almost the same of the previous one, but using the global constraint `AtMost` instead of `AtLeast`.

The `homogeneousAbsoluteWeight` and `homogeneousAbsoluteNumber` meta-constraint reformulation is only supported by the LP reformulation and it is the same as the SMT reformulation.

Finally, meta-constraints `homogeneousPercentWeight` and `homogeneousPercentNumber` are not currently supported by neither of the PB and LP reformulations.

7.5.2 Constraints reformulation

Since LP (PB) constraints are inequalities on linear combinations of integer (0-1) variables, arithmetic constraints can be easily reformulated. We only have to transform them to the LP (PB) format, which means to have all variables at the left side of the inequality and a constant at the right side.

In contrast, logical constraints such as `Or`, `Implies`, ... cannot be directly translated. To be able to express these kind of constraints we use reification, which means to reflect the constraint satisfaction into a 0-1 variable. Then, e.g., `a Or b` can be translated into $i_a + i_b \geq 1$, where i_a and i_b are the reification 0-1 variables corresponding to the satisfaction of the formulas `a` and `b`, respectively. Table 7.1 shows the reformulation of the logical constraints into LP (PB).

The reformulation of the statements is very simple, the `Forall` and `If-Then-Else` are unfolded and treated as individual constraints, and the `Exists` are unfolded and then treated as disjunction (`Or`).

Global constraints reformulation

Next we describe the reformulation of the global constraints supported by `WSimply` for the LP/PB reformulations:

- The `AllDifferent(`*List*`)`, only available for the LP reformulation, is reformulated into:

$$\bigwedge(e_i - e_j \neq 0)$$

  where e_i and e_j are the i-th and j-th elements of *List*, and $i \neq j$.

- The `Sum(`*List*`,`*val*`)` is reformulated into:

$$\left(\sum e_i\right) = val$$

  where e_i is the i-th element of *List* and *val* is an arithmetic expression. In case that *val* contains decisional variables we have to move all the variables to the left side of the equality.

- The `Count(`*List*`,`*ele*`,`*val*`)` is reformulated into:

$$\left(\sum reif_i\right) = val$$

  where $reif_i$ is the reification variable (0/1) of the equality $e_i = ele$, e_i is i-th element of *List*, *val* is the number of appearances of *ele* in *List*, *ele* is a formula or an arithmetic expression, and *val* is an arithmetic expression. Note that either *ele* or *val* can contain decisional variables. In case that *val* contains decisional variables we have to move all the variables to the left side of the equality.

- The `AtMost(`*List*`,`*ele*`,`*exp*`)` is reformulated into:

$$\left(\sum e_i\right) \leq val$$

  where e_i is the i-th element of *List*, *val* is an arithmetic expression and the elements of the lists are restricted to expressions of the form $a * x$, where a is an integer constant and x is an integer variable. Note that in this version of the `AtMost` for the PB and LP reformulations, we ignore the second parameter *ele*.

- The `AtLeast(`*List*`,`*ele*`,`*exp*`)` reformulation is exactly the same as the `AtMost` reformulation where the operator of the outer condition is $\geq$.

7.5.3 WCSP and COP reformulations

In this section we present the WCSP reformulation into optimization PB and LP problems, and the WCSP reformulation into weighted PB.

WCSP to optimization PB/LP

In order to reformulate a WCSP into a optimization PB/LP problem, first, we reformulate the hard constraints as explained in the previous subsection and then we reformulate the soft constraints.

Since soft constraints are not directly supported by optimization PB and LP, we use reification to manage their violation costs. To do that, each soft constraint is reified and a reverse reification variable is created using the equation: $reif\_var + reif\_var_{rev} = 1$. Note that in this reverse reification variable we are reflecting the falsification of the constraint.

Finally, the minimization function is generated using these reverse reification variables multiplied by their respective violation cost.

WCSP to weighted PB

In order to reformulate a WCSP into a optimization PB/LP, first, we reformulate the hard constraints as explained in the previous section and then we reformulate the soft constraints. In this case, Since soft constraints are directly supported by weighted PB we replace each soft constraint $C_i \,@\, w_i$, with the PB clause (C'_i, w_i) where C'_i is the translation of C_i into PB as described in the previous subsection.

7.6 The WSimply solving module

In this section we describe the WSimply solving process depending on the type of reformulation chosen: PB, LP and SMT.

The solving process for the PB and LP reformulations is very simple, because both types of solvers can solve optimization instances, and in the case of PB they can also solve weighted PB instances. Therefore, once the instance is properly reformulated into weighted PB, optimization PB or optimization LP, that instance can be feed into a solver (PB or LP) and it will solve the instance.

The solving process for the SMT reformulation is a bit more complicated. Next we describe the two possible solving approaches that we can currently apply for the SMT reformulation (described in Figure 7.1): WSMT solving which receives as input a WSMT instance (named S1 in the figure) and Optimization SMT solving which receives as input a COP instance (named S2 in the figure). In both approaches, WSimply uses by default the Yices 1 SMT solver through API. However, WSimply can also solve optimization SMT instances using external SMT solvers by generating SMT instances in the SMT-Lib formats (v1.2 and v2).

In the first SMT solving approach, we already have the WCSP instance translated into a WSMT instance and we only have to choose which WSMT solving method we want to use from the two next. The first WSMT solving method, is a non-exact algorithm[3] offered by default in the Yices SMT solver [DdM06b]. While the second WSMT solving method, is the adapted version of the WPM1 algorithm for SMT described in Section 6.2.2. Recall that this method calls the SMT solver incrementally and is based on the detection of unsatisfiable cores.

In the second SMT solving approach, we have adapted the dichotomic minimization algorithm for SMT described in Section 6.2.2 to be used in WSimply. We first translate all the constraints of the COP instance into SMT clauses and send them to the SMT solver. Then,

[3]Non-exact algorithms do not guarantee optimality.

before every call we bound the domain of the optimisation variable O by adding the SMT clause $O \leq k$, where k is an integer constant. If the SMT solvers returns `unsat`, we replace the clause $O \leq k$ by $O > k$. The strategy we use to determine the next k is a binary search.

7.7 Experimental results

In this section we present several experimental results of different versions of `WSimply`. First of all, we present several experimental results of `Simply`, the first version of the tool: solving four CSPs with different SMT solvers, a comparison between `Simply` and several similar tools on solving 18 CSPs, and a comparison between `WSimply` and `fzn2smt` solving the same 18 CSPs. Finally, we present experimental results of `WSimply` for solving WCSPs with meta-constraints first by using optimization SMT and WSMT, and then by using PB solvers.

7.7.1 `Simply` first version

Next we present the experimental results of the first version of `Simply` for solving classical CSPs using different SMT solvers presented at [BPSV09].

We have used four classical CSPs in this experiment: *Schur's Lemma (CSPLib [GW99] problem 015)*, *Balanced Academic Curriculum Problem (BACP) (CSPLib problem 030)*, *Job-shop scheduling* and the *Queens* problem.

We have used the SMT solvers which participated in the QF_LIA division of the Satisfiability Modulo Theories Competition[4] (SMT-COMP) 2008, namely, Z3.2, MathSAT-4.2, CVC3-1.5, Barcelogic 1.3 and Yices 1.0.10, for solving the SMT version of the previous problems generated using `Simply`.[5]

In addition, we have also used some solvers of different nature on the same problems, namely, G12 MiniZinc 0.9, ECL^{i}PS^e 6.0, SICStus Prolog 4.0.7, SPEC2SAT 1.1 and Comet 1.2, in order to have a comparison. The same benchmarks have been used for G12 MiniZinc 0.9, ECL^{i}PS^e 6.0 and SICStus Prolog 4.0.7 (after a translation from the MiniZinc modeling language to the FlatZinc low-level solver input language, by using the MiniZinc-to-FlatZinc translator `mzn2fzn`). SPEC2SAT transforms problem specifications written in NP-SPEC into SAT instances in DIMACS format, and thus can work in cooperation with any SAT solver supporting that format. In our tests, we have used SPEC2SAT together with zChaff 2007.3.12. With respect to Comet, only its constraint programming module has been tested. In order to have a fair comparison, we have preserved as much as possible the modeling used in SMT and avoided the use of any search strategy when dealing with other solvers, as no search control is possible within SMT solvers.

Table 7.2 shows the time in seconds spent by each solver in each problem, with a timeout of 1800 seconds. The benchmarks were executed on a 3.00 GHz Intel Core 2 Duo machine with 2 Gb of RAM running under GNU/Linux 2.6. The first column refers to the `Simply` compilation time. The following 5 columns contain the solving time spent by the different SMT solvers on the generated SMT instances. The rest of columns detail the times (including compilation and solving) spent by solvers of other nature. We can observe the following:

[4] SMT-COMP: The SAT Modulo Theories Competition (`http://www.smtcomp.org`).

[5] Since our naive modeling of the *Queens* problem falls into the QF_IDL fragment of QF_LIA, we have made this explicit in the generated instances. In this way, the SMT solvers with a specialized theory solver for QF_IDL can take profit of it.

Table 7.2: Time in seconds spent on solving each problem instance. The first column refers to `Simply` compilation time. Time out (t.o.) was set to 1800 seconds. Memory out is denoted by m.o.

	Simply + SMT solver						Other tools				
	Simply (compilation)	Z3.2	MathSAT-4.2	CVC3-1.5	Barcelogic 1.3	Yices 1.0.10	G12 MiniZinc 0.9	mzn2fzn + ECL^{i}PS^{e}6.0	mzn2fzn + SICStus 4.0.7	SPEC2SAT 1.1	Comet 1.2
Queens_50	0.22	t.o.	53.00	m.o.	11.72	29.47	0.22	2.04	6.98	248.01	t.o.
Queens_100	0.72	t.o.	t.o.	m.o.	389.04	19.22	0.84	t.o.	28.51	t.o.	t.o.
Queens_150	1.54	t.o.	t.o.	m.o.	995.94	t.o.	150.40	t.o.	256.18	t.o.	t.o.
Bacp_12_6	0.17	0.55	2.53	t.o.	56.98	0.19	0.84	t.o.	3.8	m.o.	268.56
Bacp_12_7	0.18	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	m.o.	t.o.
Bacp_12_8	0.22	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	m.o.	t.o.
Bacp_12_9	0.21	0.27	10.86	t.o.	314.91	0.64	0.94	t.o.	5.3	m.o.	0.51
Bacp_12_10	0.24	0.24	14.97	t.o.	190.10	0.79	1.44	t.o.	6.02	m.o.	0.60
Bacp_12_11	0.24	0.27	13.60	t.o.	237.50	1.24	1.70	t.o.	7.97	m.o.	19.56
Bacp_12_12	0.26	0.48	13.24	t.o.	338.46	1.32	40.59	t.o.	11.32	m.o.	t.o.
Schurl_12_3	0.06	0.01	0.08	7.91	0.03	0.02	t.o.	t.o.	0.24	0.38	0.39
Schurl_13_3	0.08	0.04	0.07	14.30	0.05	0.05	t.o.	t.o.	0.28	0.55	0.40
Schurl_14_3	0.09	0.23	0.50	18.24	0.12	0.16	t.o.	t.o.	0.32	0.50	0.40
Schurl_15_3	0.10	0.35	0.79	29.15	0.15	0.18	t.o.	t.o.	0.37	0.73	0.40
Jobshop_54	0.31	0.12	0.27	104.97	7.79	2.69	34.13	1.54	33.30	80.41	1.79
Jobshop_55	0.32	0.20	0.35	211.48	11.57	3.63	122.16	2.16	t.o.	80.03	11.65
Jobshop_56	0.30	0.12	0.46	358.53	12.08	4.37	396.03	3.13	t.o.	88.11	100.01
Jobshop_57	0.30	0.34	0.89	475.55	16.05	6.62	1115.09	1.13	t.o.	85.66	892.54
Jobshop_58	0.34	0.10	0.25	134.71	20.75	11.48	0.09	1.22	236.64	95.11	0.82

For the *Queens* problem, G12 obtains the best results. The poor performance of SMT solvers on this problem is probably due to the fact that the modeling results in a SMT instance with only unit clauses, while SMT is well-suited for problems whose satisfiability highly depends on the combination of the Boolean structure and the background theory.

For the *BACP* problem, similar results are obtained by SMT solvers, G12 and SICStus Prolog. However, complicated instances around the phase transition are not solved by any of them. The SMT instances generated by `Simply` for this problem include a relevant Boolean structure mixed with arithmetic expressions, mainly due to the `Count` and `Sum` global constraints. This is probably why some SMT solvers obtain extremely good results on this problem. It is also important to notice that SPEC2SAT fails to solve all the instances, since the generated SAT formula is too big for zChaff. Moreover, Comet is very unstable on this problem.

For the *Schur's Lemma* problem, good results are obtained by most of the solvers. Surprisingly, G12 consumes all the time with no answer. With the chosen modeling, `Simply` is able to generate an SMT instance with no arithmetic at all, since all expressions can be

evaluated at compile time.

For the *Job-shop* problem, best results are obtained by some of the SMT solvers and by ECL^{i}PS^e. This is again a problem with a relevant Boolean structure, and some arithmetic.

Globally, it seems that most of the SMT solvers are good in all the problems considered. This is especially relevant if we take into account that those solvers come from the verification arena and, therefore, have not been designed with those kind of constraint problems in mind. Moreover, they seem to scale up very well with the size of the problems. Let us remark that these problems are just the first ones at hand that we have considered, i.e., we have not artificially chosen them. For this reason, SMT can be expected to provide a nice compromise between expressivity and efficiency for solving CSPs in some contexts.

A deeper experimental comparison is done with the Yices 2 proto SMT solver with the FlatZinc solvers: G12, ECL^{i}PS^e, Gecode and FznTini. In this experiment we have taken 18 problems from MiniZinc library, which each problem has several instances having 208 instances in total, and each problem has been written in `Simply` trying to preserve the original encoding.

Table 7.3 shows the time in seconds spent by each solver in each problem. The timeout of each instance is set at 300 seconds. The benchmarks were executed on a 3.00 GHz Intel Core 2 Duo machine with 2 Gb of RAM running under GNU/Linux 2.6. The first column is the name of the problem. The second column indicates if the problem is a satisfaction or optimization problem. The third column is the total number of instances of each problem. And the last 5 columns are the aggregated solving time of each solver of the solved instances. In this experiments all the solver times include the compilation and solving time. The number of solved instances is indicated between parentheses. In bold we have marked the "best" result of each problem, which means, the solver that has solved more instances in less time. The last row indicates the total time spend by each solver and between parentheses the total number of instances solve.

As we can see, `Simply` +Yices 2 solves 110 instances, nearly followed by G12. But `Simply` spends more time in solving them, 3500 seconds versus 900 seconds, and seems that the difference comes from the *cars* problem where `Simply` solves 19 instances and G12 only solves 1 instance with a difference of 2400 seconds. We can also see, that `Simply` has better results in 3 of the 18 problems, while G12 has better results in 5 problems and Gecode has better results in 9 of the 18 problems.

Finally, in Table 7.4, we present a comparison between the tools `Simply` and `fzn2smt`, both using the same SMT solver: Yices 2 proto. The first column of the table is the problem name. The second column is the problem type: satisfiability problem (s) or optimization problem (o). The third column is the number of instances of the problem. And the two last columns are the total solving time spend by each tool and between parentheses the number of instances solved. The time out is 300 seconds, and the time of the non solved instances is not included. In bold we mark the "best" result, which is the one with more instances solved and in case of tie the result with smaller total time.

As the table shows, both tools, `Simply` and `fzn2smt`, solve more or less the same number of instances, 110 and 111 respectively. Where we find more difference is at the solving time, where `fzn2smt` spends 700 seconds less than `Simply` to solve the same number of instances plus one. This is reasonable, since some preprocessing techniques are used in the translation from MiniZinc to FlatZinc, whereas they are not used in the `Simply` reformulation into SMT. But, in the other side, `Simply` has better results in more problems than `fzn2smt`, solving more instances (or solving them faster when there is draw in the number of instances

Table 7.3: For 18 problems from the MiniZinc library we report the total solving time and the number of solved instances (in parentheses) using `Simply` +Yices 2 proto and several FlatZinc solvers. Time of the non solved instances is not included. The timeout is set at 300 seconds. For each problem we mark in boldface the solver that has solved more instances in less time. Optimization problems are denoted with 'o' and satisfiability problems are denoted by 's'. Column # indicates the number of instances per problem.

Problem		#	G12	ECL^{i}PS^e	Gecode	FznTini	`Simply`
alpha	s	1	**0.01(1)**	0.47(1)	0.07(1)	0.60(1)	0.21(1)
areas	s	4	0.13(4)	2.04(4)	**0.05(4)**	0.38(4)	0.63(4)
bibd	s	14	118.16(12)	4.18(7)	35.58(7)	**353.78(13)**	26.63(11)
cars	s	79	0.02(1)	0.81(1)	295.64(3)	1.21(1)	**2420.89(19)**
curriculum	s	3	0.27(2)	95.09(2)	261.82(1)	8.70(3)	**1.26(3)**
eq	s	1	**0.01(1)**	0.49(1)	**0.01(1)**	20.14(1)	0.24(1)
golomb	o	10	**251.18(9)**	85.84(8)	23.83(8)	23.87(6)	66.05(6)
langford	s	25	52.19(20)	121.85(20)	34.54(20)	310.24(18)	**12.13(20)**
latin-squares	s	7	**5.98(6)**	12.54(6)	15.35(6)	129.40(3)	1.45(4)
magicseq	s	9	25.17(7)	21.56(7)	**7.39(7)**	0.30(3)	106.06(5)
nmseq	s	10	281.73(6)	-(0)	**171.03(7)**	-(0)	213.73(2)
photo	o	2	0.10(2)	1.07(2)	**0.04(2)**	0.08(2)	0.66(2)
quasigroup7	s	10	**1.37(5)**	293.67(3)	3.65(5)	380.28(3)	7.74(5)
queens	s	7	88.88(7)	**36.24(7)**	0.52(3)	94.76(4)	110.17(5)
schur_numbers	s	3	1.30(3)	1.03(2)	**0.30(3)**	0.02(2)	0.53(3)
shortest_path	o	10	4.36(4)	292.15(6)	**141.34(7)**	-(0)	88.65(6)
slow_convergence	s	10	68.74(10)	12.84(7)	**11.03(10)**	36.98(4)	479.71(10)
tents	s	3	0.10(3)	1.52(3)	**0.04(3)**	0.30(3)	0.68(3)
Total		208	899.7(103)	983.39(87)	1002.23(98)	1361.04(71)	**3537.42(110)**

solved) in 12 of the 18 problems.

7.7.2 `WSimply` with meta-constraints

In order to show the usefulness of meta-constraints we conducted several experiments on a set of instances of the *Nurse Rostering Problem* (NRP) and on a variant of the *Balanced Academic Curriculum Problem* (BACP). Next we present the experimental results of `WSimply` using Yices 1 API, presented at [ABP+13].

NRP

The NRP (also known as Nurse Scheduling Problem (NSP)) is a classical constraint programming problem [WHFP95]. Each nurse has his/her preferences on which shifts wants to work. Conventionally a nurse can work in three shifts: day shift, night shift and late night shift. There is a required minimum of personnel for each shift (the shift cover). The problem is described as finding a schedule that both respects the hard constraints and maximizes the nurse satisfaction by fulfilling their wishes.

We have taken the instances from the NSPLib. The NSPLib is a repository of thousands of NSP instances, grouped in different sets and generated using different complexity indicators: size of the problem (number of nurses, days or shift types), shifts coverage (distributions over the number of needed nurses) and nurse preferences (distributions of the preferences over the

Table 7.4: For 18 problems from the MiniZinc library we report the total solving time and the number of solved instances (in parentheses) using `Simply` and `fzn2smt`, both with the Yices 2 proto. Time of the non solved instances is not included. The timeout is set at 300 seconds. For each problem we mark in boldface the solver that has solved more instances in less time. Optimization problems are denoted with 'o' and satisfiability problems are denoted by 's'. Column # indicates the number of instances per problem.

Problem		#	`fzn2smt`	`Simply`
alpha	s	1	0.66(1)	**0.21(1)**
areas	s	4	4.75(4)	**0.63(4)**
bibd	s	14	**79.48(12)**	26.63(11)
cars	s	79	**2271.17(21)**	2420.89(19)
curriculum	s	3	9.97(3)	**1.26(3)**
eq	s	1	0.47(1)	**0.24(1)**
golomb	o	10	**41.88(6)**	66.05(6)
langford	s	25	50.52(20)	**12.13(20)**
latin-squares	s	7	7.54(4)	**1.45(4)**
magicseq	s	9	11.01(4)	**106.06(5)**
nmseq	s	10	1.42(1)	**213.73(2)**
photo	o	2	0.78(2)	**0.66(2)**
quasigroup7	s	10	31.51(5)	**7.74(5)**
queens	s	7	**54.61(5)**	110.17(5)
schur_numbers	s	3	1.45(3)	**0.53(3)**
shortest_path	o	10	**45.79(6)**	88.65(6)
slow_convergence	s	10	**222.37(10)**	479.71(10)
tents	s	3	2.50(3)	**0.68(3)**
Total		208	**2837.88(111)**	3537.42(110)

shifts and days). Details can be found in [VM07]. In order to reduce the number of instances to work with, we have focused on a particular set: the N25 set, which contains 7920 instances.

The N25 set has the following settings:

- Number of nurses: 25
- Number of days: 7
- Number of shift types: 4 (including the free shift)
- Shift covers: minimum number of nurses required for each shift and day.
- Nurse preferences: a value between 1 and 4 (from most desirable to least desirable) for each shift and day, for each nurse.

The NSPLib also has several complementary files with more precise information like minimum and maximum number of days that a nurse should work, minimum and maximum number of consecutive days, etc. We have considered the most basic case (case 1) which only constrains that

- the number of working days of each nurse must be exactly 5.

In Figure 7.7 we propose a `WSimply` model for the NSP to be solved using SMT where we have the shift covers and the number of working days as hard constraints, and the nurse preferences as soft constraints.

```
Problem:nrp
 Data
   int n_nurses;
   int n_days;
   int n_shift_types;
   int covers[n_days, n_shift_types];
   int prefs[n_nurses, n_days, n_shift_types];
   int min_shifts;
   int max_shifts;
 Domains
   Dom dshifts_types = [1..n_shift_types];
   Dom dshifts  = [min_shifts..max_shifts];
   Dom dnurses = [0..n_nurses];
 Variables
   IntVar nurse_day_shift[n_nurses, n_days]::dshifts_types;
   IntVar nurse_working_shifts[n_nurses]::dshifts;
   IntVar day_shift_nurses[n_days, n_shift_types]::dnurses;
 Constraints
   %%% Every nurse must work only one shift per day.
   %%% This constraint is implicit in this modelling.

   %%% The minimum number of nurses per shift and day must be covered.
   %%% Variables day_shift_nurses[d,st] will contain the number of nurses working
   %%%  for every shift and day.
   Forall(d in [1..n_days], st in [1..n_shift_types]) {
     Count([nurse_day_shift[n,d] | n in [1..n_nurses]], st, day_shift_nurses[d,st]);
   };
   [day_shift_nurses[d,st] >= covers[d,st] | d in [1..n_days], st in [1..n_shift_types]];
   %%% Nurse preferences are desirable but non-mandatory.
   %%% Each preference is posted as a soft constraint with
   %%%  its label (#prefs[n,d,st]) and a violation cost according to prefs[n,d,st].
   Forall(n in [1..n_nurses], d in [1..n_days], st in [1..n_shift_types]) {
     #prefs[n,d,st]: (Not (nurse_day_shift[n,d] = st)) @ {prefs[n,d,st]};
   };
   %%% The minimum and maximum number of working days of each nurse
   %%%  must be between bounds (i.e. the domain of nurse_working_shifts[n]).
   Forall(n in [1..n_nurses]) {
     Count([ nurse_day_shift[n,d] <> n_shift_types | d in [1..n_days] ],
           True, nurse_working_shifts[n]);
   };
```

Figure 7.7: `WSimply` (SMT) model for the NRP.

Since the addition of homogeneity meta-constraints in these particular instances significantly increases their solving time, we have ruled out the instances taking more than 60 seconds to be solved with `WSimply` without the meta-constraints. The final chosen set consists of 5113 instances.

In order to evaluate the effects of the homogeneity meta-constraints on the quality of the solutions and the solving times, we have conducted an empirical study of some instances from the NSPLib. In the following we report the results of several experiments performed with `WSimply` over the set of 5113 chosen instances of the NRP, using a cluster with nodes with CPU speed 1GHz and 500 MB of RAM, and with a timeout of 600 seconds. We tested the three solving approaches available in `WSimply` using SMT:

- dico: a binary search bounding an objective function
- yices: the default Yices 1 WSMT solving method
- core: our addapted version of the WPM1 for WSMT

The times appearing in the tables are for the core approach, which was the one giving best results for this problem.

Table 7.5 shows the results of the chosen 5113 instances from the N25 set without homogeneity meta-constraints.

Table 7.5: Results of 5113 instances from the N25 set, with soft constraints on nurse preferences (without meta-constaints). μ: mean of means of costs of violated constraints per nurse; σ: standard deviation of means of costs of violated constraints per nurse; Time: mean solving time (in seconds); Cost: mean optimal cost; Abs. diff.: mean of differences between maximal and minimal costs; Rel. diff.: mean of differences between relative percentual maximal and minimal costs.

	μ	σ	Time	Cost	Abs. diff.	Rel. diff.
Normal	9.67	1.83	6.46	241.63	7.71	9.42

We can observe that if we only focus on minimising the cost of the violated constraints, we can penalise some nurses much more than others. For instance, we could assign the least preferred shifts to one nurse while assigning the most preferred shifts to others. From the results in Table 7.5, we observe that the mean of absolute differences is 7.71 while the mean cost per nurse is around 9.67, which shows that the assignments are not really fair. In order to enforce fairness, we can extend the model of Figure 7.7 by adding homogeneity meta-constraints over the soft constraints on nurse preferences, as shown in Figure 7.8.

```
%%% Ask for homogeneity with factor F, over the lists of
%%%  soft constraints on nurse preferences.
homogeneousAbsoluteWeight([ [ prefs[n,d,st] | d in [1..n_days],
                              st in [1..n_shift_types] ]
                          | n in [1..n_nurses] ], F);
```

Figure 7.8: `WSimply` constraints to add to the NRP model in order to ask for homogeneity with factor `F` in the solutions.

Table 7.6 shows the results after adding to the model of Figure 7.7 the meta-constraint `homogeneousAbsoluteWeight` of Figure 7.8, with factor `F` = 5, while Table 7.7 shows the results for the same meta-constraint with factor `F` = 10. Note that this factor represents the maximal allowed difference between the penalisation of the most penalized nurse and the least penalised nurse. From Table 7.5 we know that the mean of these differences among the chosen NRP instances is 7.71. The first row (Absolute 5) shows the results for the solved instances (2478 out of 5113) within the timeout. The second row shows the results without the meta-constraint, for the solved instances (i.e., it is like Table 7.5 but restricted to these 2478 instances).

As we can observe, we reduce the absolute difference average from 4.81 to 4.32, which is a bit more than 10%. In particular, we reduce the absolute difference between the most penalised nurse and the least penalised nurse in 892 instances out of 2478. In contrast, the average penalisation per nurse increases from 8.80 to 8.92, but this is just 1.36%. The average global cost also increases, but only from 220.03 to 222.89. Hence, it seems reasonable to argue that it pays off to enforce homogeneity in this setting, at least for some instances. However,

Table 7.6: Results when adding the `homogeneousAbsoluteWeight` meta-constraint with factor 5. Statistics for the 2478 solved instances with a timeout of 600 seconds. μ: mean of means of costs of violated constraints per nurse; σ: standard deviation of means of costs of violated constraints per nurse; Time: mean solving time (in seconds); Cost: mean optimal cost; Cost (TO): mean of best lower bounds for those instances that exceeded the timeout; Abs. diff.: mean of differences between maximal and minimal costs; #improved: number of instances with improved absolute difference.

	μ	σ	Time	Cost	Cost (TO)	Abs. diff.	#improved
Absolute 5	8.92	0.96	43.28	222.89	272.93	4.32	892
Normal	8.80	1.07	5.98	220.03	261.94	4.81	-

when homogeneity is enforced the solving time increases, since the instances become harder (there are 2635 instances which could not be solved within the timeout).

The conclusion is that an homogeneity factor `F` $= 5$ may be too restrictive. Therefore, we repeated the experiment but with a factor `F` $= 10$. The results are shown in Table 7.7.

Table 7.7: Results when adding the `homogeneousAbsoluteWeight` meta-constraint with factor 10. Statistics for the 4167 solved instances with a timeout of 600 seconds. μ: mean of means of costs of violated constraints per nurse; σ: standard deviation of means of costs of violated constraints per nurse; Time: mean solving time (in seconds); Cost: mean optimal cost; Cost (TO): mean of best lower bounds for those instances that exceeded the timeout; Abs. diff.: mean of differences between maximal and minimal costs; #improved: number of instances with improved absolute difference.

	μ	σ	Time	Cost	Cost (TO)	Abs. diff.	#improved
Absolute 10	9.32	1.46	13.86	233.01	285.83	6.29	377
Normal	9.31	1.47	6.16	232.83	280.40	6.35	-

In this case only 946 out of 5113 could not be solved within the timeout. Although fewer instances are improved (377) the difference in the solving time really decreases and the mean of the best lower bounds for the unsolved instances is closer to the optimal value of the original instances. This suggests that it is possible to find a reasonable balance between the quality of the solutions and the required solving time with respect to the original problem.

Depending on the preferences of the nurses, the absolute difference may not be a good measure for enforcing homogeneity. Nurse preferences are weighted with a value between 1 and 4 (from most desirable to least desirable shifts). Imagine a nurse who tends to weight with lower values than another. Then, even if this nurse has many unsatisfied preferences, her total penalisation could be lower than that of one of the other nurses with fewer unsatisfied preferences. Therefore, it seems more reasonable to compute the relative difference, as it allows the relative degree of unsatisfied preferences to be compared.

Table 7.8 shows the results for the meta-constraint `homogeneousPercentWeight` with factor 6, which means that the relative percent difference between the most penalised nurse and the least penalised nurse must be less than or equal to 6. The first row (Percent 6) shows the results for the solved instances (2109 out of 5113) within the timeout. The second row shows the results without the meta-constraint for those solved instances.

The mean of the percent differences is reduced from 7.76 to 5.26, which is almost 32%. In particular, we reduce the percent difference between the most penalised nurse and the least

Table 7.8: Results when adding the `homogeneousPercentWeight` meta-constraint with factor 6. Statistics for the 2109 solved instances with a timeout of 600 seconds. μ: mean of means of costs of violated constraints per nurse; σ: standard deviation of means of costs of violated constraints per nurse; Time: mean solving time (in seconds); Cost: mean optimal cost; Cost (TO): mean of best lower bounds for those instances that exceeded the timeout; Rel. diff.: mean of differences between relative percentual maximal and minimal costs; #improved: number of instances with improved relative difference.

	μ	σ	Time	Cost	Cost (TO)	Rel. diff.	#improved
Percent 6	9.39	1.10	89.47	234.72	263.06	5.26	1875
Normal	9.14	1.30	5.27	228.56	250.81	7.72	-

penalized nurse in 1875 instances out of 2109. The average penalisation per nurse increases from 9.14 to 9.39, just 2.74%, and the average global cost only increases from 228.56 to 234.72. However, the average solving time increases from 5.27 to 89.47 seconds for the solved instances. In fact, the solving time increases, no doubt, by much more than this on average if considering the timed-out instances.

As with the experiments for the absolute difference, we have conducted more experiments, in this case increasing the factor from 6 to 11. The results are reported in Table 7.9. In this case, only 492 out of 5113 could not be solved within the timeout. The number of improved instances decreases but the solving time improves. Therefore, with the `homogeneousPercentWeight` meta-constraint we can also find a reasonable balance between the quality of the solutions and the required solving time with respect to the original problem.

Table 7.9: Results when adding the `homogeneousPercentWeight` meta-constraint with factor 11. Statistics for the 4621 solved instances with a timeout of 600 seconds. μ: mean of means of costs of violated constraints per nurse; σ: standard deviation of means of costs of violated constraints per nurse; Time: mean solving time (in seconds); Cost: mean optimal cost; Cost (TO): mean of best lower bounds for those instances that exceeded the timeout; Rel. diff.: mean of differences between relative percentual maximal and minimal costs; #improved: number of instances with improved relative difference.

	μ	σ	Time	Cost	Cost (TO)	Rel. diff.	#improved
Percent 11	9.62	1.67	33.28	240.38	269.14	8.42	1592
Normal	9.57	1.71	5.51	239.23	264.20	9.04	-

SBACP

The *Balanced Academic Curriculum Problem* (BACP) consists of assigning courses to academic periods while satisfying prerequisite constraints between courses and balancing the workload (in terms of credits) and the number of courses in each period [CM01, HKW02]. In particular, given

- a set of courses, each of them with an associated number of credits representing the academic effort required to successfully follow it,
- a set of periods, with a minimum and maximum bound both on the number of courses and number of credits assigned to each period,

- and a set of prerequisites between courses stating that, if a course c has as prerequisite a course d, then d must be taught in a period previous to the one of c,

the goal of the BACP is to assign a period to every course which satisfies the constraints on the bounds of credits and courses per period, and the prerequisites between courses. In the optimisation version of the problem, the objective is to improve the balance of the workload (amount of credits) assigned to each period. This is achieved by minimising the maximum workload of the periods.

There may be situations where the prerequisites make the instance unsatisfiable. We propose to deal with unsatisfiable instances of the decision version of the BACP by relaxing the prerequisite constraints, i.e., by turning them into soft constraints. We allow the solution to violate a prerequisite constraint between two courses but then, in order to reduce the pedagogical impact of the violation, we introduce a new hard constraint, the corequisite constraint, enforcing both courses to be assigned to the same period. We call this new problem *Soft Balanced Academic Curriculum Problem* (SBACP).

The goal of the SBACP is to assign a period to every course which minimises the total amount of prerequisite constraint violations and satisfies the conditionally introduced corequisite constraints, and the constraints on the number of credits and courses per period.

In Figure 7.9 we propose a modelling of the SBACP using `WSimply` to be solved using SMT. In order to obtain instances of the SBACP, we have over-constrained the BACP instances from the MiniZinc [Nic07] repository, by reducing the number of periods to four, and proportionally adapting the bounds on the workload and number of courses in each period. With this reduction on the number of periods, we have been able to turn all these instances into unsatisfiable.

In the following we present several experiments with `WSimply`, using the same three methods: dico, core and yices, over the obtained SBACP instances, using a 2.6GHz Intel® Core™ i5, with a timeout of 600 seconds.

The best solving approach for this problem in our system is yices, closely followed by dico. The core solving approach is not competitive for this problem. The performance of the WPM1 algorithm strongly depends on the quality of the unsatisfiable cores the SMT solver is able to return at every iteration. This quality has to do, among other details, with the size of the core, the smaller the better, and the overlapping of the cores, the lower the better. For the SBACP instances, the SMT solver tends to return cores which involve almost all the soft clauses, i.e., they are as big as possible and they completely overlap. This clearly degrades the performance of the WPM1 algorithm.

Columns two to five of Table 7.10 show the results obtained by `WSimply` on our 28 instances. The second column shows the required CPU time in seconds (with the yices solving approach); the third column indicates the total amount of prerequisite violations, and the fourth and fifth columns show the maximum and minimum number of prerequisite constraint violations per course. This maximum and minimum exhibit the lack of homogeneity of each instance. Column six shows the time obtained by CPLEX for solving the SBACP instances.

As we can observe, there are instances which have courses with three, four, and even five prerequisite constraint violations, as well as courses with zero violations. It would be more egalitarian to obtain solutions where the difference in the number of prerequisite constraint violations between courses is smaller. Thanks to the `homogeneousAbsoluteNumber` meta-constraint we can easily enforce this property of the solutions, as shown in Figure 7.10.

```
Problem:sbacp
 Data
    int n_courses;
    int n_periods;
    int load_per_period_lb;
    int load_per_period_ub;
    int courses_per_period_lb;
    int courses_per_period_ub;
    int course_load[n_courses];
    int n_prereqs;
    int prereqs[n_prereqs,2];
 Domains
    Dom dperiods=[1..n_periods];
    Dom dload=[load_per_period_lb..load_per_period_ub];
    Dom dcourses=[courses_per_period_lb..courses_per_period_ub];
 Variables
    IntVar course_period[n_courses]::dperiods;
    IntVar period_load[n_periods]::dload;
    IntVar period_courses[n_periods]::dcourses;
 Constraints
 %%% Hard Constraints
 %%%
 %%% Every course must be assigned to exactly one period.
 %%% This constraint is implicit in this modelling.
 %%%
 %%% The number of courses per period must be within bounds
 %%%  (i.e. the domain of period_courses[p]).
     Forall(p in [1..n_periods]) {
       Count([ course_period[c] | c in [1..n_courses] ], p, period_courses[p]);
     };
 %%% The workload in each period must be within bounds
 %%%  (i.e. the domain of period_load[p]).
     Forall(p in [1..n_periods]) {
       Sum([If_Then_Else(course_period[c] = p) (course_load[c]) (0)
            | c in [1..n_courses]], period_load[p]);
     };
 %%% If a prerequisite is violated then the corequisite constraint is mandatory.
     Forall(np in [1..n_prereqs]) {
       Not(pre[np])
       Implies (course_period[prereqs[np,1]] = course_period[prereqs[np,2]]);
     };
 %%% Soft Constraints
 %%%
 %%% Prerequisites are desirable but non-mandatory.
 %%% Each prerequisite (np) is posted as a soft constraint with
 %%%  its label (pre[np]) and a violation cost of 1.
     Forall(np in [1..n_prereqs]) {
       #pre[np]: (course_period[prereqs[np,1]] > course_period[prereqs[np,2]]) @ 1;
     };
```

Figure 7.9: `WSimply` model for the SBACP.

```
 %%% We enforce homogeneity with factor F, over the lists of
 %%%  soft constraints on prerequisites for each course.
 homogeneousAbsoluteNumber([ [ pre[np] | np in [1..n_prereqs], prereqs[np,1] = c ]
                           | c in [1..n_courses] ], F);
```

Figure 7.10: `WSimply` constraints to add to the model for the SBACP in order to ask for homogeneity with factor `F` in the solutions.

Table 7.10: Results of the experiments on the SBACP instances without and with homogeneity. Numbers in boldface denote instance improvements while maintaining the same cost. The last row shows the median of CPU solving time and the sum of the costs found; in the homogeneity cases we show the aggregated increment of the cost with respect to the original instances.

						Homogeneity factor 1				Homogeneity factor 2			
N.	Time	Cost	V. per c.		CPLEX	Time	Cost	V. per c.		Time	Cost	V. per c.	
			Max	Min				Max	Min			Max	Min
1	0.63	19	2	0	0.45	0.26	21	1	0	0.75	19	2	0
2	4.27	16	2	0	0.39	0.27	42	2	1	1.61	16	2	0
3	0.78	17	2	0	0.83	0.26	19	1	0	1.70	17	2	0
4	32.93	28	4	0	0.69	0.26	unsatisfiable			3.69	28	**2**	0
5	0.97	15	2	0	0.43	0.25	39	2	1	0.86	15	2	0
6	0.56	10	2	0	0.43	0.31	10	**1**	0	0.47	10	2	0
7	1.35	19	2	0	0.50	0.28	40	2	1	0.86	19	2	0
8	2.63	21	3	0	0.46	0.24	unsatisfiable			0.56	23	2	0
9	5.44	27	3	0	0.92	0.22	unsatisfiable			1.26	27	**2**	0
10	3.43	21	3	0	0.57	0.28	39	2	1	3.07	21	**2**	0
11	10.23	22	3	0	0.59	0.29	38	2	1	2.29	22	**2**	0
12	18.11	27	3	0	0.66	0.30	47	2	1	4.30	27	**2**	0
13	1.29	14	3	0	0.32	0.28	17	1	0	0.72	15	2	0
14	0.47	17	2	0	0.40	0.44	33	2	1	0.34	17	2	0
15	0.17	6	2	0	0.20	0.45	28	2	1	0.26	6	2	0
16	1.61	15	2	0	0.31	0.29	15	**1**	0	1.10	15	2	0
17	10.72	23	5	0	0.66	0.24	unsatisfiable			0.24	unsatisfiable		
18	2.93	20	3	0	0.54	0.23	unsatisfiable			1.09	20	**2**	0
19	0.43	16	2	0	0.37	0.25	39	2	1	0.41	16	2	0
20	3.71	15	2	0	0.59	0.49	15	**1**	0	3.58	15	2	0
21	1.93	14	2	0	0.47	0.25	20	1	0	0.61	14	2	0
22	0.74	15	2	0	0.43	0.31	17	1	0	0.55	15	2	0
23	2.18	20	1	0	0.63	0.28	20	1	0	2.33	20	1	0
24	0.22	7	2	0	0.30	0.32	9	1	0	0.30	7	2	0
25	3.03	13	2	0	0.33	0.52	14	1	0	1.58	13	2	0
26	0.23	5	1	0	0.21	0.38	5	1	0	0.35	5	1	0
27	1.09	17	2	0	0.43	0.25	21	1	0	1.48	17	2	0
28	0.19	10	2	0	0.28	0.26	11	1	0	0.34	10	2	0
T.	1.48	451			0.44	0.28	209			0.86	3		

Also in Table 7.10, we show the results obtained by `WSimply` of these instances with homogeneity factor `F` = 1 (second block of columns) and `F` = 2 (third block of columns). The homogeneity factor bounds the difference in the violation of prerequisite constraints between courses (1 and 2 in our experiments). For homogeneity with factor 1, there are 5 unsolvable instances and 9 instances that achieve homogeneity by increasing the minimum number of violations per course (from 0 to 1) with, in addition, a dramatic increase in the total number of violations (+209). Experiments with homogeneity factor 2 give different results in 9 instances, all of which, except one becoming unsatisfiable, and are effectively improved by reducing the maximum number of violations per course, and slightly increasing the total number of violations (+3). Interestingly, the solving time has been improved when adding homogeneity.

By way of guidance a comparison between `WSimply` and CPLEX has been done only over the basic SBACP instances since CPLEX does not have any meta-constraint. `WSimply` exhibits a reasonably good performance in taking 1.48 seconds in median against the 0.44

seconds of CPLEX.

Table 7.11: Results when adding the `multiLevel` meta-constraint to deal with the optimisation version of the SBACP, minimising the maximum workload per period and the maximum number of courses per period. Numbers in boldface denote improvements. Timeout is 600s. The last row shows the median of the solving time and the improvements on the aggregated maximums of the workload and number of courses per period thanks to the `multiLevel` meta-constraint.

	Homogeneity factor 2					MLevel: prereq, workload					MLevel: prereq, wl, courses				
N.		c./p.		wl/p.			c./p.		wl/p.			c./p.		wl/p.	
	Time	Min	Max	Min	Max	Time	Min	Max	Min	Max	Time	Min	Max	Min	Max
1	0.75	11	14	51	87	23.60	12	14	65	**66**	32.68	12	**13**	65	66
2	1.61	12	13	68	77	11.06	11	15	70	**71**	9.33	12	**13**	70	71
3	1.70	12	14	62	72	23.59	11	15	67	**68**	17.00	11	**14**	66	68
4	3.69	11	17	52	112	16.09	11	15	74	**77**	5.29	12	**13**	72	77
5	0.86	11	15	41	81	6.20	9	16	59	**63**	8.67	9	**15**	58	63
6	0.47	11	15	49	86	2.35	12	13	59	**60**	2.93	12	13	59	60
7	0.86	6	17	32	104	9.73	11	14	65	**66**	59.32	10	14	65	66
8	0.56	9	16	43	86	1.92	10	16	58	**66**	2.66	12	**13**	61	66
9	1.26	8	16	39	109	15.28	12	13	74	**77**	5.00	12	13	74	77
10	3.07	8	15	33	79	26.00	11	16	58	**66**	12.19	12	**14**	57	66
11	2.29	10	15	46	88	12.26	11	14	68	**68**	45.67	12	**13**	68	68
12	4.30	6	16	37	100	97.92	10	17	69	**70**	195.52	10	17	69	70
13	0.72	6	18	44	98	10.16	10	15	71	**72**	16.06	11	**13**	71	72
14	0.34	9	18	40	106	1.18	10	16	65	**69**	1.79	10	16	65	69
15	0.26	5	16	46	92	13.05	11	16	72	**72**	46.65	11	**13**	72	72
16	1.10	9	17	50	79	61.86	11	14	61	**63**	108.07	12	**13**	61	63
17	0.24	unsatisfiable				0.59	unsatisfiable				0.67	unsatisfiable			
18	1.09	10	15	62	92	15.25	12	13	74	**75**	19.96	12	13	74	75
19	0.41	8	19	51	99	8.65	11	14	67	**68**	15.94	11	**13**	67	68
20	3.58	10	16	54	112	53.86	10	14	70	**75**	38.80	10	14	70	75
21	0.61	9	18	42	94	2.27	11	14	65	**65**	2.51	12	**13**	65	65
22	0.55	10	16	57	105	2.18	11	14	75	**77**	1.76	12	**13**	75	77
23	2.33	20	25	25	128	103.47	11	15	67	**68**	173.60	11	**13**	67	68
24	0.30	12	14	57	81	0.71	12	13	63	**78**	2.65	12	13	64	78
25	1.58	9	16	44	87	31.86	12	14	70	**70**	26.15	12	**13**	70	70
26	0.35	5	18	24	102	9.28	11	14	51	**78**	8.01	10	14	61	78
27	1.48	10	15	57	96	8.19	11	15	81	**81**	23.60	11	**14**	81	81
28	0.34	9	15	42	101	2.30	11	15	69	**70**	4.28	11	**14**	68	70
T.	0.86		439		2553	10.61				-654	14.07		-27		

The first block of columns of Table 7.11 shows the minimum and maximum number of courses per period and the minimum and maximum workload per period, for each considered instance, when asking for homogeneity with factor 2 on the number of prerequisite violations.[6] As we can see, the obtained curricula are not balanced enough with respect to the number of courses per period and the workload per period. Therefore, we propose considering the optimisation version of SBACP by extending the initial modelling and using the `multiLevel` meta-constraint in order to improve the balancing in the workload and the number of courses per period (let us recall that the homogeneity meta-constraint on the number of violations introduced so far is hard). In Figure 7.11 we show how we have implemented this extension for the workload (for the number of courses it can be done analogously). We also must set the weights of the prerequisite soft constraints to undefined to let the `multiLevel` meta-constraint compute them.

[6] We have chosen homogeneity factor 2 to continue with the experiments since, with factor 1, the number of violations increases in almost all instances.

```
    int load_distance = load_per_period_ub - load_per_period_lb;
%%% load_bound is the upper bound of the load of all periods.
    IntVar load_bound::dload;
    Forall(p in [1..n_periods]) {
      period_load[p] =< load_bound;
    };
%%% We post as soft constraints each unit of distance between
%%%  load_bound and load_per_period_lb.
    Forall(d in [1..load_distance]) {
      #bala[d]: (load_bound < (load_per_period_lb + d)) @ {_};
    };
%%% We compose the new objective function with these two components,
%%%  being more important the prerequisites than the minimization of the
%%%  maximum load per period.
    multiLevel([[pre[np] | np in [1..n_prereqs]],
                [bala[d] | d in [1..load_distance]]]);
%%% Undefined weight of prerequisites soft constraints must be the same
    samePriority([pre[np] | np in [1..n_prereqs]]);
%%% Undefined weight of load distance soft constraints must be the same
    samePriority([bala[d] | d in [1..load_distance]]);
```

Figure 7.11: Extension to minimise the maximum workload (amount of credits) of periods.

The idea of this encoding is to minimise the maximum workload per period using soft constraints. Column eleven (wl/p. Max) shows the improvement in the maximum workload per period obtained when introducing its minimisation with the `multiLevel` meta-constraint. Column fourteen (c./p. Max) shows the improvement in the the maximum number of courses per period obtained when adding its minimisation as the next level in the `multiLevel` meta-constraint.

7.7.3 `WSimply` (PB)

Next we present the experimental results of `WSimply` with PB solvers, presented at [BEPV12]. In this experiment we have used a PB model for the the NRP and SBACP presented in the previous section.

The solvers used in the experiments, classified by families, are the following.

- Integer Programming based.

  SCIP (scip-2.1.1.linux.x86) - A mixed integer programming solver.

- Pseudo-Boolean resolution based.

  Sat4j (sat4j-pb-v20110329) - A Boolean satisfaction and optimization library for Java. With PBC, it uses a cutting planes approach or pseudo-Boolean resolution [Sat12].

  Clasp (clasp-2.0.6-st-x86-linux) - A conflict-driven answer set solver.

  Bsolo (beta version 3.2 - 23/05/2010) - A solver with cardinality constraint learning.

- Translation to MaxSAT or SMT based.

  PWBO (pwbo2.0) - Also known as the parallel wbo solver. Since the other considered solvers are single-threaded, we have not used its parallel feature. When pbwo is run with one thread it uses a unsat-based core algorithm [MML11].

PB/CT (pbct0.1.2) - Translates the problem into SAT modulo the theory of costs and uses the Open-SMT solver.

SBACP

In Figure 7.12, we propose a modelling of the SBACP using `WSimply` to be solved using PB. Note that in this model we only have integer variables with pseudo-Boolean domain (`pseudo`).

```
Problem:sbacp
  Data
    int n_courses; int n_periods; int n_prerequisites;
    int load_per_period_lb;       int load_per_period_ub;
    int courses_per_period_lb;    int courses_per_period_ub;
    int loads[n_courses];         int prerequisites[n_prerequisites, 2];

  Domains
    Dom pseudo = [0..1];

  Variables
    IntVar x[n_periods, n_courses]::pseudo;
    IntVar x_aux[n_periods, n_courses]::pseudo;

  Constraints
    % each course is done only once
    Forall(c in [1..n_courses]) { Sum([x[p,c] | p in [1..n_periods]], 1); };

    Forall(p in [1..n_periods]) {
    % for each period the number of courses is within bounds
      AtLeast([x[p,c] | c in [1..n_courses]], courses_per_period_lb);
      AtMost ([x[p,c] | c in [1..n_courses]], courses_per_period_ub);
    % for each period the course load is within bounds
      AtLeast([ loads[c] * x[p,c] | c in [1..n_courses]], load_per_period_lb);
      AtMost ([ loads[c] * x[p,c] | c in [1..n_courses]], load_per_period_ub); };

    % Prerequisites and Corequisites
    % if course c is done previously to period p then x_aux[p,c] = 1
    Forall(c in [1..n_courses]) {
      Forall(p in [1..n_periods]) {
        Sum([x[p2,c] | p2 in [1..p-1]], x_aux[p,c]); }; };

    % a course being a (soft) prerequisite of another cannot be done after the other
    Forall(i in [1..n_prerequisites]) {
      Forall(p in [1..n_periods-1]) {
        x_aux[p+1, prerequisites[i,1]] >= x[p, prerequisites[i,2]]; }; };

    % prerequisite violation has cost 1
    Forall(i in [1..n_prerequisites]) {
      Forall(p in [1..n_periods]) {
        (x_aux[p, prerequisites[i,1]] >= x[p, prerequisites[i,2]])@ {1}; };
    };
```

Figure 7.12: `WSimply` (PB) model for the SBACP.

In Table 7.12, we give a small example to illustrate the meaning of the two matrices of variables `x` and `x_aux` used in the model. We consider an instance with three courses and three periods. Matrix `x` shows in what period a course is made. Matrix `x_aux` shows when a course is already done, that is, it is filled with ones for the periods following the one in which the course is done; this also can be seen as the accumulated version of matrix `x`. This duality lets expressing prerequisites easily. Note that the first period row in matrix `x_aux` always

Table 7.12: SBACP prerequisites example for the PB encoding.

x	c_1	c_2	c_3
p_1	1	0	0
p_2	0	1	0
p_3	0	0	1

x_aux	c_1	c_2	c_3
p_1	0	0	0
p_2	1	0	0
p_3	1	1	0

contains zeros.

Since in the `WSimply` PB version we cannot use the `homogeneousAbsoluteNumber` meta-constraint, in Figure 7.13, we show how we can simulate it with the `maxCost` meta-constraint, restricting the violations of prerequisites for each course to a number `n`. Note that, previously, we must associate the label `pre[i]` to the prerequisite constraints.

```
...
% prerequisite violation has cost 1
Forall(i in [1..n_prerequisites]) {
  #pre[i]: Forall(p in [1..n_periods]) {
           (x_aux[p, prerequisites[i,1]] >= x[p, prerequisites[i,2]])@ {1}; }; };

MetaConstraints
  maxCost([ [ pre[np] | np in [1..n_prerequisites], prerequisites[np,2] = c ]
   | c in [1..n_courses] ], n);
```

Figure 7.13: `maxCost` meta-constraint for the SBACP problem.

In the Table 7.13, we show the performance results for the SBACP. We have tested the pseudo-Boolean version of `WSimply` with the different back-ends described we have just described, along with the SMT version of `WSimply` using Yices 1 API, which is able to solve weighted SMT instances. The first group of rows reflect the execution times without the `maxCost` meta-constraint, while the second and third groups show the execution times with the meta-constraints `maxCost([...],1)` and `maxCost([...],2)`, respectively. Although not shown in the table, it is worth noting that in the case of using the constraint `maxCost([...],1)`, the problem becomes quite more constrained, resulting into 14 unsatisfiable instances out of the 28 instances. However, 9 of the 14 solutions of the remaining instances are improved in terms of fairness, since we have reduced the difference between the less violated course and the most violated course. Despite the fairness improvement, the mean cost of the solutions is penalized with an increment of 2.56 points. In the case of `maxCost([...],2)`, the problem becomes only slightly more constrained, transforming only one of the 28 instances in unsatisfiable. In this case, the number of improved instances, in terms of fairness, is only 2, with an increment in the mean cost of 1 point in the first case and of 2 points in the second case.

NRP

In Figure 7.14, we propose a modelling of the NRP using `WSimply` to be solved using PB. Note that in this model we only have integer variables with pseudo-Boolean domain (`pseudo`).

In Table 7.14, we show the summarized results for all the 7290 instances of the N25 set of the NSP. The solvers shown are the pseudo-Boolean version of `WSimply` with pwbo2.0 and SCIP, along with the SMT version of `WSimply` with Yices 1 API, as the rest of solvers have not been able to solve any instance within the time limit of 60 seconds. In these executions,

Table 7.13: SBACP solving times with a timeout of 60s. *: for now, meta-constrains are only available for the pbo format.

		WSimply (PB)							WSimply (SMT)
		pwbo		Sat4j	clasp	bsolo	PB/CT	SCIP	Yices
		PBO	WBO						
without	Total	31.49	48.19	44.48	23.12	70.86	53.80	33.25	125.96
	Mean	1.50	2.41	1.59	0.83	2.53	2.15	1.19	4.50
	Median	0.05	0.04	0.99	0.32	0.91	1.14	1.10	1.61
	# t.o.	7	8	0	0	0	3	0	0
maxC 1	Total	5.37	*	14.37	0.64	0.85	8.00	6.60	8.29
	Mean	0.19	*	0.51	0.02	0.03	0.29	0.24	0.30
	Median	0.04	*	0.50	0.02	0.02	0.23	0.16	0.28
	# t.o.	0	*	0	0	0	0	0	0
maxC 2	Total	88.84	*	35.05	9.21	19.27	56.72	28.34	47.34
	Mean	6.35	*	1.25	0.33	0.69	2.03	1.01	1.69
	Median	0.07	*	0.98	0.29	0.35	1.39	0.72	1.24
	# t.o.	14	*	0	0	0	0	0	0

```
Problem:nsp
  Data
    int n_nurses; int n_days; int n_shift_types; int min_turns; int max_turns;
    int covers[n_days, n_shift_types]; int prefs[n_nurses, n_days, n_shift_types];

  Domains
    Dom pseudo = [0..1];

  Variables
    IntVar x[n_nurses, n_days, n_shift_types]::pseudo;

  Constraints
    % each nurse can only work one shift per day
    Forall(n in [1..n_nurses], d in [1..n_days]) {
      Sum([x[n,d,s] | s in [1..n_shift_types]], 1); };

    % minimum covers
    Forall(d in [1..n_days], s in [1..n_shift_types-1]) {
      AtLeast([x[n,d,s] | n in [1..n_nurses]], covers[d,s]); };

    % the number of free days is within bounds
    Forall(n in [1..n_nurses]) {
      If (min_turns = max_turns) Then {
        Sum([x[n,d,n_shift_types] | d in [1..n_days]], n_days - min_turns);
      } Else {
        AtLeast([x[n,d,n_shift_types] | d in [1..n_days]], n_days - max_turns);
        AtMost ([x[n,d,n_shift_types] | d in [1..n_days]], n_days - min_turns); }; };

    % penalize each failed preference
    Forall(n in [1..n_nurses], d in [1..n_days], s in [1..n_shift_types]) {
      (x[n,d,s] = 0) @ {prefs[n,d,s]}; };
```

Figure 7.14: WSimply (PB) model for the NSP.

SCIP excels, being able to solve all instances with a mean time which is an order of magnitude lower than the one of the second fastest solver.

Table 7.14: Results on the NRP with soft constraints on nurse preferences.

	WSimply (PB)			WSimply (SMT)
	pwbo		SCIP	Yices
	PBO	WBO		
# solved	5095	5064	7290	4088
# t.o.	2195	2226	0	3202
Mean	1.63	1.77	0.10	4.49
Median	0.38	0.68	0.09	1.52

7.8 Conclusions

In this chapter we have introduced a new framework, called `WSimply`, for solving and modelling CSPs and WCSPs with its own declarative high-level language. Though it was originally thought to reformulate CSP instances into SMT instances, `WSimply` can also generate files in the PB and LP formats.

`WSimply` language is the first high-level CSP language that is directly reformulated into SMT, avoiding for instance the translation of Boolean constraints into arithmetic constraints like it happens when translating from MiniZinc to FlatZinc. In addition, soft constraints defined in `WSimply` are intensional, which reduces the work needed to model WCSPs. Furthermore, it is the first language that allows the user to define meta-constraints over soft constraints.

By default `WSimply` uses Yices 1 API as SMT solving engine, which allowed us to implement optimization and WSMT algorithms where the solver may keep some of the learned clauses through the successive calls to the SMT engine (i.e., incremental algorithms). However, `WSimply` can also generate SMT files in the "standard" SMT-Lib formats (v1.2 and v2), and perform optimization with them, generate PB files in the PBO and WBO formats and generate LP files.

In the experimental results section, we first see that `WSimply` is competitive with several FlatZinc back-end solvers. It worth to remark the comparison of `WSimply` with `fzn2smt`, where the latter solves only one instance more than the former. Nevertheless, `WSimply` has room for improvement since it does not apply any preprocessing technique during the reformulation from the high-level language into SMT.

Then, we show the usefulness of modelling WCSPs using intensional soft constraints and meta-constraints on the two classical CSPs: NRP and BACP. In the first one, we use homogeneity meta-constraints, the `homogeneousAbsoluteNumber` when using SMT and the `maxCost` when using PB, to ensure fairness on the violations of nurse shift preferences. While in the second one, we use the multilevel meta-constraint to define a priorities between the soft constraints of the problem.

In the PB experimental results, with respect to SBACP, we see that the performance of `WSimply` (PB) is slightly better than `WSimply` (SMT) with `dico`. Whereas, with respect to NRP, the performance of `WSimply` (PB) varies a lot depending on the PB solver used. The best solving approach for the NRP clearly is `WSimply` (PB) with the SCIP solver (which solves all the instances), followed by `WSimply` (PB) with the pwbo solver (which solves 69.89% of the instances) and `WSimply` (SMT) with `core` (which solves 56.07% of the instances). The other PB solvers cannot solve any instance.

Finally, we want to comment that we have also proposed a similar extension to deal with WCSPs for MiniZinc in [ABP+11b].

Chapter 8

BDD-based Incremental Optimization

In this chapter, we propose a method for solving WCSPs by translating them into a COP and performing iterative calls to an SMT solver, with successively tighter bounds of the objective function. The novelty of the method herewith described lies in representing the constraints bounding the objective function as a shared Binary Decision Diagram (SBDD), which in turn is translated into SAT. This offers two benefits: first, BDDs built for previous bounds can be used to build the BDDs for new (tighter) bounds, considerably reducing the BDD construction time; second, as a by-product, many clauses asserted to the solver in previous iterations can be reused. We also present a special case of objective function where we can use a variant of Multivalued Decision Diagrams (MDD) instead of BDDs.

The reported experimentation on the `WSimply` system shows that the use of SBDDs has better performance in general than other methods implemented in the system. Moreover, with this new technique `WSimply` outperforms some WCSP and CSP state-of-the-art solvers in most of the studied instances. As a particular case of study, we present results of applying the MDD version of the new technique on the nurse rostering problem (NRP).

This chapter is based on results from [BPSV14] and it is organised as follows. In Section 8.1 we introduce Reduced Ordered Binary Decision Diagrams (ROBDD) and Reduced Ordered Multivalued Decision Diagrams (ROMDD). In Section 8.2 we introduce the pseudo-Boolean constraints reformulation into SAT using BDDs, and the extension of this reformulation process using MDDs to deal with linear integer constraints. In Section 8.3 we present our incremental method of solving WCSPs, first by using shared ROBDDs and then the special case were we can use shared ROMDDs. In Section 8.4 we study the performance of the new method by comparing it with other methods implemented in `WSimply` and other state-of-the-art systems In the same section, we also report on some experiments showing how the use of BDDs to represent the objective function, instead of using a linear equation, has a positive impact in the learning of the SMT solver. Finally, conclusions are given in Section 8.5.

8.1 Binary Decision Diagrams (BDD)

Decision Diagrams are data structures usually used to represent Boolean functions. In this section we will first describe what a Binary Decision Diagram is, which is the data structure

most typically used for Boolean functions, and then we will describe what is a Multivalued Decision Diagram.

A *Binary Decision Diagram (BDD)* consists of a rooted, directed, acyclic graph, where each non-terminal (decision) node has two child nodes. When are used to represent Boolean functions, each non-terminal node corresponds to a Boolean variable x and the edges to the child nodes are representing a *true* and a *false* assignment to x, respectively. We talk about the *true child* (resp. *false child*) to refer to the child node linked by the *true* (resp. *false*) edge. Terminal nodes are called 0-terminal and 1-terminal, representing the truth value of the formula for the assignment leading to them. A BDD is called *ordered* if different variables appear in the same order on all paths from the root. A BDD is said to be *reduced* if the following two rules have been applied to its graph until fixpoint:

- Merge any isomorphic subgraphs.
- Eliminate any node whose two children are isomorphic.

A *Reduced Ordered Binary Decision Diagram (ROBDD)* is canonical (unique) for a particular function and variable order. Figure 8.1 shows an example of a ROBDD.

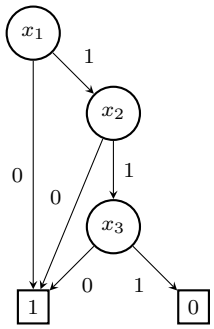

Figure 8.1: ROBDD for the Boolean function $\overline{x_1} + x_1\overline{x_2} + x_1x_2\overline{x_3}$.

An interesting property of ordered BDDs is that when multiple ordered BDDs contain isomorphic subgraphs, they can be joined in a single *shared BDD (SBDD)* [iMIY90], providing a more compact representation of the Boolean functions. Figure 8.2 shows a shared ROBDD constructed from the union of four ROBDDs.

A *Multivalued Decision Diagram (MDD)* is a generalization of a BDD where each node can have more than two outgoing edges, i.e., consists of a rooted, directed, acyclic graph, where each non-terminal (decision) node has multiple child nodes. We will see in the next section that BDDs are also used to represent pseudo-Boolean constraints, and hence we can use a MDD to represent linear integer constraints.

As well as in the case of BDDs, we can also talk about reduced ordered MDDs (ROMDD), which maintain the same properties of the ROBDDs.

8.2 BDD-based SAT encodings of PB constraints

There exist several BDD-based approaches for reformulating PB constraints into propositional clauses [ES06]. In this section we focus on the recent work of [ANO$^+$12], that proposes a

simple and efficient algorithm to construct ROBDDs for monotonic Boolean functions, and a corresponding generalized arc-consistent SAT encoding. Then we present a generalisation of the method to represent linear integer constraints using ROMDDs.

8.2.1 SAT encodings of PB constraints using BDDs

Recall from Section 2.3.1 that pseudo-Boolean (PB) constraints have the form $a_1x_1 + \cdots + a_nx_n \ \# \ K$, where the a_i and K are integer coefficients, the x_i are pseudo-Boolean (0/1) variables, and the relation operator $\#$ belongs to $\{<, >, \leq, \geq, =\}$. For our purposes we will assume that $\#$ is $\leq$ and that the a_i and K are positive. Under these assumptions, these constraints are monotonic (decreasing) Boolean functions $C : \{0,1\}^n \rightarrow \{0,1\}$, i.e., any solution for C remains a solution after flipping input values from 1 to 0.

It is quite common to use BDDs to represent PB-constraints. For example, the ROBDD of Figure 8.1 also corresponds to the PB constraint $2x_1 + 3x_2 + 4x_3 \leq 7$.

An interval of a PB constraint C is the set of values of K for which C has identical solutions. More formally, given a constraint C of the form $a_1x_1 + \cdots + a_nx_n \ \leq \ K$, the *interval of* C is the set of all integers M such that the constraint $a_1x_1 + \cdots + a_nx_n \ \leq \ M$, seen as a Boolean function, is equivalent to C (i.e., that the corresponding Boolean functions have the same truth table). For instance, the interval of $2x_1 + 3x_2 + 4x_3 \leq 7$ is $[7, 8]$ since, as no combination of coefficients adds to 8, we have that the constraint $2x_1 + 3x_2 + 4x_3 \leq 7$ is equivalent to $2x_1 + 3x_2 + 4x_3 \leq 8$.

PB constraint intervals are the key point of the algorithm proposed in [ANO+12]. It is a dynamic, bottom up BDD construction algorithm, which runs in polynomial time with respect to the ROBDD size and the number of variables. It keeps the intervals of the PB constraints built for the already visited nodes: for a given variable ordering, say $x_1, x_2, \ldots, x_n$, a list of *layers* $\mathcal{L} = L_1, \ldots, L_{n+1}$ is maintained, where each layer L_i is a set of pairs of the form $([\beta, \gamma], \mathcal{B})$, being $\mathcal{B}$ the ROBDD of the constraint $a_ix_i + \cdots + a_nx_n \ \leq \ K$, for every K in the interval $[\beta, \gamma]$.

These intervals are used to detect if some needed ROBDD has already been constructed. That is, if for some node at level i, the ROBDD for the constraint $a_ix_i + \cdots + a_nx_n \ \leq \ K$ is needed for a given K, and K belongs to some interval already computed in layer L_i, then the same ROBDD can be used for this node. Otherwise, a new ROBDD is constructed and a new pair (the ROBDD and its respective interval) is added to the layer. It is important to recall here that ROBDDs are unique for a given function and variable ordering. The construction of the new ROBDD for some node at level i is very simple. This node will have two children, therefore we recursively call the ROBDD construction method two times, one for the true child with PB constraint $a_{i+1}x_{i+1} + \cdots + a_nx_n \ \leq \ K - a_i$, and one for the false child with PB constraint $a_{i+1}x_{i+1} + \cdots + a_nx_n \ \leq \ K$. For each child, the recursive call will return a pair $([\beta_{i+1}, \gamma_{i+1}], \mathcal{B}_{i+1})$, which will be used to create the pair $([\beta_i, \gamma_i], \mathcal{B}_i)$ to be asserted to the current layer, i.e., compute the interval and create the ROBDD of the current node from the pairs of the child nodes. In order to have a reduced BDD we only have to check if the two child nodes have isomorphic sub-graphs, which means both children are falling in the same interval; in that case, we don't have to create any node and instead we use $\mathcal{B}_{i+1}$.

Once the ROBDD is constructed, we use the SAT encoding proposed in [ANO+12], which is generalized arc-consistent, and works as follows. For each node with a selector variable

x we create, a new auxiliary variable n, which represents the state[1] of the node, and two clauses:

$$\bar{f} \rightarrow \bar{n} \qquad \bar{t} \wedge x \rightarrow \bar{n}$$

being f the state variable of its false child and t the state variable of its true child. Finally, we add a unit clause with the state variable of the 1-terminal node, another clause with the negation of the variable of the 0-terminal node. To make the encoding generalized arc-consistent, a unit clause forcing the state variable of the root node to be *true* must be added.

We refer the reader to [ANO+12] for additional details on the BDD construction algorithm and the SAT encoding.

8.2.2 SAT encodings of linear integer constraints using MDDs

Based on the work of [ANO+12], next we present a SAT encoding for linear integer (LI) constraints, by first constructing a ROMDD to represent the LI constraint and then translate the ROMDD into SAT clauses.

The ROMDD construction method is almost the same as the previous one: for a given variable ordering, say $x_1, x_2, \ldots, x_n$, we keep a list of *layers* $\mathcal{L} = L_1, \ldots, L_{n+1}$, where each layer L_i is a set of pairs of the form $([\beta, \gamma], \mathcal{M})$, being $\mathcal{M}$ the ROMDD of the constraint $a_i x_i + \cdots + a_n x_n \leq K$, for every K in the interval $[\beta, \gamma]$; then at some level i, for the LI constraint $a_i x_i + \cdots + a_n x_n \leq K$, we first check if there exists a ROMDD at layer L_i for the current K, otherwise we have to create the new ROMDD, compute its interval and insert the pair into L_i. The two differences from the ROBDD construction method is that now, probably, we will recursively call the construction method more than two times and, in addition, the LI constraint to construct each child node will be $a_{i+1} x_{i+1} + \cdots + a_n x_n \leq K - (a_i \times v_j)$ where v_j is the j value in variable x_i domain, $v_1 < v_2 < \cdots < v_m$ and m is the number of values in the domain of x_i. Finally, in order to have a reduced MDD, we only have to check if all the child nodes are the same.

Once we have constructed the ROMDD, the SAT encoding is very similar to the one we have just described for BDDs, the difference is that for MDDs, in the worst case, we will have as many clauses as child nodes instead of only two:

$$\bar{c_1} \rightarrow \bar{n} \qquad \bar{c_2} \wedge (x = v_2) \rightarrow \bar{n} \qquad \ldots \qquad \bar{c_m} \wedge (x = v_m) \rightarrow \bar{n}$$

where c_i is the state variable of the i-th child, v_i is the i-th value of variable x, $v_1 < v_2 < \cdots < v_n$, m is the number of children and n is the state variable of the current node of the ROMDD. Note that the first equation is the same as in the BDDs, because every variable has to take a value and if the state variable of the first child is *false* then the state variable of the current node also has to be *false*.

When a node has two or more isomorphic child nodes we can compact the clauses of them into one of the form:

$$\bar{c_i} \wedge (x \geq v_i \wedge x \leq v_j) \rightarrow \bar{n}$$

where c_i is the state variable of the i-th child, $i < j$, v_i and v_j are the i-th and j-th values of variable x and n is the state variable of the current node of the ROMDD. Note that this is because $c_i = c_j$.

[1] That is, if the PB constraint corresponding to (the interval of) the node is satisfied or not.

Finally, we present a variant of the ROMDD construction method for the following special case: given a variable ordering, say $x_1, x_2, \ldots, x_n$, we want to construct a ROMDD to represent $f_1(x_1) + f_2(x_2) + \cdots + f_n(x_n) \leq K$, where x_i are integer variables and f_i is a function that returns an integer value (weight) depending on the value of variable x_i. For now and so on we will call this method weighted ROMDD (WROMDD) or weighted MDD (WMDD) for the sake of simplicity.

The ROMDD construction method is very similar to the previous one, but in this case when we have to create a new ROMDD node for $f_i(x_i) + \cdots + f_n(x_n) \leq K$ at some level i, the recursive call for the child nodes will use $f_{i+1}(x_{i+1}) + \cdots + f_n(x_n) \leq K - f_i(v_j)$, where v_j is the j value in x_i domain.

Once we have constructed the ROMDD we can generate the same SAT clauses:

$$\bar{c_1} \rightarrow \bar{n} \qquad \bar{c_2} \wedge (x = v_2) \rightarrow \bar{n} \qquad \ldots \qquad \bar{c_m} \wedge (x = v_m) \rightarrow \bar{n}$$

where c_i is the state variable of the i-th child, v_i is the i-th value of variable x, $v_1 < v_2 < \cdots < v_n$, m is the number of children and n is the state variable of the current node of the ROMDD.

What is different from the previous case is that, in general, when we have two or more isomorphic child nodes we cannot use the $\leq$ and $\geq$ notation and we have to use the following clause:

$$\bar{c_i} \wedge (x = v_i \vee \cdots \vee x = v_j) \rightarrow \bar{n}$$

where c_i is the state variable of the i-th child, $v_i, \ldots, v_j$ are all the values of variable x that has isomorphic sub-graphs and n is the state variable of the current node of the ROMDD.

8.3 Solving WCSPs by Incremental Optimization using Shared ROBDDs

The WCSP solving method presented here consists in reformulating the WCSP into a COP, and solving the resulting optimization problem by iteratively calling an SMT solver with the problem instance, together with successively tighter bounds for the objective function.

The novelty of the method lies in the way the objective function is treated. Inspired by the idea of intervals in the BDD construction algorithm of [ANO+12], our aim is to represent the objective function resulting from the WCSP (see Subsection 7.4.3) as a pseudo-Boolean constraint, then represent it as a BDD and take profit of BDD reuse in successive iterations of the optimization process. That is, instead of creating a (reduced ordered) BDD from scratch at every iteration, we build a shared BDD. Since the PB constraint encoded at each iteration is almost the same, with only the bound constant K changing, this will hopefully lead to high node reuse.

We claim that using shared BDDs has two important benefits. The first one, fairly evident, is that node reuse considerably reduces the BDD construction time. The second one, which is not so evident, is that, as a by-product, we will be reusing many literals and clauses resulting from the SAT encoding of BDDs from previous iterations (in addition to clauses learned at previous steps). In Section 8.4 we present some experiments to support these claims.

We want to remark that this incremental optimization algorithm can be also used for linear integer objective functions but using shared ROMDDs instead of shared ROBDDs. We have some experimental results for the variant with WMDDs described in Section 8.4.4.

8.3.1 Incremental optimization algorithm

Algorithm 6 describes our WCSP solving method. The input of the algorithm is a WCSP instance divided into a set φ_s of soft constraints and a set φ_h of hard constraints, and its output is the optimal cost of $\varphi_s \cup \varphi_h$ if φ_h is satisfiable, and UNSAT otherwise.

Algorithm 6 Solving a WCSP by incremental optimization using a shared ROBDD

Input: $\varphi_s = \{(C_1, w_1), \ldots, (C_m, w_m)\}$, $\varphi_h = \{C_{m+1}, \ldots, C_{m+m'})\}$
Output: Optimal cost of $\varphi_s \cup \varphi_h$ or *UNSAT*

```
φ ← φ_h ∪ reif_soft(φ_s)
(st, M) ← SMT_algorithm(φ)
if st = UNSAT then
  return UNSAT
else
  ub ← sum({w_i | (C_i, w_i) ∈ φ_s and C_i is falsified in M})
end if
lb ← −1
L ← init_layers(φ_s)
while ub > lb + 1 do
  K ← ⌊(ub + lb)/2⌋
  ([β, γ], B) ← ROBDD(φ_s, K, L)
  (root, φ) ← BDD2SAT(B, φ)
  φ ← φ ∪ {root}
  (st, M) ← SMT_algorithm(φ)
  if st = UNSAT then
    lb ← γ
    φ ← (φ \ {root}) ∪ {¬root}
  else
    ub ← min(β, sum({w_i | (C_i, w_i) ∈ φ_s and C_i is falsified in M}))
  end if
end while
return ub
```

The first step is to reify the soft constraints (**reif_soft**), and create a formula φ with the reified soft constraints together with the hard constraints. By reifying a soft constraint C_i we mean adding a hard constraint $x_i \leftrightarrow \overline{C_i}$, where x_i is a new pseudo-Boolean variable. Note that in fact we are reifying the negation of C_i, since the cost w_i is associated to its violation (see Subsection 7.4.3).

Then, the satisfiability of φ is checked by an SMT solver. **SMT_algorithm** returns a tuple with the satisfiability (st) of φ and, if satisfiable, an assignment (model) M of φ. Note that if φ is unsatisfiable, this implies that φ_h is unsatisfiable, and hence we return UNSAT. Otherwise, we use the solution found to compute an upper bound ub of the objective function by aggregating the weights of the violated soft constraints, and set the lower bound lb to -1.

Before starting a binary search procedure to determine the optimal cost, we initialize $\mathcal{L}$ using **init_layers**(φ_s), i.e., we initialize each layer i of $\mathcal{L}$ (for i in $1..m+1$) with the two pairs of intervals and (trivial) ROBDDs $\{((-\infty,-1],\mathbf{0}),([\sum_{j=i}^{m} w_j,\infty),\mathbf{1})\}$, meaning that the PB constraint $w_i x_i + \cdots + w_m x_m \leq K$ is trivially false for $K \in (-\infty,-1]$ and trivially true for $K \in [\sum_{j=i}^{m} w_j,\infty)$, where $x_i,\ldots,x_m$ denote the reification variables of the soft constraints $C_i,\ldots,C_m$ as described above (see Subsection 8.2 for the definition of layer and interval).

In the first step of the `while` statement, we determine a new tentative bound K for the objective function. Then, we call the **ROBDD** construction algorithm of [ANO+12] (briefly described in Subsection 8.2) with the set of soft clauses φ_s, the new bound K and the list of layers $\mathcal{L}$, being this last an input/output parameter. This way, $\mathcal{L}$ will contain the shared ROBDD with all the computed ROBDDs, and may be used in the following iterations of the search, significantly reducing the construction time and avoiding the addition of repeated clauses. This procedure returns the ROBDD $\mathcal{B}$ representing the objective function for the specific K in the current iteration.

In the next step we call the **BDD2SAT** procedure, which generates the propositional clauses from $\mathcal{B}$, as explained in Subsection 8.2, but only for the new nodes. The procedure inserts these clauses into the formula φ, and the new formula is returned together with the auxiliary variable *root* associated to the root node of $\mathcal{B}$. This variable is inserted into φ as a unit clause to effectively force the objective function to be less or equal than K.

At this point we call the SMT solver to check the satisfiability of the new φ. We remark that, since we are using the SMT solver through its API, we only need to feed it with the new clauses. If φ is satisfiable we can keep all the learned clauses. Otherwise, we need to remove the unit clause for the root node. This way, we will (only) remove the learned clauses related to this unit clause. In addition, we add a unit clause with the negation of the root node variable, stating that the objective function value must be greater than K.

Finally, we update either the lower or upper bound according to the interval $[\beta,\gamma]$ of the ROBDD $\mathcal{B}$ and the computed assignment M for φ: if φ is unsatisfiable, then the lower bound is set to γ; otherwise, the upper bound is set to $min(\beta, sum(\{w_i \,|\, (C_i,w_i) \in \varphi_s \text{ and } C_i \text{ is falsified in } M\}))$. From the invariant that the lower bound always corresponds to an unsatisfiable case, while the upper bound corresponds to a satisfiable case, when $ub = lb+1$ this value corresponds to the optimum.

Note that, thanks to the intervals, in fact we are checking the satisfiability of the PB constraints for several values at the same time and hence, sometimes, this can allow us to obtain better lower/upper bound updates.

Example 20 *Figure 8.2 and Figure 8.3 show, respectively, the evolution of the shared ROBDD and the propositional clauses added to the SMT formula for the objective* $2x_1+3x_2+4x_3 \leq K$*, with successive values* $K=7$*,* $K=3$*,* $K=5$ *and* $K=4$*.*

8.4 Experimental results

In this section we first compare the performance of the presented solving method with that of other methods implemented in `WSimply`. Second, we compare the performance of `WSimply`, using this new method, with that of several state-of-the-art CSP, WCSP and ILP solvers. Third, we study the benefits, such as learning, obtained from using a shared BDD for the objective instead of a linear equation, and the amount of BDD node reutilization between successive iterations in the optimization process. Finally, we apply the new WMDD method

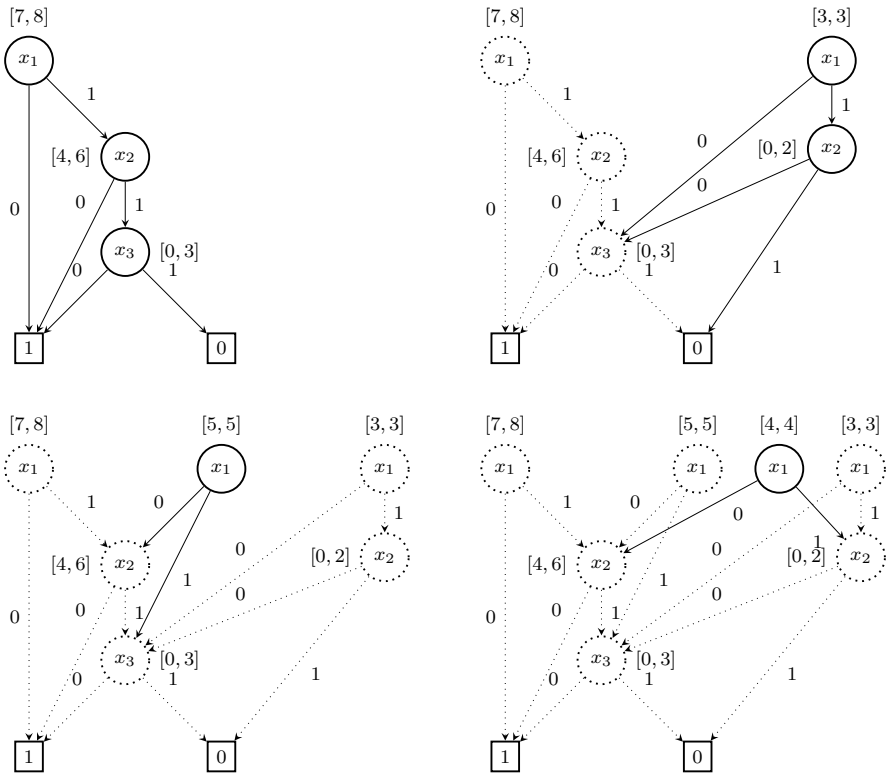

Figure 8.2: Shared ROBDDs and intervals for objective $2x_1 + 3x_2 + 4x_3 \leq K$, with successive values $K = 7$ (top left), $K = 3$ (top right), $K = 5$ (bottom left) and $K = 4$ (bottom right), illustrating the reuse of previous ROBDDs.

$$
\begin{aligned}
K = 7: \quad & \varphi_1 = \varphi \cup \{\overline{t} \rightarrow \overline{n_{1,1}},\ \overline{n_{2,1}} \wedge x_1 \rightarrow \overline{n_{1,1}},\ \overline{t} \rightarrow \overline{n_{2,1}},\ \overline{n_{3,1}} \wedge x_2 \rightarrow \overline{n_{2,1}},\ \overline{t} \rightarrow \overline{n_{3,1}},\\
& \qquad \overline{f} \wedge x_3 \rightarrow \overline{n_{3,1}},\ t,\ \overline{f}\} \cup \{n_{1,1}\}\\
K = 3: \quad & \varphi_2 = \varphi_1 \cup \{\overline{n_{3,1}} \rightarrow \overline{n_{1,2}},\ \overline{n_{2,2}} \wedge x_1 \rightarrow \overline{n_{1,2}},\ \overline{n_{3,1}} \rightarrow \overline{n_{2,2}},\ \overline{f} \wedge x_2 \rightarrow \overline{n_{2,2}}\} \cup \{n_{1,2}\}\\
K = 5: \quad & \varphi_3 = (\varphi_2 \setminus \{n_{1,2}\}) \cup \{\overline{n_{1,2}}\} \cup \{\overline{n_{2,1}} \rightarrow \overline{n_{1,3}},\ \overline{n_{3,1}} \wedge x_1 \rightarrow \overline{n_{1,3}}\} \cup \{n_{1,3}\}\\
& \qquad \text{(supposing that } \varphi_2 \text{ has been found to be unsatisfiable in iteration 2)}\\
K = 4: \quad & \varphi_4 = \varphi_3 \cup \{\overline{n_{2,1}} \rightarrow \overline{n_{1,4}},\ \overline{n_{2,2}} \wedge x_1 \rightarrow \overline{n_{1,4}}\} \cup \{n_{1,4}\}
\end{aligned}
$$

Figure 8.3: Formula update (clauses added and removed), for objective $2x_1 + 3x_2 + 4x_3 \leq K$ with successive values $K = 7$, $K = 3$, $K = 5$ and $K = 4$, according to the shared ROBDDs of Figure 8.2. φ_j denotes the SMT formula at hand when checking satisfiability in iteration number j of Algorithm 6. Each atom $n_{i,j}$ is the state variable of the node with selector variable x_i, that is added in iteration j, and t and f are the variables of the 1-terminal and 0-terminal nodes, respectively.

on the Nurse Rostering Problem (NRP) and we compare it with the current `WSimply` solving methods.

Table 8.1: Average number of variables, and weights, of the soft constraints in each pack of instances. Weights are denoted as $(min - max)$ for ranges and as $(val_1, val_2, \dots)$ for packs with a small number of distinct values in weights.

Problem	#Variables	Weights		Problem	#Variables	Weights
sbacp	67	1		*talent*	80	(4,5,10,20,40)
sbacp_h1	67	1				
sbacp_h2	67	1		*auction*	138	∼(100-1000)
sbacp_h2_ml2	312	(1,246)				
sbacp_h2_ml3	332	(1,21,5166)		*spot5*	385	(1-5,1000,2000)
s.l.	30	1		*celar*	1482	(1,10,100,1000)
s.l. free	30	1				(1,100,10000,100000)
s.l. no border	30	1				(1,2,3,4)

For the comparison we use (variants of) six problems:[2] five variants of the Soft BACP (SBACP) [ABP+13] (a softened version of the well-known Balanced Academic Curriculum Problem), three variants of the Still Life Problem and one variant of the Talent Scheduling Problem from the MiniZinc distribution[3] (all of them reformulated as a WCSP) and three classical WCSPs: CELAR, SPOT5 and Combinatorial Auctions.

Since the size of the generated BDDs strongly depends on the number of variables and the number of distinct coefficients in the objective function, we briefly describe these features for the studied problems in Table 8.1.

The experiments were run on a cluster of Intel® Xeon™CPU@3.1GHz machines, with 8GB of RAM, under 64-bit CentOS release 6.3 (kernel 2.6.32). `WSimply` was run on top of the Yices 1.0.33 [DdM06b] SMT solver. It is worth noting that by calling Yices through its API, we are able to keep learned clauses from previous calls that are still valid.

8.4.1 `WSimply` solving methods comparison

Table 8.2 shows the aggregated time (sum of the times to produce an optimal solution and prove its optimality, for all instances) for the solved instances. We consider the `yices`, `core` and `dico` solving methods described in Section 7.7, plus the new method `sbdd`$\leq$ using shared ROBDDs, where variables in the BDD are ordered from small (root) to big (leaves) coefficients. We also studied the performance of `sbdd`$\geq$, where variables are ordered from big (root) to small (leaves) coefficients, but the performance of `sbdd`$\leq$ was slightly better.

The new solving method with shared ROBDDs is clearly the best in the majority of problems. The performance of the `sbdd`$\leq$ method is better on instances with a small number of distinct coefficients (especially when all coefficients are 1), namely, in the *sbacp* and *still life* problems. In these cases, the BDDs are relatively small, having the bigger ones only thousands of nodes. On the other side, the timeouts occurring in the *celar* and *spot5* problems are mostly in the instances with higher number of variables in the objective function (e.g., from approximately 1200 to 5300 variables in *celar*), generating BDDs of hundreds of thousands nodes (a similar situation occurs in the *auction* instances running out of time, where there are not so many variables but a lot of distinct variable coefficients). In spite of this, `sbdd`$\leq$ is still the best method on the *celar* and *spot5* problems.

It is worth to notice that there are some *unknowns* in the `yices` method, due to the fact

[2] All instances available in `http://ima.udg.edu/Recerca/lap/simply`

[3] `http://www.minizinc.org`

Table 8.2: Aggregated time in seconds for the solved instances in each set, with a cutoff of 600s per instance. The column # indicates the number of instances per set. The indication (n^1) refers to the number of instances for which the solver ran out of time, (n^2) refers to the number of instances for which the solver returned *unknown*, and (n^3) refers to the number of instances for which the solver ran out of memory.

Problem	#	`dico`	`yices`	`core`	`sbdd≤`
sbacp	28	70.53	40.17	1342.16 (13^1)	**9.83**
sbacp_h1	28	**6.68**	12.02	363.78 (12^1)	6.93
sbacp_h2	28	26.04	26.65	748.81 (13^1)	**9.97**
sbacp_h2_ml2	28	280.68	109.42	75.82 (19^1)	**78.36**
sbacp_h2_ml3	28	804.74	516.51	547.55 (23^1)	**92.37**
s.l.	10	118.05 (1^1)	102.97 (1^2)	3.35 (4^1)	**191.56**
s.l. free	10	434.05 (2^1)	2.69 (3^2)	386.91 (3^1)	**98.75**
s.l. no border	10	187.94 (1^1)	166.91 (1^2)	81.24 (3^1)	**105.30**
talent	11	289.90	60.72	94.82 (8^1)	**38.25**
auction	170	9084,03 (93^1)	**1083.44 (85^2)**	0 (80^1 90^3)	12028.50 (78^1 11^3)
celar	16	582,64 (14^1)	750.32 (10^1 2^2)	94.82 (8^1 6^3)	**1417.19 (6^1)**
spot5	20	205,2 (18^1)	9.32 (5^1 13^2)	94.82 (14^1 3^3)	**1990.59 (7^1)**

that the Yices MaxSMT solver is non exact and incomplete. The `core` method has really bad performance on the crafted instances considered in this experiment, probably due to the bad quality of the unsatisfiable cores found during the solving process.

8.4.2 SBDD-based versus state-of-the-art CSP and WCSP solvers

For the sake of completeness we tested the performance of the MATHSAT5-MAX and LL_{WPM} MaxSMT solvers of [CGSS13b] on the weighted SMT instances generated by `WSimply` in the previous experiment. However we do not report these results, since they are comparable to those of the `WSimply core` method.

The second part of our experiments consists in reformulating the WCSPs into (MiniZinc) COPs, and compare the performance of `WSimply` using the `sbdd≤` method with that of IBM ILOG CPLEX 12.6 and some state-of-the-art CSP and WCSP solvers, namely Toulbar2 0.9.5.0, G12-CPX 1.6.0 and Opturion 1.0.2, all of them capable of solving MiniZinc instances. We used Numberjack [HOO10] as a platform for solving MiniZinc COPs through the Toulbar2 API. We also used the Toulbar2 binary to solve the original WCSP instances of *auction*, *celar* and *spot5*, indicated in Table 8.3 as *auction (wcsp)*, *celar (wcsp)* and *spot5 (wcsp)*, respectively. In order to test the performance of CPLEX on the considered problems, we used `WSimply` to translate the instances to pseudo-Boolean constraints. For the experiments with Opturion we used a slightly different computer (Intel® Core™CPU@2.8GHz, with 12GB of RAM, under 64-bit Ubuntu 12.04.3, kernel 3.2.0) due to some library dependence problems.

Table 8.3 shows the results of this second experiment. We can observe that, in general, `sbdd≤` is the most robust method, considering the number of instances solved and their aggregated solving time. The *sbacp* and *still life* problems seem to be reasonably well suited for SMT solvers (in particular, the latter consists of disjunctions of arithmetic constraints). We highlight the *talent scheduling* problem, where `sbdd≤` clearly outperforms the other solvers, being the only solver capable to solve all the instances. In fact, this is probably the best suited problem for SMT solvers, as it consists of binary clauses of *difference logic* constraints. Unfortunately, in most of the instances of this problem, Toulbar2 reported an error saying

Table 8.3: Aggregated time in seconds for the solved instances in each set, with a cutoff of 600s per instance. The column # indicates the number of instances per set. The indication (n^1) refers to the number of instances for which the solver ran out of time, (n^3) refers to the number of instances for which the solver ran out of memory, and (n^4) refers to the number of instances for which the solver returned another type of error.

Problem	#	**TB2**	**G12-CPX**	**Opturion**	`sbdd≤`	**CPLEX**
sbacp	28	662.58 (24^1)	76.31	83.47	**9.83**	36.81
sbacp_h1	28	707.78	**4.00**	13.25	6.93	21.78
sbacp_h2	28	483.98	19.53	30.51	**9.97**	24.89
sbacp h2_ml2	28	3849.38 (6^1)	166.69	91.55	78.36	**64.98**
sbacp h2_ml3	28	2408.66 (12^1)	336.25	99.77	92.37	**72.95**
s.l.	10	166.08 (2^1)	232.1 (1^1)	43.81 (2^1)	**191.56**	234.71
s.l. free	10	69.85 (3^1)	267.13 (2^1)	394.28 (2^1)	98.75	**29.91**
s.l. no border	10	122.05 (2^1)	354.41 (1^1)	102.29 (2^1)	105.3	**68.37**
talent	11	2.38 (9^4)	1.81 (8^1)	2.07 (8^1)	**38.25**	1269.13 (2^1)
auction	170	888.99	7288.73 (95^1)	7053.43 (100^1)	12028.5 (78^1 11^3)	**220.12**
auction (wcsp)		5038.49 (3^1)				
celar	16	0 (16^1)	0 (16^4)	0 (16^1)	1417.19 (6^1)	0 (16^1)
celar (wcsp)		**311.23** (**4^1**)				
spot5	20	114.66 (8^1 8^4)	0 (20^1)	0 (20^1)	**1990.59** (**7^1**)	297.59 (7^1 10^4)
spot5 (wcsp)		76.28 (16^1)				

that the model decomposition was too big. In the *auction* problem, CPLEX is by far the best solver, solving all 170 instances in only 220.12 seconds. This is clearly one of the worst kind of problems for SMT solvers, as it simply consists of a conjunction of arithmetic (0/1) constraints, i.e., it has a trivial Boolean structure and all deductions need to be performed by the theory solver. For *celar*, only `sbdd≤` and Toulbar2 (in the WCSP version) were able to solve some instances (10 and 12 out of 16, respectively). This problem also has a balanced combination of arithmetic constraints and clauses. G12-CPX reported an error (not finding the `int_plus` constraint), and Toulbar2 (in the COP version), Opturion and CPLEX ran out of time on all instances. Finally, for *spot5*, `sbdd≤` was able to solve some instances (13 out of 20), Toulbar2 only 4, CPLEX only 3, and Opturion and G12-CPX ran out of time on all instances. This problem is also well suited for SMT solvers since it basically consists of clauses with equality constraint literals.

We remark that we also tested the G12-Lazy, G12-FD and Gecode 4.2.1 solvers, but since they presented slightly worst performance than G12-CPX we have not included them in Table 8.3.

8.4.3 SBDD incrementality

In this section we study the benefits of using a shared BDD for the objective instead of a linear equation, in particular, the effect that this has on the learning capability of the SMT solver. Note that with this technique we are increasing the Boolean component of the formula at hand and, hence, we should expect some improvement in learning. Finally, we quantify

Table 8.4: Aggregated time in seconds for the solved instances of each problem, with a cutoff of 600s per instance, and indication of the number of unsolved instances due to time out (TO) and memory out (MO).

		SBDD+L		**SBDD-L**		**LIA+L**		**LIA-L**	
Problem	# Inst.	Solving	# TO/MO	Solving	# TO/MO	Solving	# TO	Solving	# TO
sbacp	140	58.03	0	221.30	1	853.56	0	924.83	2
s.l.	30	393.22	0	715.17	0	795.62	4	895.28	4
talent	11	35.36	0	55.57	0	363.43	0	472.09	0
auction	170	9631.19	73/12	5673.22	75/7	8540.76	92	9919.79	90
celar	16	1199.99	6	1299.99	8	10.94	15	8.29	15
spot5	20	1675.90	7	943.96	10	165.48	18	59.99	18

the amount of BDD node reutilization through iterations.

We compare the performance of four solving approaches: using either SBDDs or arithmetic expressions to bound the objective function, with and without using the learning capabilities of the solver. From now on we will denote by SBDD+L, SBDD-L, LIA+L and LIA-L these four possibilities, where LIA stands for linear integer arithmetic expressions and +L/-L indicates the use or not of learning. Note that `WSimply` uses Yices as a back-end solver, and it does not provide statistics about learned clauses. Therefore, we used the Yices commands *push* and *pop* to define backtrack points and to backtrack to that points, respectively, in order to force the solver to forget learned clauses.

Table 8.4 summarizes the solving times for the four options. They do not include neither BDD construction, clause generation nor assertion times. Since solving times are very sensitive to the bounds on the objective function, first of all we solved all the instances with the SBDD+L approach and stored the bounds at each iteration. Then these bounds were passed to the other three methods to obtain their solving times under the similar conditions.

If we first compare SBDD-L and LIA-L we can appreciate that using BDDs considerably reduces the number of timeouts in *still life*, *auction*, *celar* and *spot5*, and the solving time, almost 5 times in *sbacp* and 10 times in *talent scheduling*.

Furthermore, comparing SBDD+L and SBDD-L, we can easily appreciate that learning reduces even more the solving times. In *sbacp* the number of timeouts is reduced to 0 and the sum of solving times is reduced almost 4 times; in *talent scheduling* and *still life* the solving time is reduced almost to the half; and in *celar* and *spot5* the number of timeouts is reduced in 2 and 3 instances respectively. In *auction*, the SBDD+L method has the drawback of increasing the number of memory outs.

If we compare SBDD-L and LIA-L with respect to SBDD+L and LIA+L, we can see that the improvement in terms of number of timeouts and solving time is higher in the SBDD approach than in the LIA approach.

Finally, to quantify the contribution of the shared BDD to the amount of BDD node reutilization, we computed the percentage of reused nodes at each iteration of the optimization process. Our experiments shown an average 50% of node reuse when the solving process was about the 40.73% of the search, and a 80% of node reuse when it was about the 48.72% of the search. This is especially relevant because the BDDs attained hundreds of thousands of nodes.

8.4.4 The Nurse Rostering Problem (NRP)

In this section we show how the new WMDD method, presented at the end of Section 8.2.2, can be applied to the NRP, and we compare it with the current `WSimply` solving methods. To do this we will use the N25 set of the NSPLib, which has 7290 instances with the following settings: 25 nurses, 7 days, 4 shift types, a certain number of nurses required for every shift of each day and a set of nurse preferences. A nurse preference is a value between 1 and 4 (from most desirable to less desirable) for each nurse, shift and day. Nurses are required to work exactly 5 days.

In the model of this problem all constraints are considered as hard except for the nurse preferences which are the soft constraints. The weight of each soft constraint is a value between 1 and 4 representing the penalty of working that shift for each nurse. We want to clarify that in these instances a nurse can assign a penalty of 4 to all the shifts of a specific day. For instance, the following soft constraints (written in `WSimply` language) could be the preferences of nurse 1 for day 3:

```
shift[1,3] = 1 @ 4;
shift[1,3] = 2 @ 4;
shift[1,3] = 3 @ 4;
shift[1,3] = 4 @ 4;
```

Since we have 25 nurses, 7 days and 4 shifts, this means that, in total, these instances have 700 soft constraints and hence, that they have an objective function of 700 PB variables. It is easy to see that these instances wouldn't be very suitable for the SBDD method, the current best `WSimply` solving method, because it could be hard to construct BDDs of 700 layers. Therefore, we propose to use a MDD in order to reduce the number of layers. This can be done in some cases like the NRP taking into account that a nurse can work only one shift per day, and hence, given a nurse n and a day d we can use the function of Algorithm 7 in the WMDD construction variant presented at the end of Section 8.2.2.

Algorithm 7 Mapping function example for the WMDD construction algorithm.

Input: s: nurse shift matrix, n : nurse, d: day
Output: Shift preference value of nurse n for day d

```
if s[n,d] = 1 then
  return w1
else if s[n,d] = 2 then
  return w2
else if s[n,d] = 3 then
  return w3
else if s[n,d] = 4 then
  return w4
end if
```

This way we will only have 175 layers, which is far more reasonable than 700. Although each node will have up to four child nodes the idea is to enhance and take advantage of the node reuse at each layer.

Table 8.5 shows the aggregated time for the solved instances and the number of instances solved. We consider the previously used `yices`, `core`, `dico` and `sbdd`$\leq$ solving methods, plus

Table 8.5: Aggregated time and number of instances solved by the different `WSimply` methds.

Method	Time	# Inst.
`wmdd`	18828.92	5476
`core`	8958.54	5164
`sbdd`$\leq$	-	0
`dico`	-	0
`yices`	-	0

the new method `wmdd` using shared ROMDDs. Note that in the WMDD method we cannot define any order because we have multiple values at each layer.

What is interesting of these results is that only `wmdd` and `core` solving methods are capable of solving instances, being the first one the solving method who solved more instances (5476) with a 60 seconds cutoff. This is very relevant because the other two similar methods, `dico` and `sbdd`$\leq$, weren't able to solve any instance.

Finally, we compare the size of the shared MDDs versus the size of the shared BDDs. To do so we compute the median of the size of the shared MDD (BDD) at all the 7290 instances. The median of the SMDDs is 12308.5 and the median of the SBDDs is 94963.5, which means the SMDD size is about 7.7 times smaller than the SBDD size. We remark that this comparison is approximate because of the time-outs, but it is enough to illustrate the usefulness of the shared MDDs in this problem.

8.5 Conclusions

In this chapter we have presented a new WCSP solving method, implemented in the `WSimply` system, based on using shared ROBDDs to generate propositional clauses representing the constraints bounding the objective function. We think that it opens a promising research line, taking into account that the presented SBDD method clearly outperforms not only the previously implemented solving methods in `WSimply`, but also some state-of-the art solvers, on several problems. We have also shown how to boost the generation of ROBDDs for objective function bounds using previously generated ROBDDs, more precisely constructing a shared ROBDD. In fact, the use of a shared ROBDD to represent the objective function bounds has another advantage aside of reducing the ROBDD construction time, that is the SAT clause reuse from the previously generated nodes, which enhances the learning capabilities of the SMT solver.

We have also presented a generalisation of the ROBDD construction method to represent linear integer constraints, instead of pseudo-Boolean constraints, by means of creating a ROMDD (this method is independently defined in [AS14]). Finally, we have presented a special case where we can use weighted MDD to represent the objective function, and we provide experimental results where the new WMDD method clearly outperforms the previous `WSimply` methods.

In conclusion, once more, increasing the Boolean structure of the problem has shown to be good for solving with SMT. In this case, we have reformulated the objective function into SAT clauses, but we guess that reformulating some other kind of constraints into SAT may help the SMT solver and will be interesting to find which ones.

Chapter 9

Scheduling of Business-to-business (B2B) meetings

In this chapter we use `WSimply` to address the problem of scheduling Business-to-business meetings, which consists of scheduling meetings between research groups, companies and investors in a scientific and technological forum. `WSimply` perfectly fits for working on this problem. It offers several back-end reformulations and it provides interesting modelling features like weighted constraints and meta-constraints.

Business-to-business (B2B) events typically provide bilateral meeting sessions between participants with affine interests. These B2B events occur in several fields like sports, social life, research, etc. In this chapter we describe the application developed to generate the timetable of such meetings in the *4th Forum of the Scientific and Technological Park of the University of Girona*.[1] The goal of this forum is to be a technological marketplace in Girona by bringing the opportunity to companies, research groups, investors, etc., to find future business partnerships.

The scheduling of the meetings is a tough task and requires expertise at distinct levels: on the one side, the human matchmaker should choose the appropriate matches maximizing somehow the potential results of the meetings according to the interests of the participants. On the other hand side, the timetable generation must satisfy several constraints, e.g., avoid meeting collisions, avoid unnecessary idle time between meetings for each participant or too many meeting location changes, etc.

In previous editions of the forum, and in other events with B2B meetings held in the park of the University of Girona, the human matchmaker made both tasks by hand. This process showed to be highly unsatisfactory with respect to the human effort required and also with respect to the final result obtained. Moreover, the growing participation makes the work much harder.

As far as we know, there are no many works dealing with this problem. On the one hand, in [GGSS13] we can find a system that is used by the company *piranha womex AG* for computing matchmaking schedules in several fairs. This system differs from ours in some aspects. For instance, it does not consider forbidden time slots but unpreferred ones, and it allows meeting collisions under the assumption that the companies can send another participant. Moreover, it is based on answer set programming, whereas we follow a different model-and-

[1] `http://www.forumparcudg.com`

solve approach. On the other hand, the system B2Match [b2m] is a commercial system which does not support location change minimization.

Our work focuses on the generation of the meetings' timetable. That is, the human matchmaker provide us with the meetings that she wants to be scheduled and we generate the timetable. We provide both a CP model and a Pseudo-Boolean model for the problem, and compare the efficiency of different solving techniques such as CP, lazy clause generation, SMT, and ILP, on several real instances.

The obtained results for this year's edition of the Forum of the Scientific and Technological Park of the University of Girona have been very satisfactory. Moreover, using instances of previous years we have observed that the number of idle time slots could be dramatically improved.

This chapter is based on results from [BEG+14] and it is organised as follows. In Section 9.1 we define the problem at hand. In Section 9.2 we present the different models considered, and in Section 9.3 we check the efficiency of distinct solvers on these models, using real instances. Conclusions are given in Section 9.4.

9.1 The B2B problem

In the Forum of the Scientific and Technological Park of the University of Girona, the participants answer some questions about their interests and expertise, in the event registration phase. This information is made public to the participants, who may ask for bilateral meetings with other participants with a (normal or high) priority. They also indicate their time availability for the meetings (notice that in the forum there is also a conference with talks given by the participants and therefore they should reserve some time slots for their own talks). Moreover, the participants may ask for meetings in the morning or the afternoon session. Then, according to this information (participants availability and priorities for the desired meetings), the human matchmaker proposes a set of matches (meetings) to be scheduled. Once this schedule is obtained, the matchmaker asks the participants for confirmation of the meetings timetable. Then, the confirmed ones are fixed and the rejected ones are retracted. With this partial timetable, and considering the late arrivals of participants interested in having some meeting, the expert tries to add other meetings in the timetable by individually contacting with the participants. Finally, the distribution of tables is made. Below we formally define the basic problem.

Definition 9.1.1 *Let P be a set of participants, T a list of time slots and L a set of available locations (tables). Let M be a set of unordered pairs of participants in P (meetings to be scheduled). Additionally, for each participant $p \in P$, let $f(p) \subset T$ be a set of forbidden time slots.*

A feasible B2B schedule *S is a total mapping from M to $T \times L$ such that the following constraints are satisfied:*

- *Each participant has at most one meeting scheduled in each time slot.*

(9.1) $\forall m_1, m_2 \in M$ *such that* $m_1 \neq m_2$:

$$\pi_1(S(m_1)) = \pi_1(S(m_2)) \implies m_1 \cap m_2 = \emptyset$$

- *No meeting of a participant is scheduled in one of her forbidden time slots.*

  (9.2) $\forall p \in P, m \in M$:

  $$p \in m \implies \pi_1(S(m)) \notin f(p)$$

- *At most one meeting is scheduled in a given time slot and location.*

  (9.3) $\forall m_1, m_2 \in M$ *such that* $m_1 \neq m_2$:

  $$\pi_1(S(m_1)) = \pi_1(S(m_2)) \implies \pi_2(S(m_1)) \neq \pi_2(S(m_2))$$

The B2B scheduling problem *(B2BSP) is the problem of finding a feasible B2B schedule.*

The B2BSP is clearly in NP, and can be easily proved to be NP-complete by reduction from the restricted timetable problem (RTT) in [EIS75].

Typically, we are interested in schedules that minimize the number of idle time periods. By an *idle time period* we refer to a group of idle time slots between a meeting of a participant and her next meeting. Before formally defining this optimization version of the B2BSP, we need to introduce some auxiliary definitions.

Definition 9.1.2 *Given a B2B schedule S for a set of meetings M, and a participant $p \in P$, we define $L_S(p)$ as the list of meetings in M involving p, ordered by its scheduled time according to S:*

$$L_S(p) = [m_1, \ldots, m_k], \text{ with}$$
$$\forall i \in 1..k : p \in m_i$$
$$\forall m \in M : p \in m \Rightarrow \exists! i \in 1..k : m_i = m$$
$$\forall i \in 1..(k-1) : \pi_1(S(m_i)) < \pi_1(S(m_{i+1}))$$

By $L_S(p)[i]$ we refer to the i-th element of $L_S(p)$, i.e., m_i.

Definition 9.1.3 *We define the* B2B Scheduling Optimization Problem (B2BSOP) *as the problem of finding a feasible B2B schedule S, where the total number of idle time periods of the participants is minimal, i.e., minimizes*

$$\sum_{p \in P} \#\{L_S(p)[i] \mid i \in 1..|L_S(p)| - 1, \pi_1(S(m_i)) + 1 \neq \pi_1(S(m_{i+1}))\} \tag{9.4}$$

It is also quite common to ask for the minimization of location changes between consecutive meetings of the same participants. Due to the requirements of the event organization (there is a phase where the human matchmaker manually arranges new meetings by reusing time slots and participants' availabilities), we do this minimization with the time slots of the meetings already fixed.

Definition 9.1.4 *We define the* B2B location scheduling optimization problem (B2BLOP) *as the problem of, given a B2B schedule S, re-assign to each meeting a location in a way such that the total number of location changes for consecutive meetings of the same participant is minimal, i.e., minimizes*

$$\sum_{p \in P} \#\{L_S(p)[i] \mid$$
$$i \in 1..|L_S(p)| - 1, \pi_1(S(m_i)) + 1 = \pi_1(S(m_{i+1})), \pi_2(S(m_i)) \neq \pi_2(S(m_{i+1}))\}$$

As an additional constraint, we consider the case where meetings may have a morning/afternoon requirement, i.e., that some meetings must necessarily be celebrated in the morning or in the afternoon. Let's then consider that the set of time slots T is divided into two disjoint sets T_1 and T_2 and, moreover, that we have a mapping t from meetings m in M to $\{1, 2, 3\}$, where 1 means that the meeting m must take place at some time slot in T_1, 2 means that it must take place at some time slot in T_2, and 3 means that it does not matter. Then the schedule should also satisfy the following requirement:

(9.5) $\forall m \in M$:

$$(t(m) = 1 \Longrightarrow \pi_1(S(m)) \in T_1) \ \wedge \ (t(m) = 2 \Longrightarrow \pi_1(S(m)) \in T_2)$$

Summing up, the scheduling process goes as follows:

1. The human matchmaker arranges the meetings taking into account the participants' requirements and preferences (who they want to meet, with which priority, and a possible restriction to the morning or afternoon frame for the meeting) and their forbidden hours (for example, the hours where the participant is giving a talk at the event).
2. We solve the B2BSOP with the chosen meetings.
3. The human matchmaker removes the revoked meetings (for instance, one participant may not be interested in a meeting requested by another participant) from the solution found in Step 2, and manually adds new last-arrival meetings.
4. We solve the B2BLOP with the meetings proposed in Step 3.

9.2 Models

In order to model the aforementioned problem we have used `WSimply`, because it supports weighted constraints and several predefined meta-constraints which allow to express homogeneity in soft constraints violations, in order to enforce fairness of solutions. For the sake of completeness, we also consider MiniZinc which is supported by several state-of-the-art CP solvers. A proposal for incorporating similar ideas into MiniZinc has been presented in [ABP+11b].

Moreover, since a priori it is not clear if a high-level or a low-level model will be better in terms of performance, we have considered two different models: a CP model (with finite domain variables like t_i, denoting the time at which meeting i takes place) and a Pseudo-Boolean model (with 0/1 variables like $x_{i,j}$, stating that meeting i is celebrated at time j).

9.2.1 A WCSP model for the B2BSOP

Here we give the `WSimply` version of the CP model for the B2BSOP, expressed as a weighted CSP.

Parameters

```
int nParticipants;                    % # of participants
int nMeetings;                        % # of meetings
int nTables;                          % # of locations
```

```
int nTimeSlots;                          % # of time slots
int nMorningSlots;                       % # of morning time slots
int meetings[nMeetings,3];               % Meetings to schedule
int tnForbidden;                         % Total # of forbidden slots
int forbidden[tnForbidden];              % Forbidden slots
int indexForbidden[nParticipants+1];     % Start indices in forbidden
int nMeetingsParticipant[nParticipants]; % # of meetings of each part.
```

The number of afternoon time slots is `nTimeSlots` − `nMorningSlots`, and for this reason it is not explicitly defined.

The `meetings` matrix has three columns, denoting the number of the first participant, the number of the second participant and the type of session required (1: morning, 2: afternoon, 3: don't care) for each meeting to be scheduled.

The `indexForbidden` array contains the indices where the forbidden time slots of each participant do begin in the `forbidden` array. This array has an extra position in order to ease the modelling of the constraints (see the `Forall` statements below), with value `indexForbidden[nParticipants+1] = tnForbidden+1`.

Variables and domains

```
Dom dTimeSlots = [1..nTimeSlots];
Dom dUsedSlots = [0..1];
Dom dTables    = [0..nTables];

IntVar schedule[nMeetings]::dTimeSlots;
IntVar tablesSlot[nTimeSlots]::dTables;
IntVar usedSlots[nParticipants,nTimeSlots]::dUsedSlots;
IntVar fromSlots[nParticipants,nTimeSlots]::dUsedSlots;
```

The array variable `schedule` shall contain the time slot assigned to each meeting.

The array variable `tablesSlot` will indicate the number of meetings to be celebrated at each time slot (and hence the number of tables needed). Note that by setting the domain to `[0..nTables]` we are already restricting the number of tables available.

Variable `usedSlots` is a two dimensional 0/1 array representing, for each participant i and time slot j, if i has a meeting scheduled at time j.

Finally, variable `fromSlots` is a two dimensional 0/1 array such that, for each participant i, `fromSlots`$[i, j] = 1$ for all j from the first time slot at which a meeting for i is scheduled on.

These two last variables are used for optimization, as shown below.

Constraints

- *Each participant has at most one meeting scheduled at each time slot*, i.e., constraint (9.1). We force two meetings sharing one member to be scheduled at a different time slot:

```
Forall (n in [1..nMeetings]) {
  Forall (m in [n+1..nMeetings]) {
    If (meetings[n,1] = meetings[m,1] Or
        meetings[n,2] = meetings[m,1] Or
        meetings[n,1] = meetings[m,2] Or
        meetings[n,2] = meetings[m,2])
```

```
    Then { schedule[n] <> schedule[m]; };
  };
};
```

- *No meeting of a participant is scheduled in one of her forbidden time slots*, i.e., constraint (9.2). We force, for each meeting, to be scheduled in a time slot distinct from all forbidden time slots for both members of the meeting:

```
Forall(j in [1..nMeetings]) {
  Forall(k in [ indexForbidden[meetings[j,1]]..
                indexForbidden[meetings[j,1]+1]-1 ]) {
    schedule[j] <> forbidden[k];
  };
  Forall(k in [ indexForbidden[meetings[j,2]]..
                indexForbidden[meetings[j,2]+1]-1 ]) {
    schedule[j] <> forbidden[k];
  };
};
```

- *At most one meeting is scheduled in a given time slot and location*, i.e., constraint (9.3). This constraint is ensured by matching the number of meetings scheduled in time slot i with `tablesSlot`$[i]$, whose value is bounded by the number of tables available:

```
Forall(i in [1..nTimeSlots]) {
 Sum([ If_Then_Else(schedule[n] = i)(1)(0) | n in [1..nMeetings] ],
     tablesSlot[i]);
};
```

  Note that we are not assigning a particular table to each meeting, but just forcing that there are enough tables available for the meetings taking place at the same time.

- *Each meeting must be scheduled in a required (if any) set of time slots*, i.e., constraint (9.5). Since we know the number of morning time slots, we can easily enforce this constraint:

```
Forall (n in [1..nMeetings]) {
  If (meetings[n,3] = 1) Then {schedule[n] =< nMorningSlots;}
  Else {If (meetings[n,3] = 2) Then {schedule[n] > nMorningSlots;};
  };
};
```

- *Channeling constraints.* In order to be able to minimize the number of idle time periods, i.e., objective function (9.4), we introduce channeling constraints mapping the variable `schedule` to the variable `usedSlots`. That is, if meeting k is scheduled at time slot j, we need to state that time j is used by both members of meeting k, by setting accordingly the corresponding value in `usedSlots`:

```
Forall(j in [1..nTimeSlots]) {
  Forall(k in [1..nMeetings]) {
```

```
      (schedule[k] = j) Implies
         (usedSlots[meetings[k,1],j] = 1 And
          usedSlots[meetings[k,2],j] = 1);
  };
};
```

In the reverse direction, for each participant e, her number of meetings as derived from `usedSlots` must match her known total number of meetings `nMeetingsParticipant`$[e]$:

```
Forall(e in [1..nParticipants]) {
  Sum([ usedSlots[e,f] | f in [1..nTimeSlots] ],
      nMeetingsParticipant[e]);
};
```

Next, we impose the required constraints on the variable `fromSlots`:

```
Forall(e in [1..nParticipants]) {
  Forall(f in [1..nTimeSlots]) {
    usedSlots[e,f] = 1 Implies fromSlots[e,f] = 1;
  };

  Forall(f in [1..nTimeSlots-1]) {
    fromSlots[e,f] = 1 Implies fromSlots[e,f+1] = 1;
  };
};
```

It is worth noting that, for any participant e, having `fromSlots`$[e, f] = 1$ for all f is possible, even if e has no meeting at time slot 1. However, the soft constraints used for optimization will prevent this from happening, as commented below.

Optimization Minimization of the objective function (9.4) is achieved by means of soft constraints, where a group of contiguous idle time slots between two meetings of the same participant is given a cost of 1.

Soft constraints are labeled with `Holes`$[e, f]$, where e denotes a participant and f denotes a time slot, and state that if e does not have any meeting in time slot f, but it has some meeting before, then she does not have any meeting in the following time slot:

```
Forall(e in [1..nParticipants], f in [1..nTimeSlots-1]) {
  #Holes[e,f]:((usedSlots[e,f] = 0 And fromSlots[e,f] = 1) Implies
               usedSlots[e,f+1] = 0)@{1};
};
```

We claim that, with these constraints, an optimal solution will be one having the least number of groups of contiguous idle time slots between meetings of the same participant. Note that, for each participant, we increase the cost by 1 for each meeting following some idle period. Moreover, it is not difficult to see that the (possibly null) period of time preceding the first meeting of a participant e will have no cost, since `fromSlots`$[e, f]$ can freely take value 0 for all f prior to the time of the first meeting.

Finally, since we are not only interested in optimal solutions, but in fair ones, we can add the following meta-constraint, stating that the difference between the number of idle periods of any two distinct participants is, e.g., at most 2:

```
homogeneousAbsoluteNumber([ [ Holes[e,f] | f in [1..nTimeSlots-1] ] |
                            e in [1..nParticipants] ], 2);
```

Note that this meta-constraint makes use of the labels introduced in the soft constraints. It has as first argument a list of lists ll of soft constraint labels, and as second argument a natural number n. It ensures that, for each pair of lists in ll, the difference between the number of violated constraints in the two lists is at most n.

The `WSimply` model presented above has been translated to MiniZinc in order to test the performance of a greater number of different solvers (see Section 9.3). The translation of hard constraints is straightforward, due to the similarities between the two languages. Soft constraints have been translated into a linear objective function, as they are not directly supported in MiniZinc. The meta-constraint used in the `WSimply` model has been translated into the MiniZinc constraints that would result from the translation process implemented in the `WSimply` compiler.

9.2.2 A PB model for the B2BSOP

Here we give the `WSimply` version of the Pseudo-Boolean model for the B2BSOP. The parameters are the same as in the CP model but, in this case, we only use 0/1 variables.

Variables and domains

```
Dom pseudo = [0..1];

IntVar schedule[nMeetings,nTimeSlots]::pseudo;
IntVar usedSlots[nParticipants,nTimeSlots]::pseudo;
IntVar fromSlots[nParticipants,nTimeSlots]::pseudo;
IntVar holes[nParticipants,nTimeSlots-1]::pseudo;
IntVar nHoles[nParticipants,5]::pseudo;
IntVar max[5]::pseudo;
IntVar min[5]::pseudo;
```

Here, the array variable `schedule` shall contain a 1 at position i, j if and only if meeting i is celebrated at time slot j.

Variables `usedSlots` and `fromSlots` are the same as in the CP case.

Variable `holes` will indicate, for each participant, when a time slot is the last of an idle period of time.

Variable `nHoles` will hold the 5-bit binary representation of the number of idle time periods of each participant, i.e., the number of groups of contiguous idle time slots between meetings of each participant.

Variables `max` and `min` will hold the 5-bit binary representation of an upper bound and a lower bound of the maximum and minimum values in `nHoles`, respectively. As we will see, these variables will be used to enforce fairness of solutions, by restricting their difference to be less than a certain value.

All variables but `schedule` are only necessary for optimization.

Constraints

- *Each participant has at most one meeting scheduled at each time slot*:

```
Forall (f in [1..nTimeSlots]) {
  Forall (n in [1..nParticipants]) {
    AtMost([ schedule[m,f] |
               m in [1..nMeetings],
               (meetings[m,1] = n Or meetings[m,2] = n) ], 1, 1);
  };
};
```

The `atMost`(l, e, n) global constraint (where l is a list, e is an expression of the same type of the elements in l, and n is an integer arithmetic expression) forces the number of elements in l that match e to be at most n.

- *No meeting of a participant is scheduled in one of her forbidden time slots*:

```
Forall (r in [1..nMeetings]) {
  Forall (p in [indexForbidden[meetings[r,1]]..
                indexForbidden[meetings[r,1]+1]-1]) {
      schedule[r,forbidden[p]] = 0;
  };
  Forall (p in [indexForbidden[meetings[r,2]]..
                indexForbidden[meetings[r,2]+1]-1]) {
      schedule[r,forbidden[p]] = 0;
  };
};
```

- *At most one meeting is scheduled in a given time slot and location.* We ensure this by forcing that no more than `nTables` meetings are scheduled in the same time slot:

```
Forall(f in [1..nTimeSlots]) {
  AtMost([schedule[n,f] | n in [1..nMeetings]],1,nTables);
};
```

- *Each meeting must be scheduled in the required (if any) set of time slots.* By means of sums, we force that each meeting is scheduled exactly once in its required set of time slots:

```
Forall (n in [1..nMeetings]) {
  If (meetings[n,3] = 1) Then {
    Sum([ schedule[n,f] | f in [1..nMorningSlots] ], 1);
    [ schedule[n,f] = 0 | f in [nMorningSlots+1..nTimeSlots] ];
  } Else {
  If (meetings[n,3] = 2 ) Then {
    [ schedule[n,f] = 0 | f in [1..nMorningSlots] ];
    Sum([ schedule[n,f] | f in [nMorningSlots+1..nTimeSlots] ], 1);
  } Else {
    Sum([ schedule[n,f] | f in [1..nTimeSlots] ], 1);
  }};
};
```

Note that list comprehensions can be used to post constraints.

- *Channeling constraints.* The channeling constraints are analogous to before:

```
Forall(j in [1..nTimeSlots]) {
  Forall(k in [1..nMeetings]) {
    (schedule[k,j] = 1) Implies
       (usedSlots[meetings[k,1],j] = 1) And
       (usedSlots[meetings[k,2],j] = 1));
  };
};

Forall(e in [1..nParticipants]) {
  Sum([ usedSlots[e,f] | f in [1..nTimeSlots] ],
      nMeetingsParticipant[e]);
};

Forall(e in [1..nParticipants]) {
  Forall(f in [1..nTimeSlots]) {
    usedSlots[e,f] = 1 Implies fromSlots[e,f] = 1;
  };

  Forall(f in [1..nTimeSlots-1]) {
    fromSlots[e,f] = 1 Implies fromSlots[e,f+1] = 1;
  };
};
```

Optimization The soft constraints are the same as in the CP model:

```
Forall(e in [1..nParticipants], f in [1..nTimeSlots-1]) {
  ((usedSlots[e,f] = 0 And fromSlots[e,f] = 1) Implies
  usedSlots[e,f+1] = 0)@{1};
};
```

The homogeneity meta-constraint `homogeneousAbsoluteNumber` used in the CP model cannot be used in the Pseudo-Boolean model since, in its current implementation, `WSimply` will translate this meta-constraint into a set of non (Pseudo-)Boolean constraints. For this reason, this meta-constraint needs to be simulated here. We proceed as follows.

On the one hand, we post the constraints for the array variable `holes` which contains, for each participant, the last time slot of each idle period of time:

```
Forall(e in [1..nParticipants], f in [1..nTimeSlots-1]) {
  (usedSlots[e,f] = 0 And fromSlots[e,f] = 1 And usedSlots[e,f+1] = 1)
      Implies holes[e,f]=1;
  holes[e,f]=1 Implies
  (usedSlots[e,f] = 0 And fromSlots[e,f] = 1 And usedSlots[e,f+1] = 1);
};
```

On the other hand, we post the constraints defining the 5-bit binary representation of the number of idle time periods of each participant (the `sum1` function returns the sum of the elements in the list it receives as argument):

```
Forall(e in [1..nParticipants]){
  sum1([ holes[e,f] | f in [nTimeSlots-1] ]) =
  16*nHoles[e,1]+8*nHoles[e,2]+4*nHoles[e,3]+2*nHoles[e,4]+nHoles[e,5];
};
```

Finally, we constrain the difference between the number of idle time periods of any two distinct participants to be at most 2. We do this by constraining the difference between an upper bound and a lower bound of the maximal and minimal number, respectively, of idle time periods of all participants, to be at most 2:

```
Forall(e in [1..nParticipants]){
  16*max[1]+8*max[2]+4*max[3]+2*max[4]+max[5] >=
  16*nHoles[e,1]+8*nHoles[e,2]+4*nHoles[e,3]+2*nHoles[e,4]+nHoles[e,5];

  16*nHoles[e,1]+8*nHoles[e,2]+4*nHoles[e,3]+2*nHoles[e,4]+nHoles[e,5]
  >= 16*min[1]+8*min[2]+4*min[3]+2*min[4]+min[5];
};

2 >= 16*max[1]+8*max[2]+4*max[3]+2*max[4]+max[5] -
     16*min[1]+8*min[2]+4*min[3]+2*min[4]+min[5];
```

9.3 Experimental results

In this section we compare the performance of several state-of-the-art CSP, WCSP, ILP and PB solvers with the proposed models. We also analyse which is the improvement of our solution over some handmade solutions from past editions of the forum.

As said, apart from the two (CP and PB) `WSimply` models presented, we have also considered a MiniZinc model in order to test the performance of a greater number of different solvers. The MiniZinc model considered is an accurate translation of the first `WSimply` (CP) model. All models and data can be found in `http://imae.udg.edu/recerca/lap/simply/`.

For solving the `WSimply` instances we have used the SMT solver Yices 1.0.33 [DdM06b] through its API, with two different solving methods: WPM1 (described in Chapter 7), and SBDD (described in Chapter 8).

We have also considered the translation of the CP model written in `WSimply` into ILP, and used IBM ILOG CPLEX 12.6 for solving the resulting instances.

For solving the MiniZinc instances we have used Gecode 4.2.1 [SLT10], a state-of-the-art CP solver, and Opturion 1.0.2 [opt], winner of the 2013 MiniZinc Challenge[2] in the free search category, using lazy clause generation [OSC09].

For solving the PB `WSimply` instances we have used CPLEX 12.6, SCIP 3.0.1 [Ach09] and clasp 3.1.0 [GKNS07]. The two former obtained the best results in the last PB competition[3] in the category "optimisation, small integers, linear constraints", which fits our problem. However, the latter has shown a much better performance on this problem.

All experiments have been run on a cluster of Intel® Xeon™CPU@3.1GHz machines, with 8GB of RAM, under 64-bit CentOS release 6.3, kernel 2.6.32, except for the experiments with Opturion, where we have used a slightly different computer (Intel® Core™CPU@2.8GHz,

[2] `http://www.minizinc.org/challenge2013/results2013.html`
[3] `http://www.cril.univ-artois.fr/PB12/`

with 12GB of RAM, under 64-bit Ubuntu 12.04.3, kernel 3.2.0) due to some library dependence problems. We have run each instance with a cutoff of 2 hours.

We have considered four industrial instances provided by the human expert: instances tic-2012a and tic-2013a, from past editions of a technology and health forum, and instances forum-2013a and forum-2014a, from the previous year and this year's editions of the scientific and technological forum.

Taking as basis these four instances, we have crafted five more instances of distinct hardness by increasing the number of meetings, reducing the number of locations and changing the morning/afternoon preferences of some meetings. Table 9.1 summarizes the results of the experiments.

Table 9.1: Solving time (in seconds) and optimum found (number of idle time periods) per instance, model and solver. The instances are tagged with (#meetings, #participants, #locations, #time slots, #morning time slots). TO stands for time out and MO for memory out. The cutoff is 2 hours. For aborted executions we report the (sub)optimum found if the solver reported any. Best running times and upper bounds of the objective function are indicated in boldface.

Instance	CP Model										PB Model					
	WPM1		WSimply SBDD		CPLEX		MINIZINC Gecode		MINIZINC Opturion		CPLEX		WSimply SCIP		clasp	
tic-2012a (125,42,21,8,0)	2.0	0	2.7	0	3375.6	0	1.4	0	8.1	0	7.3	0	126.1	0	**0.1**	**0**
tic-2012c (125,42,16,8,0)	51.1	0	235.4	0	TO	4	TO	-	2206.2	0	**18.0**	**0**	1692.7	0	977.9	0
tic-2013a (180,47,21,10,0)	25.0	0	65.2	0	TO	9	2923.1	0	394.5	0	1345.3	0	TO	13	**2.1**	**0**
tic-2013b (184,46,21,10,0)	5.6	0	24.1	0	TO	8	3.4	0	108.0	0	121.0	0	4975.4	0	**2.1**	**0**
tic-2013c (180,47,19,10,0)	TO	-	TO	8	TO	18	TO	-	TO	8	**TO**	**4**	TO	31	TO	-
forum-2013a (154,70,14,21,13)	5542.3	0	3128.1	0	MO	83	TO	-	1142.8	0	TO	20	TO	-	**52.4**	**0**
forum-2013b (195,76,14,21,13)	TO	-	**TO**	**12**	TO	-	TO	-	TO	50	MO	-	TO	-	TO	24
forum-2013c (154,70,12,21,13)	TO	-	**TO**	**20**	MO	50	TO	-	TO	23	TO	30	TO	-	TO	-
forum-2014a (302,78,22,22,12)	TO	-	**TO**	**7**	TO	-	TO	-	TO	90	MO	-	TO	-	TO	-

Looking at the results, it can be said that clasp is the best solver on the easier instances (only beaten by CPLEX in two cases), and SBDD is the best on the harder instances, both of them using conflict driven solvers. The latter is also the most robust method when considering all instances, with a good compromise between solving time and quality of the solutions. Due to the fact that SBDD is based on representing the objective function as a BDD, and iteratively calling the decision procedure (an SMT solver) with successively tighter bounds, we can obtain a feasible solution at each stage of the optimization process. The WPM1 method also exhibits good performance on many instances, but it cannot provide suboptimal solutions as, roughly, it approaches to solutions from the unsatisfiable side.

An interesting aspect is that, in all real instances from past editions (tic-2012a, tic-2013a and forum-2013a) we obtained an optimum of 0 idle time periods between meetings of the same participant, whereas the expert handmade solutions included 20, 39 and 99 idle time periods respectively, as shown in Table 9.2. Note also that only for some crafted instances, and the bigger real instance from the last forum, we could not certify the optimality of the solutions found within two hours.

In Section 9.2 we have not included the model used for the B2BLOP for the sake of simplicity. However, one aspect of the model that deserves a comment is the use of meta-constraints, to ensure fairness in the number of location changes of the participants. With the meta-constraint

```
homogeneousAbsoluteNumber([[Changes[e,f] | f in [1..nTimeSlots-1]]
                           | e in [1..nParticipants]], Hfactor);
```

the user can look for solutions where the difference on the number of location changes between participants is at most `HFactor`, and with the meta-constraint

```
maxCost([[Changes[e,f] | f in [1..nTimeSlots]]
         | e in [1..nParticipants]], MaxChanges);
```

the user can look also for solutions where the number of location changes per participant is at most `MaxChanges`.

We have solved the B2BLOP with the schedules obtained from real instances of previous editions (tic-2012a, tic-2013a and forum-2013a), in order to compare the obtained results to the handmade ones, and with the schedule proposed for this year's edition of the forum. This last schedule, which we refer to as forum-2014a-t, has been obtained by the human expert by retracting 6 cancelled meetings, and by adding 15 last arrival meetings, to the schedule obtained for forum-2014a with the WPM1 method, in approximately 2.5 hours. The resulting B2BLOP instances have been solved with the WPM1 method in approximately 3, 120, 420 and 480 seconds respectively.

In Table 9.2 we provide a comparison on the quality of the solutions with respect to the number of idle time periods and location changes, when solved by hand and with `WSimply`.

Table 9.2: Number of idle time periods and location changes for the real instances when solved by hand and with `WSimply`. The maximum difference (homogeneity) between those numbers for distinct participants is given between parentheses.

Instance	# idle periods		# location changes	
	Handmade	`WSimply`	Handmade	`WSimply`
tic-2012a	20 (4)	0 (0)	112 (7)	103 (3)
tic-2013a	39 (4)	0 (0)	191 (9)	156 (4)
forum-2013a	99 (5)	0 (0)	27 (5)	105 (4)
forum-2014a-t		22 (2)		249 (6)

Note that the number of idle time periods is reduced to 0 when solving the problem with `WSimply` in almost all cases. In spite of this, we are still able to reduce the number of location changes with respect to the handmade solutions, except for the case of forum-2013a. But the solution obtained with `WSimply` in this case is still significantly better than the handmade one, which implied 99 idle time periods and was far less homogeneous. In any case, we are prioritizing the minimization of idle time periods of participants and the fairness of solutions, and leave location change minimization as a secondary desirable property.

9.4 Conclusion

In this chapter we have provided two distinct models for the B2BSOP and compared the efficiency of several types of solvers when dealing with some industrial and crafted instances of this problem, in a 'model-and-solve' approach. We also provide several new nontrivial industrial timetabling instances to the community.

The solutions found have been highly satisfactory for the human matchmaker, as they dramatically improve the handmade ones with respect to the number of idle time periods as well as location changes for the participants and, obviously, with quite less effort. Another aspect that the human matchmaker has really appreciated is the facility of adding meta-constraints to the model, like the ones we have used for achieving some level of fairness in the solutions. Fairness is crucial since participants may complain if they have the feeling of being discriminated, either with respect to idle time periods or with respect to location changes. The possibility of being able to fix partial solutions and adding new meetings to schedule has also been appreciated, since this is a hard task to do typically the day before the event due to last meeting request arrivals.

With respect to performance, we have noted that clasp is especially good on small instances, while `WSimply` (with the SBDD approach) appears to be better on bigger ones. However, it is not easy to draw conclusions, as we have not tuned the model for any solver in particular.

Recently, in [PRR15] better encodings for MIP an CP using global constraints are obtained. Also in [BGSV15] a MaxSAT model is shown to be the state-of-the-art solving approach.

Chapter 10

Anomalies Modelling Combinatorial Problems using SMT

Much research on modelling combinatorial problems for a solver compares alternative models of a problem in an attempt to understand the characteristics of good models. The aim is to discover principles and heuristics that in the future would guide someone to develop good models or to select the more effective model from a set of alternatives.

For many years this methodology has been moderately successful in studying modelling for SAT solvers and for finite-domain constraint solvers, and now it is also applied for SMT. But, as we show in this chapter, sometimes this task could be more difficult in SMT, because SMT solvers, in some cases, have apparently shown an erratic behaviour.

In this chapter we present four of the more extreme anomalies that we have encountered in our work. Of course we label these phenomena as anomalies because we have no explanation for them and they run counter to our intuitions, even taking into account that SMT solvers are very complicate tools (see e.g. [BSST09]) and a small change in the formula can result in a large change in the behaviour of the solver. For example, it is well known that changing the order of the clauses in an SMT formula can greatly increase or decrease the solution time. The work reported here is solely concerned with SMT using the QF_LIA logic and we refer to this combination as SMT(QF_LIA). We investigate four SMT(QF_LIA) solvers and use all with their default settings.

Problem instances and data associated with this research can be found at `http://www.cs.york.ac.uk/aig/constraints/SMT/`.

This chapter is based on results from [FP14] and it is organised as follows. First of all, we present two anomalies found representing finite domains. Then, we present two anomalies found representing the Pedigree Reconstruction Problem. Finally, we present some conclusions about the anomalies.

10.1 Anomalies in Representing Finite Domains

The first two anomalies arise in representing finite integer domains and considering two kinds of propagation.

Throughout we consider the representation of a variable x with domain $D = \{d_1, \ldots, d_n\}$ where $d_1 \leq d_2 \leq \cdots \leq d_n$. We use $\overline{D}$ to denote $\{d' | d_1 < d' < d_n \text{ and } d' \notin D\}$.

To simplify the presentation we use the following schemas:

- $ALO(x) \stackrel{\text{def}}{=} \bigvee_{d \in D} x = d.$
- $BOUND(x) \stackrel{\text{def}}{=} (d_1 \leq x \leq d_n).$
- $NEG(x) \stackrel{\text{def}}{=} \bigwedge_{d \in \overline{D}} \neg(x = d).$

In all the experiments of Anomalies 1 and 2 we have used the following four SMT(QF_LIA) solvers: Yices 1.0.35 [DdM06b], Yices 2.1.0, Z3-4.1 [dMB08] and MathSat 5.1.10 [CGSS13a]. The input formulas for the first three solvers are written using SMT-Lib 1.2 and using SMT-Lib 2 for the fourth. The experiments were run on a cluster of Intel® Xeon™CPU@3.1GHz machines, with 8GB of RAM, under 64-bit CentOS release 6.3 (kernel 2.6.32).

10.1.1 Anomaly 1

Anomaly 1 arises in a test to measure how fast different finite domain representations are in recognising unsatisfiability when all of the domain values are eliminated.

Here we consider the domain of variable x to be the contiguous set of values $1..n$. The domain values are all eliminated by asserting the following formula:

$$UNSAT\_ALL(x) \stackrel{\text{def}}{=} \bigwedge_{d \in D} \neg(x = d)$$

We conjoin this with each of three domain representations resulting in three test cases

1. $ALO(x) \wedge UNSAT\_ALL(x)$
2. $BOUND(x) \wedge UNSAT\_ALL(x)$
3. $BOUND(x) \wedge ALO(x) \wedge UNSAT\_ALL(x)$

We measured the solve time of each of these formulas on each of the four SMT(QF_LIA) solvers. Each of Figures 10.1–10.4 shows the run time as a function of domain size, n, of one solver on the three domain encodings. The first three of these figures show that each of Yices 1, Yices 2 and MatSat 5 perform similarly on the three encodings. Indeed, in each case two of the curves overlay each other almost perfectly. MathSat 5 is remarkable in that the solve time is almost independent of both domain size and encoding.

The anomaly occurs with Z3; as domain size increases the bounds-only encoding performs far worse than the other two encodings, which behave almost identically. Notice that the scale in Fig. 10.4 goes over 100 seconds, whereas the scales in the other plots go to only 10 seconds. More remarkably, with the bounds-only representation Z3 performs search. For example, with domain size 10000 it reports having made 5843 decisions and 5460 conflicts, while in the ALO representations does not report any information about decisions or conflicts.

In contrast, the other solvers don't report having made any decision as expected, but they report different number of conflicts. On all problem instances Yices 1 reports 1 conflict, Yices 2 does not report any statistic when the instance is unsatisfiable and MathSat 5 reports 1 conflict with the bounds-only representation and 0 conflicts in the ALO representations. Finally, MathSat 5 is the only solver reporting calls to theory solvers, calling 5001 times the linear arithmetic theory solver with the domain size 10000. A summary of the reported conflicts and calls to the linear arithmetic theory solver by the SMT solvers can be found in Table 10.1.

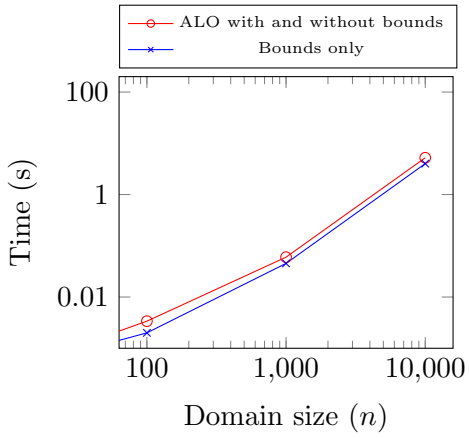

Figure 10.1: Anomaly 1: Yices 1

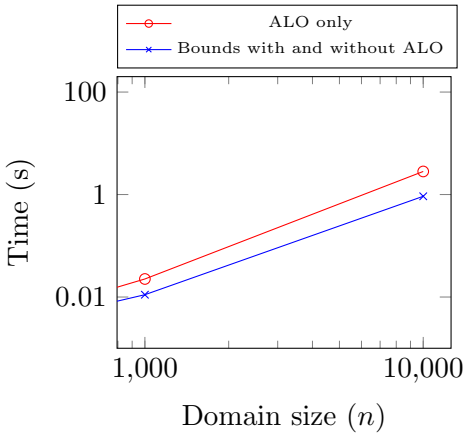

Figure 10.2: Anomaly 1: Yices 2

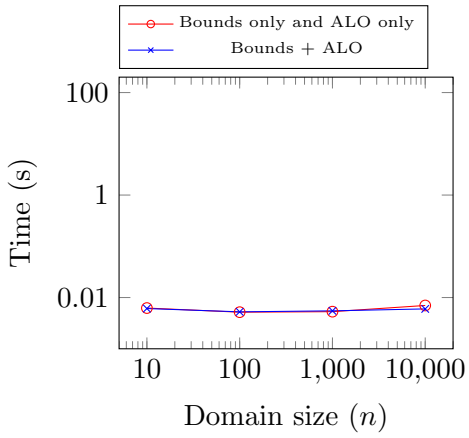

Figure 10.3: Anomaly 1: MathSat 5

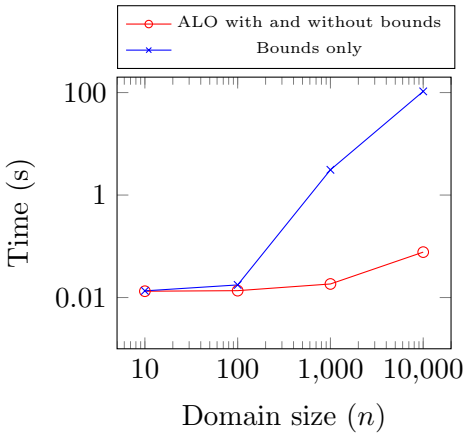

Figure 10.4: Anomaly 1: Z3

10.1.2 Anomaly 2

Anomaly 2 arises in a test to measure how fast different domain representations are in recognising that a variable has a particular value because all but one of it's domain values are eliminated.

Here we consider the domain of variable x to be $\{1, 3, 5, \ldots, 2n-1\}$, i.e., the first n odd natural numbers. All but one of the domain values are eliminated by asserting the following formula:

$$ONLY1(x, v) \stackrel{\text{def}}{=} \bigwedge_{d \in D \setminus \{v\}} \neg(x = d)$$

We have evaluated the performance of four solvers in determining the satisfiability of $ONLY1(x, d_n)$ when conjoined with each of four domain representations. Anomaly 2 arises in the conjunction with one particular domain representation

$$BOUND(x) \wedge NEG(s) \wedge ONLY1(x, d_n)$$

This formula has been solved with each of the four SMT(QF_LIA) solvers. The resulting

Table 10.1: Anomaly 1: Number of conflicts and number of calls to the linear arithmetic theory solver (between parenthesis) for instance with domain size 10000 for each SMT solver. n/a means that the solver has not reported any information.

	Bounds	ALO	Bounds + ALO
Yices 1	1	1	1
Yices 2	n/a	n/a	n/a
Z3	5460	n/a	n/a
MathSat 5	1 (5001)	0 (0)	0 (0)

solve times as a function of domain size, n, are shown in Figure 10.5. Here we see that Yices 2 and MathSat 5 scale very well with increasing n. In contrast, the solve time of Yices 1 increases rapidly with increasing n and if n is 13,000 or more the solver encounters a memory overflow.

The main anomaly here is that the solve time of Z3 does not increase monotonically with n. Furthermore, Z3 performs search in solving this simple problem and the metrics that quantify this search also do not increase monotonically. As shown in Figure 10.6, their non-monotonic behaviour tracks that of the runtime measurement. In contrast, the three other solvers report a constant number of conflicts and decisions; on all instances Yices 1 and Yices 2 report 0 conflict and MathSat 5 reports 1 conflict. MathSat makes $2n + 3$ calls to the QF_LIA theory solver on instances with domain size n; the other solvers do not report this statistic.

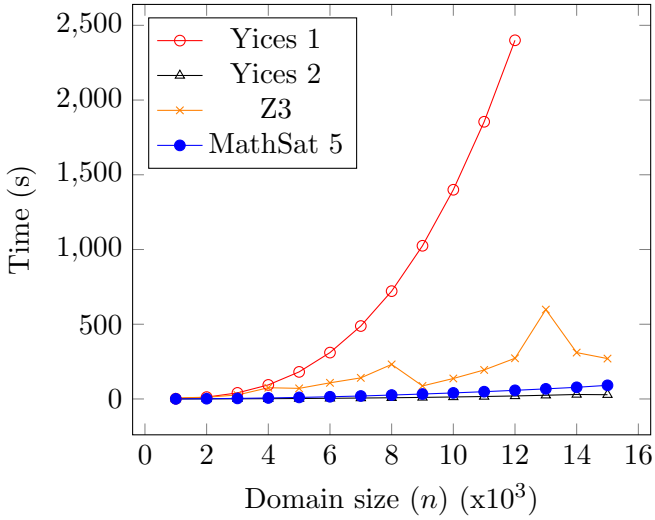

Figure 10.5: Anomaly 2

10.2 Anomalies in representing the Pedigree Reconstruction Problem

The second two anomalies arise in representing the Pedigree Reconstruction Problem (PRP)[CBJS13], which involves identifying relatives amongst a group G of individuals from genetic marker

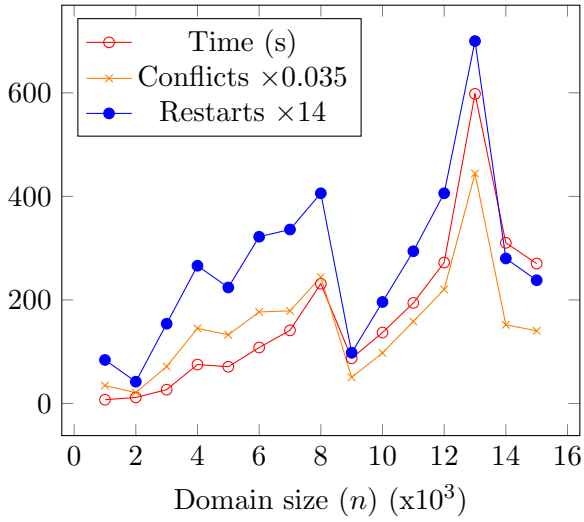

Figure 10.6: Z3 statistics in function of domain size (n). The number of decisions are not shown because the decisions ×0.033 curve will look indistinguishable to the conflicts ×0.035 curve.

data.

In particular, the goal here is to find the maximum likelihood pedigree. A pedigree for G assigns to each individual i in G a parent set, which takes one of three forms:

- j, k, where i, j and k are distinct members of G. This indicates that the parents of i are j and k.
- j, where i and j are distinct members of G. This indicates that j is a parent of i and the other parent of i is not in G.
- $\emptyset$, which indicates that neither parent of i is in G.

For each member $i \in G$, the genetic markers of the members of G determine a probability distribution over all possible parent sets of i. In typical problem instances most potential parent sets of i have probability zero.

The probability of a pedigree is the product of the probability that each $i \in G$ has the parent set assigned by the pedigree.

Every assignment of individuals to parent sets is not a valid pedigree. Sexual reproduction imposes two constraints:

acyclicity: No individual can be an ancestor of itself, and

gender consistency: The two parents of an individual have opposite gender. The genetic marker data does not identify the gender of the individuals, so gender consistency requires that a gender could be assigned to each individual so that the parents of every individual have opposite genders. As an example, an assignment in which a and b have a child, b and c have a child, and a and c have a child is not gender consistent.

Summing up, the pedigree reconstruction problem is: Given G a finite set of individuals and for each $i \in G$ a probability distribution over all possible parent sets of i, find a maximum

likelihood assignment A of a parent set to each $i \in G$ such that A is acyclic and gender consistent.

Following Cussens [CBJS13] we simplify our model by assuming that we are given not the probability of each parent set but rather the log of that probability. This enables us to use a linear objective function: the sum of the logarithm of the probability of each assigned parentset. This value is to be maximised.

Table 10.2 shows the basic model written in SMT(QF_LIA) of the PRP. The model represents the individuals of G by the integers $1..n$. For each individual i the instance specifies the probability distribution over the possible parent sets of i by three parameters.

- k_i is the number of possible parent sets of i that have non-zero probability
- $pps_i[1..k-i]$ is an array whose elements are the k_i parent sets i that have non-zero probability.
- $logLikelihood_i[1..k_i]$is an array such that element j is the log of the probability that i has parentset $pps_i[j]$.

For each individual i the model has a decision variable $parentset_i$ such that assigning it p indicates the decision to assign individual i the parentset $pps_i[p]$. The decision variable $globalValue$ is the objective value to be maximised. It is the sum of values of $localValue_i$ $(1 \leq i \leq n)$—see constraint (c3)— where $localValue_i$ is constrained by (c5) to be equal to $logLiklihood_i[parentset_i]$. Constraints (c1) and (c4) bound the $parentset_i$ and $globalValue$ variables, in effect giving each a finite domain. Constraint (c2) imposes an upper bound on the $localValue_i$ variables; the use of a lower bound is an issue discussed below.

Acyclicity is enforced by associating a generation number, the decision variable gen_i, with each individual i. The generation number of an individual is constrained to be one greater than the maximum generation number of its parents. This is stipulated in (c7) if i has two parents and in (c8) if i has one parent. Constraint (c6) bounds each gen_i, giving it a finite domain.

Finally, gender consistency is enforced by associating a gender, the boolean decision variable $female_i$ with each individual. The only constraint is that if an individual has two parents then those parents must have opposite gender. This is stipulated in the last conjunct of (c7).

The basic model imposes used upper and lower bounds on the variables $parentset$ and gen, but only an upper bound on the variable $localValue$. This is because during our experiments we have detected surprising behaviour when we use a lower bound on that variable. Therefore, to study this behaviour, we have defined two variants of constraint (c2):

- (c2$^+$) when in addition to (c2) we also use the tighter lower bound:

$$localValue_i \geq \min_{j \in 1..k_i} logLikelihood_i[j]$$

- (c2^0) when in addition to $c2$ we also use a weaker lower bound:

$$localValue_i > 0$$

In some of the experiments we extended the model to use three additional constraints to represent the domain of the variables:

Given

n:	positive integer	
k_i:	positive integer	$(1 \leq i \leq n)$
$pps_i[1..k_i]$:	set maxsize 2 drawn from $1..n$	$(1 \leq i \leq n)$
$logLikelihood_i[1..k_i]$:	positive integer	$(1 \leq i \leq n)$

Decision Variables

$parentset_i$: int	$(1 \leq i \leq n)$
gen_i: int	$(1 \leq i \leq n)$
$localValue_i$: int	$(1 \leq i \leq n)$
$globalValue$: int	
$female_i$:bool	$(1 \leq i \leq n)$

Constraints

(c1) $1 \leq parentset_i \wedge parentset_i \leq k_i \ (1 \leq i \leq n)$

(c2) $localValue_i \leq \max_{j \in 1..k_i} logLikelihood_i[j] \ (1 \leq i \leq n)$

(c3) $globalValue = \sum_{i=1}^{n} localValue_i$

(c4) $globalValue \geq 0 \wedge globalValue \leq \sum_{i=1}^{n} \max_{j \in 1..k_i} logLikelihood_i[j]$

(c5) not $(parentset_i = j) \vee localValue_i = logLikelihood_i[j] \ (1 \leq i \leq n, 1 \leq j \leq k_i)$

(c6) $0 \leq gen_i \wedge gen_i \leq n \ (1 \leq i \leq n)$

(c7) $(\text{not}(parentset_i = r) \vee gen_i - gen_p \geq 1) \wedge$
$(\text{not}(parentset_i = r) \vee gen_i - gen_{p'} \geq 1) \wedge$
$(\text{not}(parentset_i = r) \vee gen_i - gen_p = 1 \vee gen_i - gen_{p'} = 1) \wedge$
$(\text{not}(parentset_i = r) \vee not(female_p = female_{p'}))$
$(1 \leq i \leq n, 1 \leq j \leq k_i, pps_i[j] = \{p, p'\})$

(c8) not $(parentset_i = r) \vee gen_i - gen_p \geq 1 \ (1 \leq i \leq n, 1 \leq j \leq k_i, pps_i[j] = \{p\})$

Objective

maximize $globalValue$

Table 10.2: Basic SMT(QF_LIA) model of the pedigree reconstruction problem

(c9) DomainALO(parentset): $\bigvee_{j\in 1..k_i} parentset_i = j \ (1 \leq i \leq n)$
(c10) DomainALO(gen): $\bigvee_{j\in 0..n}(gen_i = j) \ (1 \leq i \leq n)$
(c11) DomainALO(lc): $\bigvee_{j\in 1..k_i} localValue_i = logLikelihood_i[j]$

To simplify our study we replace the PRP optimisation problem with the corresponding decision problem. In particular, from each PRP instance we generate the hardest satisfiable instance and the hardest unsatisfiable instance. We precompute the global value, c^*, of the optimal solution and create two benchmark problem instances: a satisfiable instance in which the objective is replaced with the constraint $globalValue \geq c^*$ and an unsatisfiable instance in which the objective is replaced with the constraint $globalValue > c^*$.

Notice that in both the satisfiable and unsatisfiable instances the constraints (c10) and (c11) are implied. To fully appreciate Anomaly 4 it is important to bear in mind that because these are implied constraints they prune no solutions.

All of our experiments use the same suite of 100 PRP instances generated randomly by pedsim [Cow09], configured to use genetically realistic random distributions. Such instances are known to be harder to solve than those generated using random distributions. All of our instances contain 46 individuals.

In all the experiments of Anomalies 3 and 4 we have used the same four SMT(QF_LIA) solvers used in Anomalies 1 and 2. But the experiments were run on a slightly different computer, Intel® Core™ i5 CPU@2.66GHz, with 3GB of RAM, under 32-bit openSUSE 11.2 (kernel 2.6.31).

10.2.1 Anomaly 3

Anomaly 3 arose through a surreptitious observation. Solve time can be reduced if we replace constraint (c3) with

$$(c3') \qquad globalValue2 = \sum_{i=1}^{n} localValue_i \wedge globalValue2 = globalValue$$

where $globalValue$ is a new integer variable.

This anomaly only appears in MathSat 5 and Yices 1 solvers when we are also using (c2$^+$) instead of (c2), and in the case of Yices 1 when we are also using ALO for the variables (constraints (c9), (c10) and (c11)). It appears only in the satisfiable instances and it does not appear in the unsatisfiable ones. Table 10.3 and Table 10.4 show the mean time and median time with and without the extra variable. In both cases the extra variable reduces the median time about 20%.

Table 10.3: Anomaly 3: MathSat 5 solves the 100 instances about 20% faster with an extra variable in the basic model with tighter lower bound.

	Mean time (s)	Median time (s)
non ex.	5.1283	2.085
extra	4.0067	1.56

Table 10.4: Anomaly 3: Yices 1 solves the 100 instances about 20% faster with an extra variable in the basic model with tighter lower bound and ALO.

	Mean time (s)	Median time (s)
non ex.	1.2569	1.005
extra	1.0584	0.82

One would expect that adding this extra variable would have only a trivial affect on solution time. However, the two scatter plots, Figure 10.7 and Figure 10.8, show that the

effect of adding an extra variable can drastically increase or decrease the solving time of an instance. More precisely, in Figure 10.7 when adding the extra variable there is a variability on solving one instance from 52.9 times faster (decreasing the solving time from 35.92 to 0.68) to 32.4 times slower (increasing the solving time from 0.86 to 27.87). In Figure 10.8 this variability is smaller, ranging from 4.6 times faster (decreasing the solving time from 5.1 to 1.11) to 2.0 times slower (increasing the solving time from 1.51 to 3.02).

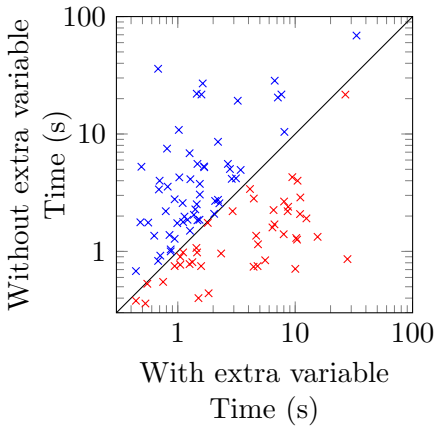

Figure 10.7: Anomaly 3: scatter-plot of MathSat 5 with and without extra variable in the basic model with tighter lower bound. 57 blue crosses and 43 red crosses.

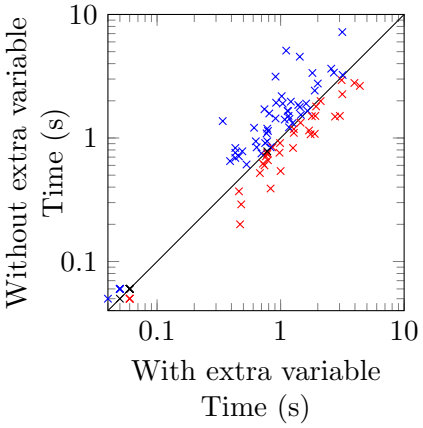

Figure 10.8: Anomaly 3: scatter-plot of Yices 1 with and without extra variable in the basic model with tighter lower bound and ALO. 53 blue crosses, 43 red crosses and 12 black crosses.

10.2.2 Anomaly 4

The last anomaly arises in test to measure in the basic model which way to represent the lower bound is the best for Z3 solver: without any lower bound (c2), with a tighter lower bound ($c2^+$) or with a lighter lower bound ($c2^0$). Again the first impression is that $c2^+$ has to be the best option, but in practise the best option is $c2$, that is not using any lower bound. This is shown at Table 10.5 where we can see the mean and median times of solving the 100 instances for the three experiments. The first two columns are the solving times for the satisfiable instances and the two last columns are the times for the unsatisfiable instances. We can see that solving the basic model with $c2$ is about 5 to 6 times faster than with $c2^+$ or $c2^0$ in the satisfiable case. A similar phenomenon happens with the unsatisfiable instances, where the basic model with $c2$ is about 3 to 4 times faster then the other two representations.

We want to note that this anomaly disappears if we add ALO to represent $localValue$ variable domain (adding constraint $c11$).

Finally, we present four scatter plots comparing $c2$ with $c2^+$ and $c2^0$ in Figure 10.9 and Figure 10.10 respectively for the satisfiable case and in Figure 10.11 and Figure 10.12 respectively for the unsatisfiable case. What is interesting of these plots is that in all the cases $c2$ is better in all the instances (specially for the unsatisfiable case). More precisely, $c2$ is better than $c2^+$ between 1.18 and 18.90 times in the satisfiable case and between 1.38 and 9.46 times better in the unsatisfiable case, and $c2$ is better than $c2^0$ between 1.17 and 22.56 times in the

Table 10.5: Anomaly 4: Z3 solves the satisfiable instances about 5 to 6 times faster in the basic model without any lower bound compared to using any lower bound, and about 3 to 4 times faster for the unsatisfiable instances.

	Satisfiable		Unsatisfiable	
	Mean time (s)	Median time (s)	Mean time (s)	Median time (s)
$c2^+$	29.94	31.73	37.28	35.60
$c2^0$	35.43	34.64	41.47	38.76
$c2$	7.06	5.59	12.61	9.68

satisfiable case and between 1.54 and 9.71 times better in the unsatisfiable case.

10.3 Conclusions

In this chapter we have presented four anomalies in the behaviour of some SMT(QF_LIA) solvers, two of them representing finite domains and the other two representing the PRP. The behaviours of the SMT(QF_LIA) solvers are anomalous in that they contradict our expectations, which have been formed from our experience with modelling problems for CP solvers and modelling problems for SMT solvers. In Anomaly 1 we see that a simple, obvious representation of finite domains is problematic for one SMT(QF_LIA) solver, though not for three others.

Anomaly 2 arises in considering parameterised problem instances in which solving larger instances requires a superset of the reasoning involved in solving smaller instances. Nonetheless, for one solver the solution time is not monotonic in the size of the problem instance.

In Anomaly 3 a trivial change to a model, that would have little to no effect on a CP solver, can greatly reduce the solve time of one instance while greatly increasing that of another instance. Though we would expect any change in performance to be a very small negative one, we see an average *improvement* of 20% with two of the solvers.

Finally, Anomaly 4 is perhaps the most surprising to anyone trying to build better models. Here we see a case where tightening bounds on a variable hinders performance even in unsatisfiable instances.

In seeking an explanation of these anomalies we have contacted with the SMT solver developers and next we present our conclusions to the anomalies from their responses.

The first thing to take into account is that SMT solvers are very complicated tools which involve several components: various forms of preprocessing, a CDCL SAT solver and specific solvers for each background theory (QF_LIA in our case). Since these components are used depending on different heuristics and strategies, the behaviour could vary largely depending on the configuration. Also, it is important to recall that we are using SMT solvers as a black box with the default configurations for solving problems which are very far from these for which SMT solvers are usually tested (and tuned) for.

Concerning anomalies 1 and 2, the developer pointed that is due to a performance bottleneck/bug in how bounds axioms are asserted to Z3. Although anomaly 2, which only appears in the satisfiable instances, could also be because of an heuristic or strategy configuration of the solver.

Concerning variations of anomalies 3 and 4, the developers stated that are well within the variation range due to different choices of Boolean decision variables, in particular on

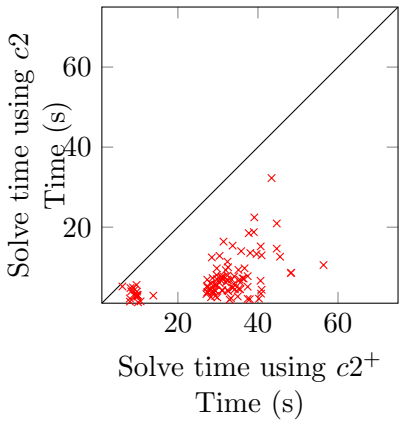

Figure 10.9: Anomaly 4: scatter-plot of Z3 with tighter lower bound ($c2^+$) and without lower bound ($c2$) in the basic model for the satisfiable case.

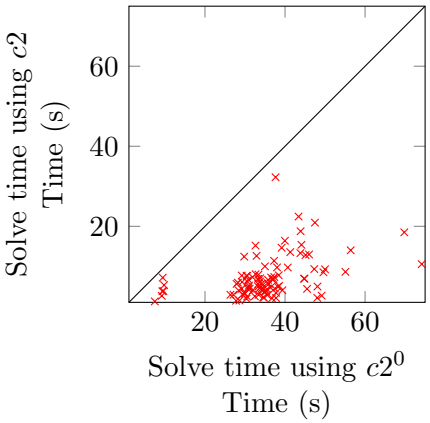

Figure 10.10: Anomaly 4: scatter-plot of Z3 with lower bound greater than 0 ($c2^0$) and without lower bound ($c2$) in the basic model for the satisfiable case.

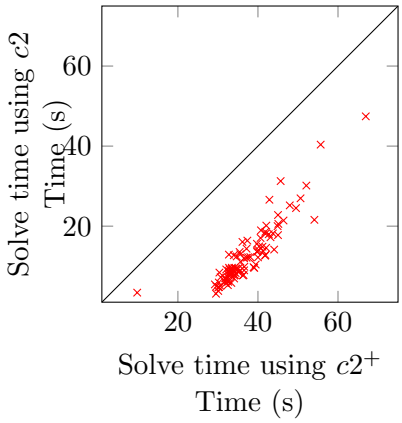

Figure 10.11: Anomaly 4: scatter-plot of Z3 with tighter lower bound ($c2^+$) and without lower bound ($c2$) in the basic model for the unsatisfiable case.

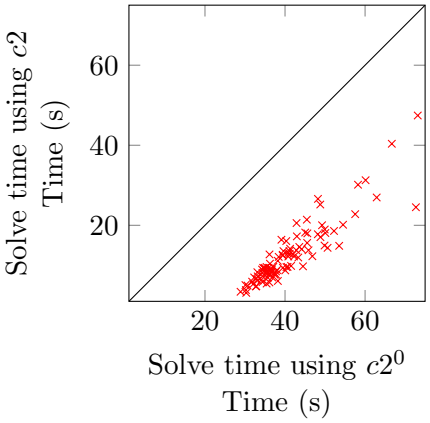

Figure 10.12: Anomaly 4: scatter-plot of Z3 with lower bound greater than 0 ($c2^0$) and without lower bound ($c2$) in the basic model for the satisfiable case.

satisfiable instances. But the case of anomaly 4, which appears in both type of instances (satisfiable and unsatisfiable) and only using the Z3 solver, it could also be because removing the bound we are reducing (or avoiding) the bug/bottleneck of anomaly 1.

Overall, it is clear that modelling combinatorial problems using SMT is more complicated than using CP and more interaction with SMT developers could help to maximize the benefits of SMT for combinatorial problems.

Chapter 11

Conclusions and future work

11.1 Conclusions

During this thesis we have succeeded to achieve the proposed objectives. We have developed two systems, called `fzn2smt` and `WSimply`, that reformulate constraint models into SMT instances in order to be solved using SMT solvers. The former reformulates CSP instances written in the low-level language FlatZinc into SMT instances, while the latter directly reformulates CSP instances written in its own high-level language into SMT instances. Both of them can deal with CSPs and COPs, but `WSimply` can also deal with WCSPs and, furthermore, allows the user to post meta-constraints over the soft constraints. We used both tools to show that SMT is competitive with state-of-the-art CSP and WCSP solvers. In addition we have solved specific problems either by direct reformulation into SMT or using `WSimply`.

`fzn2smt`

First of all, we used `fzn2smt` to see the performance of several SMT solvers. For the experiments we considered 294 instances from 32 different problems written in FlatZinc. We selected the solvers that have had good performance in the QF_LIA division of the SMT-COMP, since is the logic where (almost) all the problems fall. The solver with best performance was Yices 2 nearly followed by Yices 1. Overall, the latter solved one more instance than the former, however, the former got better results in 19 of the 32 problems. Next we figured out which is the best way to encode arrays of variables and related constraints, either by using the uninterpreted functions (UF) theory or decomposing them into single variables. The result was that the decomposition approach clearly outperformed the UF approach. Then we tested which of the three optimization approaches: binary, lineal or hybrid, was better solving the subset of optimization problems, which were 16 of 32. In this experiment we had a draw in the number of instances solved between the binary and the hybrid approaches. However, the former was a bit faster than the latter. Therefore, we set `fzn2smt` default configuration to use Yices 2 with variable decomposition and binary search.

Once we set up the configuration of `fzn2smt`, we used it to compare the performance of an SMT solver (Yices 2) with several other FlatZinc solvers on two different comparisons. In the first one we did not use solver specific global constraints in the FlatZinc models, i.e., global

constraints were decomposed into basic constraints during the reformulation from MiniZinc to FlatZinc. In the second one we used global constraints in the FlatZinc models. Note that in this second comparison we only used the problems containing global constraints in their model and only the FlatZinc solvers which support them, G12 and Gecode. `fzn2smt` does not provide any specialized control for global constraints, hence in the second comparison we used the `fzn2smt` results of the first comparison. In both cases `fzn2smt` got better performance than the other solvers and was nearly followed by Gecode.

Finally, we made an statistical study of the relation between the Boolean structure of the problems and the performance of `fzn2smt` compared to Gecode. For each problem we computed the *Boolean variable ratio* and the *disjunction ratio*, and we compared the number of instances solved by `fzn2smt` and Gecode with respect to these ratios. In almost all the problems, when the Boolean variable ratio was above 0.2, `fzn2smt` was able to solve more instances than Gecode, whereas in the disjunction ratio this happens when it was above 0.4. In conclusion, we can affirm that the performance of `fzn2smt`, and hence SMT solvers, compared to FlatZinc solvers, is directly related to the Boolean structure of the problem. This makes sense since in SMT solvers the search is controlled by the SAT solver.

`WSimply`

First of all, we developed a system, called `Simply`, for modelling CSPs and COPs with its own high-level declarative language. In `Simply`, CSP instances were reformulated into SMT instances and solved using SMT solvers. This way we had a complete control of the reformulation from the high-level language into SMT. Since most SMT solvers did not support optimization, we implemented a binary search algorithm for solving COPs. We compared `Simply`, using Yices 2 as SMT solver, with several FlatZinc solvers on 208 instances from 18 problems, taken from the MiniZinc distribution and properly encoded in `Simply` language. `Simply` was the solver that solved more instances in total. However, it only was the best (solved more instances with less time) in 3 problems. In contrast, Gecode solved fewer instances but it was the best in 8 problems. Finally, we conducted a comparison of both tools, `Simply` with `fzn2smt`, on the same 18 problems. This comparison is interesting because it might give us an idea of which reformulation into SMT is better. On the one hand, `fzn2smt` solved one instance more than `Simply` in 700 seconds less. On the other hand, `Simply` was the best on 12 problems, whereas `fzn2smt` was the best only on 6 problems. Therefore, in general, both tools have similar performance. However, in some problems, SMT instances generated from the reformulation from MiniZinc into FlatZinc are solved faster than SMT instances directly generated from the `Simply` language. Therefore, although `Simply` reformulation into SMT is more natural, in some cases, the MiniZinc to FlatZinc reformulation is useful for SMT. It may be interesting to add to `Simply` some of the preprocessing techniques used during the MiniZinc reformulation into FlatZinc.

Then we extended `Simply` to deal with WCSPs, calling it `WSimply`. On the one hand, the language was extended to allow the user to post soft constraints. On the other hand, we implemented a solving module to solve WCSPs either by reformulating them into SMT (optimization using SMT or WSMT), or by reformulating them into PB (PBO and WBO) or by reformulating them into LP. It worth to remark that `WSimply` uses by default the Yices 1 SMT solver through API, which allowed us to implement an incremental version of the binary search algorithm and to implement an SMT version of the weighted MaxSAT algorithm WMP1.

We also extended the language by adding meta-constraints over the soft constraints. `WSimply` was the first system that allowed the user to model intensional WCSPs and use meta-constraints. Such meta-constraints provide the language a greater expressivity, allowing the user, for instance, to define fairness between groups of soft constraints. In the experimental results we first showed the usefulness of meta-constraints when modelling a variant of the BACP (called SBACP) and the NRP. Then we compared the performance of `WSimply` using PB solvers (`WSimply` (PB)) with `WSimply` using SMT (`WSimply` (SMT)) on the same two problems. `WSimply` (PB) showed slightly better results in the SBACP. In the NRP, the performance, completely depended on the PB solver used, where `WSimply` +SCIP was clearly the best solving approach.

We also added to `WSimply` (SMT) a new incremental optimization algorithm for solving WCSPs. The new method consists in using SAT clauses to represent the bound to the objective function instead of using an arithmetic expression. The idea of the new method was to represent the bound to the objective function using a BDD that is then used to generate the SAT clauses. The novelty of the method was to use a shared BDD to represent the all bounds to the objective function used during the search.

In the experimental results we studied the impact of using shared BDDs. On the one hand, we considerably reduced the BDD construction time thanks to the node reuse. We got an average of 80% of node reuse when the search process was nearly at the halfway. On the other hand, we were also reusing the previously generated SAT clauses, keeping more learning from the previous iterations and hence reducing the solving times. One last advantage shown in the experiments was that representing the bounds by SAT clauses instead of using an arithmetic expression had a direct benefit to the SAT solver search. This may be because the SAT solver directly knowns if the bound constraint is satisfied or not, instead of having to ask it to the theory solver when it has a partial assignment.

In order to see the performance of the SBDD method we made two comparisons on six WCSPs. The first comparison was with the other `WSimply` (SMT) methods, and the second one with state-of-the-art WCSP, COP and LP solvers. Clearly the new SBDD method outperformed the previous `WSimply` solving methods. In fact SBDD had the best results on almost all the problems. In the comparison with state-of-the-art WCSP, COP and LP solvers, the new SBDD method alternate with CPLEX on having the best results.

Finally, we defined an MDD method for linear integer constraints, which is a generalization of the BDD method for PB constraints. In addition, we presented an special case of objective functions where we can use what we call weighted MDDs. We find this type of objective functions in the NRP. In the experimental results on using the WMDD method for the NRP we showed that the new shared WMDD method clearly outperformed the `WSimply` core-based method, which was the only `WSimply` method that was able to solve some instances of the NRP. Although we could not solve any instance using the SBDD method, we compared the size of the shared BDDs with the size of the shared WMDDs. In median, the size of the final shared WMDD was 7.7 times smaller than the size of the final shared BDDs. Which means, that having more branches at each node is enhancing the node reuse in the lower layers.

Modelling problems

We also showed that SMT can be competitive for specific problems. First of all we presented the `rcp2smt` tool to solve the RCPSP. We showed that it was competitive with the state-of-the-art solvers of this problem. `rcp2smt` is an ad hoc tool that reads RCPSP instances and

solves them using Yices 1 SMT solver through API. We implemented several preprocessing steps and adapted the three encodings for SMT. It worth to remark, that the preprocessing steps significantly improved the performance of the tool solving the problem, specially for the Time encoding since they allowed us to considerably reduce the number of constraints and variables.

Another problem that we solved using SMT is the scheduling of B2B meetings. In this case we used `WSimply` to model the problem, and we solved it by reformulating it into SMT, PB and LP. We compared the performance of `WSimply` with some FlatZinc solvers. The experimental results showed that clasp (which is a PB solver) was specially good on solving the small instances, whereas `WSimply` with SBDDs was better on the bigger ones.

Finally, we presented four anomalies found in the behaviour of SMT solvers (using the QF_LIA logic) when modelling the finite domains using SMT and solving the PRP. To explain them we contacted with the SMT solver developers. Although, at the end, three of four anomalies may had been caused by a bug, it seems that the behaviour appearing in our experiments could also be normal depending on the configuration of the solver. What is clear, is that SMT solvers are very complicated tools and, although we can get very good results using them as black boxes, it would be better to work closer to SMT solver developers to achieve better results on specific problems.

11.2 Future Work

Next we present some of the possible steps to follow from this thesis.

We believe that there is much room for improvement in solving CSPs with SMT, although we have had very good results using SMT solvers as black boxes. For instance, as pointed out by de Moura in [dM11], the efficiency of a solver is strongly dependent on its predefined strategies. Hence changing these heuristics could dramatically affect the SMT solver performance in solving CSPs. In addition, we think that developing theory solvers for global constraints is also a promising research line. In fact, there exist already some promising results in this direction, for instance, for the `alldifferent` theory [BM10].

We think that there are still several things that we can do to improve our `WSimply` tool. First of all, we think that the reformulation from `WSimply` language into SMT might be improved. We can find some clue to do this by analysing the SMT formula resulting from the reformulation of `fzn2smt` on the six problems where it has had better results than `WSimply`.

We also want to use meta-constraints in more problems and guarantee the support of all meta-constraints in all the reformulations. In addition, we think that finding new meta-constraints is also an interesting goal to increase the expressivity of the language.

Another possible improvement may be to reformulate the constraints using other theories and logics for the SMT solvers, since at the moment `WSimply` only works with the QF_LIA logic. This change could simplify another extension of `WSimply` we have in mind, that is to allow non lineal weights in the soft constraints, which could be done, for instance, by using non linear arithmetic theories.

Concerning the new BDD-based methods, we want to use them to solve more WCSPs and COPs. We are also interested in finding a variable ordering that maximizes the node reuse through iterations, although it is known that the problem of finding the optimal variable ordering in order to generate a minimal BDD is NP-hard. We also want to compare them with the ones recently proposed in [AS14], or even it could be interesting to see if we can

find better results by merging both methods. Another interesting thing is to find other places where we can use BDD-based encodings, or even find other special cases of constraints, either in objective functions or in normal constraints, where we can apply better techniques like WMDD presented for the NRP. For instance, we are planning to use in the `rcp2smt` tool the BDD version of the *at-most-k* implemented inside `WSimply`.

Finally, since we have obtained very good results using SMT solvers in scheduling problems, we also plan to deal with other variants of such type of problems.

Bibliography

[ABL09] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy, *Solving (Weighted) Partial MaxSAT through Satisfiability Testing*, Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT), LNCS, vol. 5584, 2009, pp. 427–440.

[ABP+11a] Carlos Ansótegui, Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *Satisfiability Modulo Theories: An Efficient Approach for the Resource-Constrained Project Scheduling Problem*, Proceedings of the 9th Symposium on Abstraction, Reformulation, and Approximation (SARA), 2011, pp. 2–9.

[ABP+11b] Carlos Ansótegui, Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *W-MiniZinc: A Proposal for Modeling Weighted CSPs with MiniZinc*, Proceedings of the 1st Minizinc workshop (MZN), 2011.

[ABP+13] Carlos Ansótegui, Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *Solving weighted csps with meta-constraints by reformulation into satisfiability modulo theories*, Constraints **18** (2013), no. 2, 236–268.

[Ach09] Tobias Achterberg, *SCIP: solving constraint integer programs*, Mathematical Programming Computation **1** (2009), no. 1, 1–41.

[Ack54] W. Ackermann, *Solvable cases of the decision problem*, Studies in logic and the foundations of mathematics, North-Holland, 1954.

[AM04] Carlos Ansótegui and Felip Manyà, *Mapping Problems with Finite-Domain Variables to Problems with Boolean Variables*, Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT), LNCS, vol. 3542, 2004, pp. 1–15.

[AMR03] Christian Artigues, Philippe Michelon, and Stephane Reusser, *Insertion Techniques for Static and Dynamic Resource-Constrained Project Scheduling*, European Journal of Operational Research **149** (2003), no. 2, 249–267.

[ANO+12] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Valentin Mayer-Eichberger, *A New Look at BDDs for Pseudo-Boolean Constraints*, Journal of Artificial Intelligence Research (JAIR) **45** (2012), 443–480.

[AS12] Ignasi Abío and Peter J. Stuckey, *Conflict directed lazy decomposition*, Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP), LNCS, 2012, pp. 70–85.

[AS14] Ignasi Abío and Peter J. Stuckey, *Encoding linear constraints into SAT*, Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP), LNCS, vol. 8656, 2014, pp. 75–91.

[AW07] Krzysztof R. Apt and Mark Wallace, *Constraint Logic Programming using Eclipse*, Cambridge University Press, New York, NY, USA, 2007.

[b2m] `http://www.b2match.com`, Accessed 11 Apr 2014.

[Bap09] Philippe Baptiste, *Constraint-Based Schedulers, Do They Really Work?*, Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP), LNCS, vol. 5732, 2009, pp. 1–1.

[Bar05] Roman Barták, *Constraint propagation and backtracking-based search*, 1st International summer school on CP, Maratea, Italy, 2005.

[BB03] Olivier Bailleux and Yacine Boufkhad, *Efficient CNF Encoding of Boolean Cardinality Constraints*, In Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP), LNCS, vol. 2833, 2003, pp. 108–122.

[BBC+06] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi A. Junttila, Silvio Ranise, Peter van Rossum, and Roberto Sebastiani, *Efficient Theory Combination via Boolean Search*, Information and Computation **204** (2006), no. 10, 1493–1525.

[BCBL04] Edmund K. Burke, Patrick De Causmaecker, Greet Vanden Berghe, and Hendrik Van Landeghem, *The State of the Art of Nurse Rostering*, Journal of Scheduling **7** (2004), no. 6, 441–499.

[BCF+06] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Alessandro Santuari, and Roberto Sebastiani, *To Ackermann-ize or Not to Ackermann-ize? On Efficiently Handling Uninterpreted Function Symbols in SMT($\mathcal{EUF} \cup \mathcal{T}$)*, Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), LNCS, vol. 4246, 2006, pp. 557–571.

[BCF+08] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani, *The MathSAT 4 SMT Solver*, Proceedings of the 20th International Conference Computer Aided Verification (CAV), LNCS, vol. 5123, 2008, pp. 299–303.

[BDM+99] Peter Brucker, Andreas Drexl, Rolf Möhring, Klaus Neumann, and Erwin Pesch, *Resource-Constrained Project Scheduling: Notation, Classification, Models, and Methods*, European Journal of Operational Research **112** (1999), no. 1, 3 – 41.

[BEG+14] Miquel Bofill, Joan Espasa, Marc Garcia, Miquel Palahí, Josep Suy, and Mateu Villaret, *Scheduling B2B meetings*, Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP), LNCS, vol. 8656, 2014, pp. 781–796.

[BEPV12] Miquel Bofill, Joan Espasa, Miquel Palahí, and Mateu Villaret, *An extension to simply for solving weighted constraint satisfaction problems with pseudo-boolean constraints*, 12th Spanish Conference on Programming and Computer Languages (PROLE), 2012, pp. 141–155.

[Ber93] Pierre Berlandier, *The use and interpretation of meta level constaints*, Proceedings of the 6th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence (EPIA), LNCS, vol. 727, 1993, pp. 271–280.

[Bes06] Christian Bessiere, *Constraint propagation*, Tech. Report LIRMM 06020, CNRS/University of Montpellier, 2006.

[BFM97] Thomas Back, David B. Fogel, and Zbigniew Michalewicz (eds.), *Handbook of Evolutionary Computation*, 1st ed., IOP Publishing Ltd., 1997.

[BGS99] Laure Brisoux, Éric Grégoire, and Lakhdar Sais, *Improving Backtrack Search for SAT by Means of Redundancy*, Proceedings of the 11th International Symposium on Foundations of Intelligent Systems (ISMIS), LNCS, vol. 1609, 1999, pp. 301–309.

[BGSV15] Miquel Bofill, Marc Garcia, Josep Suy, and Mateu Villaret, *Maxsat-based scheduling of B2B meetings*, Proceedings of the 12th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR), LNCS, vol. 9075, 2015, pp. 65–73.

[BH02] Endre Boros and Peter L. Hammer, *Pseudo-Boolean optimization*, Discrete Applied Mathematics **123** (2002), no. 1-3, 155–225.

[BHM01] Ramón Béjar, Reiner Hähnle, and Felip Manyà, *A Modular Reduction of Regular Logic to Classical Logic*, Proceedings of the 31st IEEE International Symposium on Multiple-Valued Logic (ISMVL), 2001, pp. 221–226.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (eds.), *Handbook of Satisfiability*, Frontiers in Artificial Intelligence and Applications, vol. 185, IOS Press, 2009.

[BKO+07] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan A. Brady, *Deciding Bit-Vector Arithmetic with Abstraction*, Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 4424, 2007, pp. 358–372.

[BLK83] J. Blazewicz, J. K. Lenstra, and A. H. G. Rinnooy Kan, *Scheduling Subject to Resource Constraints: Classification and Complexity*, Discrete Applied Mathematics **5** (1983), no. 1, 11 – 24.

[BLO+12] Cristina Borralleras, Salvador Lucas, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio, *SAT Modulo Linear Arithmetic for Solving Polynomial Constraints*, Journal of Automated Reasoning **48** (2012), no. 1, 107–131.

[BM10] Milan Bankovic and Filip Maric, *An Alldifferent Constraint Solver in SMT*, Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (SMT), 2010.

[BMR88] M. Bartusch, R. H. Mohring, and F. J. Radermacher, *Scheduling Project Networks with Resource Constraints and Time Windows*, Annals of Operations Research **16** (1988), 201–240.

[BMR97] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi, *Semiring-based constraint satisfaction and optimization*, Journal of the ACM **44** (1997), no. 2, 201–236.

[BMR+99] S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, and H. Fargier, *Semiring-Based CSPs and valued CSPs: Frameworks, Properties, and Comparison*, Constraints **4** (1999), no. 3, 199–240.

[BNO+08a] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio, *A Write-Based Solver for SAT Modulo the Theory of Arrays*, Proceedings of the 8th Conference on Formal Methods in Computer-Aided Design (FMCAD), 2008, pp. 1–8.

[BNO+08b] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio, *The Barcelogic SMT Solver*, Proceedings of the 20th International Conference Computer Aided Verification (CAV), LNCS, vol. 5123, 2008, pp. 294–298.

[BPF15] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein, *νz - an optimizing SMT solver*, Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 9035, 2015, pp. 194–199.

[BPSV09] Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *SIMPLY: a Compiler from a CSP Modeling Language to the SMT-LIB Format*, Proceedings of the 8th International Workshop on Constraint Modelling and Reformulation (ModRef), 2009, pp. 30–44.

[BPSV12] Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *Solving constraint satisfaction problems with SAT modulo theories*, Constraints **17** (2012), no. 3, 273–303.

[BPSV14] Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *Solving intensional weighted csps by incremental optimization with bdds*, Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP), LNCS, vol. 8656, 2014, pp. 207–223.

[BS94] Belaid Benhamou and Lakhdar Sais, *Tractability Through Symmetries in Propositional Calculus*, J. Autom. Reasoning **12** (1994), no. 1, 89–102.

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli, *Satisfiability Modulo Theories*, Handbook of Satisfiability, vol. 185, IOS Press, 2009, pp. 825–885.

[BST10a] Clark Barrett, Aaron Stump, and Cesare Tinelli, *The Satisfiability Modulo Theories Library (SMT-LIB)*, http://www.SMT-LIB.org, 2010.

[BST10b] Clark Barrett, Aaron Stump, and Cesare Tinelli, *The SMT-LIB Standard: Version 2.0*, Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (SMT), 2010.

[CB94] James M. Crawford and Andrew B. Baker, *Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems*, In Proceedings of the 12th National Conference on Artificial Intelligence (AAAI), 1994, pp. 1092–1097.

[CBJS13] James Cussens, Mark Bartlett, Elinor M. Jones, and Nuala A. Sheehan, *Maximum Likelihood Pedigree Reconstruction Using Integer Linear Programming*, Genetic Epidemiology **37** (2013), no. 1, 69–83.

[CFG+10] Alessandro Cimatti, Anders Franzén, Alberto Griggio, Roberto Sebastiani, and Cristian Stenico, *Satisfiability Modulo the Theory of Costs: Foundations and Applications*, Proceedings of the 16h International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 6015, 2010, pp. 99–113.

[CGSS13a] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani, *The MathSAT5 SMT solver*, Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, 2013, pp. 93–107.

[CGSS13b] ———, *A modular approach to maxsat modulo theories*, Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT), LNCS, vol. 7962, 2013, pp. 150–165.

[CM01] Carlos Castro and Sebastian Manzano, *Variable and Value Ordering When Solving Balanced Academic Curriculum Problems*, Proceedings of the 6th Annual Workshop of the ERCIM Working Group on Constraints, 2001.

[CN03] Jacques Carlier and Emmanuel Néron, *On Linear Lower Bounds for the Resource Constrained Project Scheduling Problem*, European Journal of Operational Research **149** (2003), no. 2, 314–324.

[Coo03] Martin C. Cooper, *Reduction operations in fuzzy or valued constraint satisfaction*, Fuzzy Sets and Systems **134** (2003), no. 3, 311–342.

[Cow09] Robert G. Cowell, *Efficient maximum likelihood pedigree reconstruction*, Theoretical Population Biology **76** (2009), no. 4, 285–291.

[DdM06a] Bruno Dutertre and Leonardo Mendonça de Moura, *A Fast Linear-Arithmetic Solver for DPLL(T)*, Proceedings of the 18th International Conference Computer Aided Verification (CAV), LNCS, vol. 4144, 2006, pp. 81–94.

[DdM06b] Bruno Dutertre and Leonardo Mendonça de Moura, *The Yices SMT solver*, Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[DF98] Rina Dechter and Daniel Frost, *Backtracking algorithms for constraint satisfaction problems - a tutorial survey*, Tech. report, University of California, 1998.

[DHN06] Nachum Dershowitz, Ziyad Hanna, and Er Nadel, *A scalable algorithm for minimal unsatisfiable core extraction*, Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT), LNCS, vol. 4121, 2006, pp. 36–41.

[dK89] Johan de Kleer, *A comparison of ATMS and CSP techniques*, Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI), 1989, pp. 290–296.

[DLL62] Martin Davis, George Logemann, and Donald Loveland, *A machine program for theorem-proving*, Communications of the ACM **5** (1962), no. 7, 394–397.

[dM11] Leonardo Mendonça de Moura, *Orchestrating Satisfiability Engines*, Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP), LNCS, vol. 6876, 2011, p. 1.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner, *Z3: An Efficient SMT Solver*, Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 4963, 2008, pp. 337–340.

[dMR04] L. de Moura and H. Ruess, *An Experimental Evaluation of Ground Decision Procedures*, Proceedings of the 16th International Conference Computer Aided Verification (CAV), LNCS, vol. 3114, 2004, pp. 162–174.

[DP60] Martin Davis and Hilary Putnam, *A Computing Procedure for Quantification Theory*, Journal of the ACM **7** (1960), 201–215.

[DPBC93] Olivier Dubois, André Pascal, Yacine Boufkhad, and Jaques Carlier, *Can a very simple algorithm be eficient for solving SAT problem?*, Proceedings of the DIMACS Challenge II Workshop, 1993.

[DPPH00] U. Dorndorf, E. Pesch, and T. Phan-Huy, *A Branch-and-Bound Algorithm for the Resource-Constrained Project Scheduling Problem*, Mathematical Methods of Operations Research **52** (2000), 413–439.

[DV07] Dieter Debels and Mario Vanhoucke, *A Decomposition-Based Genetic Algorithm for the Resource-Constrained Project-Scheduling Problem*, Operations Research **55** (2007), no. 3, 457–469.

[EIS75] Shimon Even, Alon Itai, and Adi Shamir, *On the complexity of time table and multi-commodity flow problems*, Proceedings of the 16th Annual Symposium on Foundations of Computer Science (FOCS), 1975, pp. 184–193.

[ES06] Niklas Eén and Niklas Sörensson, *Translating Pseudo-Boolean Constraints into SAT*, Journal on Satisfiability, Boolean Modeling and Computation (JSAT) **2** (2006), no. 1-4, 1–26.

[FM06] Zhaohui Fu and Sharad Malik, *On Solving the Partial MAX-SAT Problem*, Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT), LNCS, vol. 4121, 2006, pp. 252–265.

[FP01] Alan M. Frisch and Timothy J. Peugniez, *Solving Non-Boolean Satisfiability Problems with Stochastic Local Search*, Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI), 2001, pp. 282–290.

[FP14] Alan M. Frisch and Miquel Palahí, *Anomalies in smt solving, difficulties in modelling combinatorial problems*, Proceedings of the 13th International Workshop on Constraint Modelling and Reformulation (ModRef), 2014, pp. 97–110.

[Fre95] Jon William Freeman, *Improvements to propositional satisfiability search algorithms*, Ph.D. thesis, University of Pennsylvania, 1995, UMI Order No. GAX95-32175.

[FS09] Thibaut Feydy and Peter J. Stuckey, *Lazy Clause Generation Reengineered*, 15th International Conference on Principles and Practice of Constraint Programming (CP), LNCS, vol. 5732, 2009, pp. 352–366.

[G1210] *Minizinc + Flatzinc*, http://www.g12.csse.unimelb.edu.au/minizinc/, 2010.

[Gas79] J. Gaschnig, *Performance Measurement and Analysis of Certain Search Algorithms*, Ph.D. thesis, Carnegie-Mellon University, 1979.

[GGSS13] Martin Gebser, Thomas Glase, Orkunt Sabuncu, and Torsten Schaub, *Matchmaking with Answer Set Programming*, Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), LNCS, vol. 8148, 2013, pp. 342–347.

[GKNS07] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub, *clasp : A Conflict-Driven Answer Set Solver*, Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), LNCS, vol. 4483, 2007, pp. 260–265.

[Glo89] Fred Glover, *Tabu Search - Part I*, INFORMS Journal on Computing **1** (1989), no. 3, 190–206.

[Glo90] Fred Glover, *Tabu Search - Part II*, INFORMS Journal on Computing **2** (1990), no. 1, 4–32.

[GN04] Ian P Gent and Peter Nightingale, *A new encoding of alldifferent into sat*, Proceedings of the 3rd International Workshop on Constraint Modelling and Reformulation (ModRef), 2004, pp. 95–110.

[Gom58] R. E. Gomory, *Outline of an Algorithm for Integer Solutions to Linear Programs*, Bulletin of the American Society **64** (1958), 275–278.

[GW99] Ian P. Gent and Toby Walsh, *CSP_{LIB}: A benchmark library for constraints*, Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP), LNCS, vol. 1713, 1999, pp. 480–481.

[HB10] Sönke Hartmann and Dirk Briskorn, *A Survey of Variants and Extensions of the Resource-Constrained Project Scheduling Problem*, European Journal of Operational Research **207** (2010), no. 1, 1 – 14.

[HE80] R. Haralick and G. Elliot, *Increasing Tree Search Efficiency for Constraint Satisfaction Problems*, Artificial Intelligence **14** (1980), no. 3, 263–313.

[HKW02] Brahim Hnich, Zeynep Kiziltan, and Toby Walsh, *Modelling a Balanced Academic Curriculum Problem*, Proceedings of the 4th International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR), 2002, pp. 121–131.

[HL05] Willy Herroelen and Roel Leus, *Project Scheduling under Uncertainty: Survey and Research Potentials*, European Journal of Operational Research **165** (2005), no. 2, 289–306.

[Hoo99] Holger H. Hoos, *Sat-encodings, search space structure, and local search performance*, Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI), 1999, pp. 296–303.

[HOO10] Emmanuel Hebrard, Eoin O'Mahony, and Barry O'Sullivan, *Constraint Programming and Combinatorial Optimisation in Numberjack*, Proceedings of the 7th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR), LNCS, vol. 6140, 2010, pp. 181–185.

[Hor10] Andrei Horbach, *A Boolean Satisfiability Approach to the Resource-Constrained Project Scheduling Problem*, Annals of Operations Research **181** (2010), 89–107.

[HRD98] Willy Herroelen, Bert De Reyck, and Erik Demeulemeester, *Resource-Constrained Project Scheduling: A Survey of Recent Developments*, Computers and Operations Research **25** (1998), no. 4, 279 – 302.

[Hua07] Jinbo Huang, *The Effect of Restarts on the Efficiency of Clause Learning*, Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI), 2007, pp. 2318–2323.

[IM94] Kazuo Iwama and Shuichi Miyazaki, *SAT-Variable Complexity of Hard Combinatorial Problems*, Proceedings of the world computer congres of the IFIP, 1994, pp. 253–258.

[iMIY90] Shin ichi Minato, Nagisa Ishiura, and Shuzo Yajima, *Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean function Manipulation*, Proceedings of the 27th ACM/IEEE Conference on Design Automation (DAC), 1990, pp. 52–57.

[Jac10] *JaCoP Java Constraint Programming Solver*, http://jacop.osolpro.com, 2010.

[JW90] Robert G. Jeroslow and Jinchang Wang, *Solving Propositional Satisfiability Problems*, Annals of Mathematics and Artificial Intelligence **1** (1990), 167–187.

[KALM11] Oumar Koné, Christian Artigues, Pierre Lopez, and Marcel Mongeau, *Event-Based MILP Models for Resource-Constrained Project Scheduling Problems*, Computers and Operations Research **38** (2011), 3–13.

[KP01] R. Kolisch and R. Padman, *An Integrated Survey of Deterministic Project Scheduling*, Omega **29** (2001), no. 3, 249–272.

[KS97] Rainer Kolisch and Arno Sprecher, *PSPLIB - A Project Scheduling Problem Library*, European Journal of Operational Research **96** (1997), no. 1, 205–216.

[KS99] Robert Klein and Armin Scholl, *Computing Lower Bounds by Destructive Improvement: An Application to Resource-Constrained Project Scheduling*, European Journal of Operational Research **112** (1999), no. 2, 322–346.

[Kum92] Vipin Kumar, *Algorithms for Constraint Satisfaction Problems: A Survey*, AI Magazine **13** (1992), no. 1, 32–44.

[Lab05] Philippe Laborie, *Complete MCS-Based Search: Application to Resource Constrained Project Scheduling*, Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI), 2005, pp. 181–186.

[LM08] Olivier Liess and Philippe Michelon, *A Constraint Programming Approach for the Resource-Constrained Project Scheduling Problem*, Annals of Operations Research **157** (2008), 25–36.

[LMS05] Inês Lynce and João Marques-Silva, *Efficient Data Structures for Backtrack Search SAT Solvers*, Annals of Mathematics and Artificial Intelligence **43** (2005), no. 1-4, 137–152.

[LS04] Javier Larrosa and Thomas Schiex, *Solving weighted CSP by maintaining arc consistency*, Artificial Intelligence **159** (2004), no. 1-2, 1–26.

[Mac77] Alan Mackworth, *Consistency in Networks of Relations*, Artificial Intelligence **8** (1977), no. 1, 99–118.

[MH86] Roger Mohr and Thomas C. Henderson, *Arc and Path Consistency Revisited*, Artificial Intelligence **28** (1986), no. 2, 225–233.

[MML11] R. Martins, V. Manquinho, and I. Lynce, *Exploiting cardinality encodings in parallel maximum satisfiability*, Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI), 2011, pp. 313–320.

[MMRB98] Aristide Mingozzi, Vittorio Maniezzo, Salvatore Ricciardelli, and Lucio Bianco, *An Exact Algorithm for the Resource-Constrained Project Scheduling Problem Based on a New Mathematical Formulation*, Management Science **44** (1998), 714–729.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik, *Chaff: Engineering an Efficient SAT Solver*, Proceedings of the 38th ACM/IEEE Conference on Design Automation (DAC), 2001, pp. 530–535.

[Mon74] Ugo Montanari, *Networks of constraints: Fundamental properties and applications to picture processing*, Information Sciences **7** (1974), 95–132.

[MS01] Pedro Meseguer and Martí Sánchez, *Specializing Russian Doll Search*, In Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP), LNCS, vol. 2239, 2001, pp. 464–478.

[MSP09] Vasco M. Manquinho, João P. Marques Silva, and Jordi Planes, *Algorithms for Weighted Boolean Optimization*, Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT), LNCS, vol. 5584, 2009, pp. 495–508.

[MTW+99] Patrick Mills, Edward Tsang, Richard Williams, John Ford, James Borrett, and Wivenhoe Park, *EaCL 1.5: An Easy Constraint optimisation Programming Language*, Tech. Report CSM-324, University of Essex, Colchester, U.K., 1999.

[Nic07] Nicholas Nethercote and Peter J. Stuckey and Ralph Becket and Sebastian Brand and Gregory J. Duck and Guido Tack, *MiniZinc: Towards a Standard CP Modelling Language*, In Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP), LNCS, vol. 4741, 2007, pp. 529–543.

[NO79] G. Nelson and D. C. Oppen, *Simplification by Cooperating Decision Procedures*, ACM Transactions on Programming Languages and Systems, TOPLAS **1** (1979), no. 2, 245–257.

[NO80] Greg Nelson and Derek C. Oppen, *Fast decision procedures based on congruence closure*, Journal of the ACM **27** (1980), no. 2, 356–364.

[NO05] Robert Nieuwenhuis and Albert Oliveras, *DPLL(T) with exhaustive theory propagation and its application to difference logic*, Proceedings of the 17th International Conference Computer Aided Verification (CAV), LNCS, vol. 3576, 2005, pp. 321–334.

[NO06] Robert Nieuwenhuis and Albert Oliveras, *On SAT Modulo Theories and Optimization Problems*, Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT), LNCS, vol. 4121, 2006, pp. 156–169.

[OEK99] A. O. El-Kholy, *Resource Feasibility in Planning*, Ph.D. thesis, Imperial College, University of London, 1999.

[opt] *http://www.opturion.com*, Accessed 11 Apr 2014.

[OSC07] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish, *Propagation = lazy clause generation*, 13th International Conference on Principles and Practice of Constraint Programming (CP), LNCS, vol. 4741, 2007, pp. 544–558.

[OSC09] ———, *Propagation via lazy clause generation*, Constraints **14** (2009), no. 3, 357–391.

[PR91] Manfred Padberg and Giovanni Rinaldi, *A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems*, SIAM review **33** (1991), no. 1, 60–100.

[PRB00] T. Petit, J. C. Regin, and C. Bessiere, *Meta-constraints on violations for over constrained problems*, Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI), 2000, pp. 358–365.

[Pre93] Daniele Pretolani, *Efficiency and stability of hypergraph SAT algorithms*, Proceedings of the DIMACS Challenge II Workshop, 1993.

[PRR15] Gilles Pesant, Gregory Rix, and Louis-Martin Rousseau, *A comparative study of mip and cp formulations for the b2b scheduling optimization problem*, Proceedings of the 12th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR), LNCS, vol. 9075, 2015, pp. 306–321.

[PW96] Lawrence J. Watters Pritsker, A. Alan B. and Philip S. Wolfe, *Multiproject Scheduling with Limited Resources: A Zero-One Programming Approach*, Management Science **16** (1996), 93–108.

[RBW06] Francesca Rossi, Peter van Beek, and Toby Walsh, *Handbook of constraint programming (foundations of artificial intelligence)*, Elsevier Science Inc., 2006.

[Rég94] Jean-Charles Régin, *A filtering algorithm for constraints of difference in csps*, Proceedings of the 12th National Conference on Artificial Intelligence, vol. 1, 1994, pp. 362–367.

[RL09] Olivier Roussel and Christophe Lecoutre, *XML Representation of Constraint Networks: Format XCSP 2.1*, CoRR **abs/0902.2362** (2009), 1–47.

[RS08] Vadim Ryvchin and Ofer Strichman, *Local restarts*, Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT), LNCS, vol. 4996, 2008, pp. 271–276.

[Sat12] *Sat4j pseudo-boolean competition implementation*, `http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf`, 2009 (accessed 02/05/2012).

[SBDL01] A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt, *A Decision Procedure for an Extensional Theory of Arrays*, Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS), 2001, pp. 29–37.

[SBF10] Peter J. Stuckey, Ralph Becket, and Julien Fischer, *Philosophy of the MiniZinc challenge*, Constraints **15** (2010), 307–316.

[Sci10] *SCIP, Solving constraint integer programs*, http://scip.zib.de/scip.shtml, 2010.

[Seb07] Roberto Sebastiani, *Lazy Satisability Modulo Theories*, Journal on Satisfiability, Boolean Modeling and Computation **3** (2007), no. 3-4, 141–224.

[SFS13] Andreas Schutt, Thibaut Feydy, and Peter J. Stuckey, *Explaining time-table-edge-finding propagation for the cumulative resource constraint*, Proceedings of the 10th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR), LNCS, vol. 7874, 2013, pp. 234–250.

[SFSW09] Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark Wallace, *Why Cumulative Decomposition Is Not as Bad as It Sounds*, Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP), LNCS, vol. 5732, 2009, pp. 746–761.

[SFSW10] Andreas Schutt, Thibaut Feydy, Peter Stuckey, and Mark Wallace, *Explaining the Cumulative Propagator*, Constraints (2010), 1–33.

[Sho84] Robert E. Shostak, *Deciding Combinations of Theories*, Journal of the ACM **31** (1984), no. 1, 1–12.

[Sic10] *SICStus Prolog*, http://www.sics.se/sisctus, 2010.

[Sil99] João P. Marques Silva, *The Impact of Branching Heuristics in Propositional Satisfiability Algorithms*, Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence (EPIA), LNCS, vol. 1695, 1999, pp. 62–74.

[Sin05] Carsten Sinz, *Towards an optimal cnf encoding of boolean cardinality constraints*, Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP), 2005, pp. 827–831.

[SLT10] C. Schulte, M. Lagerkvist, and G. Tack, *Gecode*, http://www.gecode.org, 2010.

[SS06] Hossein M. Sheini and Karem A. Sakallah, *From Propositional Satisfiability to Satisfiability Modulo Theories*, Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT), LNCS, vol. 4121, 2006, pp. 1–9.

[TTKB09] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara, *Compiling Finite Linear CSP into SAT*, Constraints **14** (2009), no. 2, 254–272.

[VLS96] Gérard Verfaillie, Michel Lemaître, and Thomas Schiex, *Russian Doll Search for Solving Constraint Optimization Problems*, Proceedings of the 13th National Conference on Artificial Intelligence and 8th Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI), vol. 1, 1996, pp. 181–187.

[VM07] M Vanhoucke and B Maenhout, *NSPLIB - a nurse scheduling problem library: A tool to evaluate (meta-) heuristic procedures*, Operational Research for Health Policy Making Better Decisions (2007), 151–165.

[WHFP95] G. Weil, K. Heus, P. Francois, and M. Poujade, *Constraint programming for nurse scheduling*, Engineering in Medicine and Biology Magazine, IEEE **14** (1995), no. 4, 417–422.

[Wil99] H. P. Williams, *Model Building in Mathematical Programming*, 4th ed., Wiley, 1999.

[Zha97] Hantao Zhang, *SATO: An Efficient Propositional Prover*, In Proceedings of the 14th International Conference on Automated Deduction (CADE), LNCS, vol. 1249, 1997, pp. 272–275.

[ZLS06] Jianmin Zhang, Sikun Li, and Shengyu Shen, *Extracting minimum unsatisfiable cores with a greedy genetic algorithm*, Proceedings of the 19th Australian joint conference on Artificial Intelligence: advances in Artificial Intelligence (AI), LNCS, 2006, pp. 847–856.

[ZM02] Lintao Zhang and Sharad Malik, *The Quest for Efficient Boolean Satisfiability Solvers*, Proceedings of the 14th International Conference Computer Aided Verification (CAV), LNCS, vol. 2404, 2002, pp. 17–36.

[ZM03] Lintao Zhang and Sharad Malik, *Cache Performance of SAT Solvers: a Case Study for Efficient Implementation of Algorithms*, Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT), LNCS, vol. 2919, 2003, pp. 287–298.