RIGOROUS SOLUTION TECHNIQUES FOR NUMERICAL CONSTRAINT SATISFACTION PROBLEMS

THÈSE N^{O} 3155 (2005)

PRÉSENTÉE À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Institut d'informatique fondamentale

SECTION D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDERALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Xuan-Ha VU

Engineer in informatics, Hanoi University of Technology, Viet-Nam
et de nationalité vietnamienne

acceptée sur proposition du jury:

Dr J. Sam, directrice de thèse
Prof. B. Faltings, rapporteur
Prof. A. Neumaier, rapporteur
Prof. M. Rueher, rapporteur
Prof. M. Shokrollahi, rapporteur

Lausanne, EPFL
2005

RIGOROUS SOLUTION TECHNIQUES FOR NUMERICAL CONSTRAINT SATISFACTION PROBLEMS

THESIS N° 3155 (2005)

SUBMITTED TO THE SCHOOL OF COMPUTER AND COMMUNICATION SCIENCES

Institute of Core Computing Science

DEPARTMENT OF COMPUTER SCIENCE

SWISS FEDERAL INSTITUTE OF TECHNOLOGY IN LAUSANNE

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

BY

Xuan-Ha VU

Engineer in Computer Science, Hanoi University of Technology, Vietnam

Vietnamese nationality

Accepted on the recommendation of the thesis committee:

Dr. Jamila Sam, *advisor*
Prof. Dr. Boi Faltings, *examiner*
Prof. Dr. Arnold Neumaier, *examiner*
Prof. Dr. Michel Rueher, *examiner*
Prof. Dr. Mohammad Amin Shokrollahi, *examiner*

Lausanne, EPFL
2005

Contents

Abstract **vii**

Résumé **ix**

Acknowledgements **xi**

List of Algorithms **xiii**

List of Definitions **xv**

List of Figures **xix**

List of Tables **xxi**

1 Introduction **1**
- 1.1 A Preview of Constraint Programming . . . 1
- 1.2 The Goal of the Thesis . . . 3
- 1.3 Contributions of the Thesis . . . 5
- 1.4 Organization of the Thesis . . . 6

2 Background and Definition **9**
- 2.1 Basic Concepts in Constraint Programming . . . 9
  - 2.1.1 A Short History of Constraint Programming . . . 9
  - 2.1.2 Constraint Satisfaction . . . 10
    - 2.1.2.1 Constraint Satisfaction Problems . . . 10
    - 2.1.2.2 Set Theory Concepts for Constraints . . . 15
    - 2.1.2.3 Basic Concepts of Problem Solving . . . 17
    - 2.1.2.4 Major Solution Approaches . . . 22
  - 2.1.3 Numerical Constraint Satisfaction Problems . . . 24
    - 2.1.3.1 Numerical Constraints . . . 24
    - 2.1.3.2 Problem Formulation . . . 25
    - 2.1.3.3 Factorable Form . . . 27
    - 2.1.3.4 Separable Form . . . 29
    - 2.1.3.5 Ternary Form . . . 30
- 2.2 Common Arithmetics for Numerical Computations . . . 30
  - 2.2.1 Floating-Point Numbers and IEEE 754 Standard . . . 30
    - 2.2.1.1 Number Representation . . . 31
    - 2.2.1.2 IEEE 754 Standard and Conventions . . . 31

2.2.2 Interval Arithmetic 32
2.2.2.1 Real Intervals 32
2.2.2.2 Exact Interval Arithmetic 33
2.2.2.3 Rounded Interval Arithmetic 34
2.2.2.4 Interval Functions 35
2.2.2.5 From Closed Intervals to Open Intervals 35
2.2.3 Affine Arithmetic 36
2.2.3.1 Affine Form 37
2.2.3.2 Affine Operations 37
2.2.3.3 Non-Affine Operations 37
2.2.3.4 Variants of Affine Arithmetic 39
2.3 Fundamental Consistency Notions 43
2.3.1 Global Consistency 43
2.3.2 Classical Local Consistency Notions 44
2.3.2.1 Node Consistency 44
2.3.2.2 Arc Consistency 44
2.3.2.3 Path Consistency 45
2.3.2.4 k-Consistency 47
2.3.3 Local Consistency Notions for Numerical Constraints 48
2.3.3.1 Hull Consistency 48
2.3.3.2 kB-Consistency 49
2.3.3.3 Box Consistency 51
2.3.4 Extended Local Consistency Notions 53
2.3.4.1 (i, j)-Consistency 53
2.3.4.2 Relational Consistency 53
2.3.4.3 Singleton Consistency 54

3 An Overview of Solution Methods **55**
3.1 Mathematical Solution Methods 55
3.1.1 Fundamental Interval Fixed Point Methods 55
3.1.1.1 Krawczyk Iteration for Linear Equations 57
3.1.1.2 Interval Gauss-Seidel Iteration 58
3.1.1.3 Krawczyk Iteration for Nonlinear Equations 60
3.1.1.4 Hansen-Sengupta Iteration 62
3.1.1.5 Interval Newton Iteration 63
3.1.2 Other Interval Methods for Linear Systems 64
3.1.2.1 Interval Gauss Elimination 65
3.1.2.2 Hull Method 66
3.1.2.3 Linear Interval Inequalities 67
3.1.3 Exclusion Tests 68
3.1.3.1 Lipschitz Functions 69
3.1.3.2 Taylor Expansion 70
3.1.4 Exclusion Regions 71
3.1.5 Existence and Uniqueness Tests 73
3.1.5.1 Epsilon-Inflation 74
3.1.6 Inclusion Tests 75
3.2 Constraint Programming Methods 78

3.2.1 Classical Complete Search Methods 78
3.2.2 Branch-and-Prune Methods 79
3.2.2.1 Hull Consistency by Search 80
3.2.2.2 Hull Consistency by Propagation 81
3.2.2.3 Box Consistency by Search 85
3.2.2.4 Box Consistency by Propagation 86
3.2.2.5 kB-Consistency by Search 87
3.2.3 Cooperation of Solution Techniques 88
3.2.3.1 Cooperation of Domain Reduction Techniques 88
3.2.3.2 Cooperation of Symbolic-Interval Techniques 89
3.3 Relaxation Based Methods 90
3.3.1 Linear Relaxation with Linear Programming 90
3.3.1.1 Linear Relaxation Based on Quadratic Terms 91
3.3.1.2 More General Linear Relaxation Techniques 91
3.3.2 Exclusion Test Using Linear Programming 92
3.3.2.1 Exclusion Test Using Dual Simplex Method 92
3.3.2.2 Exclusion Test Using Taylor Expansion and Linear Programming 95
3.3.3 Linear Relaxation with Fixed Point Methods 96
3.3.3.1 Linear Relaxation of Separable Functions 96
3.3.3.2 Linear Relaxation of Factorable Systems 98
3.3.4 Convexification Based Methods 99
3.4 Incomplete Search Methods 100

4 Improvements to Search Strategies for Numerical CSPs 101
4.1 Introduction 101
4.2 Motivation 103
4.3 Representation of Non-isolated Solutions 105
4.3.1 Inner and Outer Approximations 105
4.3.2 Union Approximations 105
4.3.3 Qualification of Union Approximations 107
4.4 Exhaustive Search for CSPs with Non-isolated Solutions 108
4.4.1 Domain Reduction Operators 108
4.4.2 Complementary Boxing Operators 109
4.4.3 Domain Splitting Operators 113
4.4.4 Controlling the Reduction of Small Domains 114
4.4.5 Compact Representation of Solutions 116
4.4.6 Search Algorithms 119
4.5 Experiments 125
4.6 Conclusion 127

5 Modification and Abstraction of Inclusion Techniques 129
5.1 Introduction 129
5.2 Extended Functions 130
5.3 Modification to Interval Arithmetic 132
5.4 Revised Affine Arithmetic 133
5.4.1 Revised Affine Form 133
5.4.2 Multiplication 135

5.4.3 Division . . . 137
5.4.4 Non-Affine Unary Operations . . . 137
5.5 Abstraction of Inclusion Concepts . . . 141
5.5.1 Inclusion Representations . . . 141
5.5.2 Inclusion Functions . . . 146
5.6 Conclusion . . . 149

6 Numerical Constraint Propagation on Directed Acyclic Graphs 151
6.1 Introduction . . . 151
6.2 DAG Representations for Numerical CSPs . . . 152
6.2.1 Basic Concepts of Directed Acyclic Graphs . . . 152
6.2.2 DAG Representations . . . 153
6.3 Forward-Backward Propagation on DAG Representations . . . 155
6.3.1 Forward Evaluation on DAG Representations . . . 156
6.3.2 Backward Propagation on DAG Representations . . . 157
6.4 Partial DAG Representations for Numerical CSPs . . . 158
6.5 Constraint Propagation on Partial DAG Representations . . . 160
6.6 Coordinating Constraint Propagation and Search . . . 164
6.7 Experiments . . . 165
6.8 Conclusion . . . 168

7 Combination of Inclusion Techniques in Constraint Propagation 169
7.1 Introduction . . . 169
7.2 A Combination Scheme for Constraint Propagation . . . 170
7.2.1 Node Range Evaluations . . . 170
7.2.2 Induced Constraint Systems for Domain Reduction . . . 173
7.2.3 **CIRD** – A Generic Combination Scheme . . . 177
7.3 Specific Combination Strategies as Instances of **CIRD** . . . 179
7.3.1 Step 1a: Initial Node Evaluation . . . 179
7.3.2 Step 1b: Initialization of Waiting Lists . . . 180
7.3.3 Step 2a: Getting the Next Node . . . 180
7.3.4 Step 2b: Node Evaluation . . . 181
7.3.5 Step 2c: Node Pruning . . . 181
7.3.5.1 Backward Propagation . . . 182
7.3.5.2 Affine Pruning . . . 182
7.4 Experiments . . . 183
7.4.1 Comparisons with Interval Constraint Propagation Techniques . . . 183
7.4.2 Comparisons with Linear Relaxation Based Techniques . . . 185
7.5 Potential Directions for **CIRD** . . . 186
7.6 Conclusion . . . 187

8 Clustering Techniques for Disconnected Solution Sets 189
8.1 Introduction . . . 189
8.2 Goals of Clustering . . . 190
8.2.1 Basic Concepts . . . 190
8.2.2 Goal Setting . . . 191
8.3 Algorithms . . . 192

8.3.1 Optimal Max-Connected Clustering . . . 192
8.3.2 Separator Computation . . . 193
8.3.3 Max-Connected Clustering . . . 196
8.3.4 Separator-Driven Clustering . . . 198
8.3.5 Combinations of Algorithms . . . 200
8.3.5.1 Combination of **MCC** and **Colonization** . . . 200
8.3.5.2 Combination of **SDC** and **Colonization** . . . 201
8.4 Experiments . . . 202
8.5 Conclusion . . . 204

9 Conclusions **205**
9.1 Contributions . . . 205
9.2 Limitations and Challenges . . . 207
9.3 Further Research . . . 207
9.4 The Final Conclusion . . . 208

A Extended Concepts of Interval Arithmetic **209**
A.1 A Short History of Interval Arithmetic . . . 209
A.2 Typical Interval Functions . . . 210
A.2.1 Natural Interval Form . . . 210
A.2.2 Centered Interval Form . . . 210
A.2.3 Mixed Centered Interval Form . . . 210
A.2.4 Taylor Interval Form . . . 211
A.2.5 Linear Interval Mapping . . . 211
A.3 Advanced Concepts on Intervals . . . 212
A.3.1 Interval Matrix . . . 212
A.3.2 Interval Matrix Inverse . . . 213
A.3.3 Interval Slope . . . 214

B Fixed Point Theory in Metric Spaces **217**
B.1 Basic Concepts on Metric Spaces . . . 217
B.2 Fundamental Fixed Point Theorems . . . 219

C Numerical Benchmarks **223**
C.1 Problems with Continuums of Solutions . . . 223
C.2 Test Case T_1: Problems with Isolated Solutions . . . 227
C.3 Test Case T_2: Problems with Isolated Solutions . . . 229
C.4 Test Case T_3: Problems with Isolated Solutions . . . 230
C.5 Test Case T_4: Problems with Continuums of Solutions . . . 232
C.6 Test Case T_5: Problems with Continuums of Solutions . . . 232

Bibliography **235**

Glossary **253**

Index **257**

Curriculum Vitae **271**

Abstract

A *constraint satisfaction problem* (e.g., a system of equations and inequalities) consists of a finite set of constraints specifying which value combinations from given *variable domains* are admitted. It is called *numerical* if its variable domains are continuous. Such problems arise in many applications, but form a difficult problem class since they are NP-hard. Solving a constraint satisfaction problem is to find one or more value combinations satisfying all its constraints. Numerical computations on *floating-point numbers* in computers often suffer from rounding errors. The *rigorous control of rounding errors* during numerical computations is highly desired in many applications because it would benefit the quality and reliability of the decisions based on the solutions found by the computations. Various aspects of *rigorous numerical computations* in solving constraint satisfaction problems are addressed in this thesis: search, constraint propagation, combination of inclusion techniques, and post-processing.

The solution of a constraint satisfaction problem is essentially performed by a *search*. In this thesis, we propose a new *complete* search technique (i.e., it can find all solutions within a predetermined tolerance) for numerical constraint satisfaction problems. This technique is general and can be used in place of *branching* steps in most *branch-and-prune* methods. Moreover, this new technique speeds up the most recent general search strategy (often by an order of magnitude) and provides a concise representation of solutions.

To make a constraint satisfaction problem easier to solve, a major approach, called *constraint propagation*, in the *constraint programming*[1] field is often used to reduce the variable domains (by discarding redundant value combinations from the domains). Basing on *directed acyclic graphs*, we propose a new constraint propagation technique and a method for coordinating constraint propagation and search. More importantly, we propose a novel generic scheme for combining multiple *inclusion techniques*[2] in *numerical* constraint propagation. This scheme allows bringing into the constraint propagation framework the strengths of various techniques coming from different fields. To illustrate the flexibility and efficiency of the generic scheme, we base on this scheme and devise several specific combination strategies for *rigorous* numerical constraint propagation using *interval constraint propagation*, *interval arithmetic*, *affine arithmetic*, and *linear programming*. Our experiments show that the new propagation techniques outperform previously available methods by 1 to 4 orders of magnitude or more in speed.

We also propose several post-processing techniques for the representation of continuums of solutions. Based on connectedness, they allow grouping each cluster of connected solution subsets into a larger subset, thus allowing getting additional grouping information. Potentially, these techniques enable interval-based solution techniques to be alternatives to *bounding-volume techniques* in applications such as collision detection and interactive graphics.

[1] *Constraint programming* is an approach to programming that relies on both *reasoning* and *computing*.
[2] An *inclusion technique* is to include a set of interest into enclosures. It is also called an *enclosure technique*.

Résumé

Un *problème de satisfaction de contraintes* se compose d'un ensemble d'énoncé de contraintes quel tuples des valeurs dans les *domaines des variables* sont admis. Il s'appelle *numérique* si ses domaines des variables sont continus. La résolution d'un problème de satisfaction de contraintes est de trouver un ou plusieurs tuples des valeurs satisfaisant à toutes ses contraintes. Les calculs numériques sur les nombres en points flottants des ordinateurs souffrent souvent d'erreurs d'arrondis. *Le contrôle rigoureux de ces erreurs d'arrondis* est fortement désiré dans de nombreuse applications, parce qu'il permet d'améliorer la qualité et la fiabilité des décisions basées sur les solutions trouvées. Cette thèse est consacrée à divers aspects du *calcul numérique rigoureux* lors de la résolution de problèmes de satisfaction de contraintes: la recherche, la propagation de contraintes, la combinaison des techniques d'inclusion, et le post-traitement.

La résolution d'un problème de satisfaction de contraintes est habituellement exécutée par une recherche. Nous proposons une nouvelle technique de recherche complète (c-à-d, elle peut trouver toutes les solutions à une tolérance prédéterminée) pour des problèmes de satisfaction de contraintes avec des *domaines continues*. Cette technique est générique et peut être utilisée en guise de la technique de *branchement* dans la plupart des méthodes de type *branch-and-prune*. De plus, cette technique accélère la technique de recherche générale la plus récente (souvent par un ordre de magnitude) et fournit une représentation concise des solutions.

Pour faciliter un problème de satisfaction de contraintes à résoudre, une approche dans *la programmation par contrainte*[1], appelée *la propagation de contraintes*, est souvent utilisée pour réduire les domaines des variables (en enlevant quelques tuples de valeurs redondantes des domaines). Baser sur les *graphes acycliques orientés*, nous proposons une nouvelle technique de propagation de contraintes et une méthode pour coordonner la propagation de contraintes et la recherche. De plus, nous proposons un nouveau schéma générique pour combiner des *techniques d'inclusion*[2] multiples dans la propagation de contraintes numériques. Ce schéma permet de bénéficier de la propagation de contraintes de la puissance de techniques provenant de diverses disciplines. Pour illustrer la flexibilité et l'efficacité du schéma générique, nous imaginons du schéma plusieurs stratégies spécifiques de combinaison impliquant *la propagation de contraintes d'intervalles*, *l'arithmétique d'intervalle*, *l'arithmétique affine*, et *la programmation linéaire*. Nos expériences prouvent que la nouvelle approche surpasse des techniques précédemment disponibles par un à quatre ordres de magnitude ou plus en vitesse.

En plus, nous proposons plusieurs techniques de post-traitement pour la représentation des continuums de solutions. Elles permettent de grouper des sous-ensembles de solutions connectées dans de plus grand sous-ensembles, ameliorant ainsi la concision de la représentation. Ceci permet potentiellement aux techniques de résolution basées sur des intervalles de se substituer aux *techniques de bounding-volume* classiques dans des applications telles que la détection de collision ou le graphisme interactif.

[1] *La programmation par contrainte* est une approche qui combine *raisonnement* et *calcul*.
[2] Une *technique d'inclusion* (ou *technique de clôture*) doit inclure un ensemble d'intérêts dans des clôtures.

Acknowledgements

This thesis is dedicated to my advisors, Dr. Djamila Sam-Haround and Prof. Boi Faltings. By this, I wish to express my deepest appreciation and gratitude to my advisors who have sponsored and inspired me continuously. Without their sponsorships, this thesis would never be completed. I am much more than thankful to them for their advices, encouragements and assistances during my PhD career. I am much graceful to Djamila Sam-Haround for giving me complete freedom in time management, in research direction, and even in leaning to mathematics, the subject I love since high school time. However, she has always been there to help me when necessary. I also thank Dr. Martin Rajman for accepting to fund me in the last year of the thesis, the crucial year for completing the thesis. Specially, I wish to thank Vu Duong Thang for giving me the pointer to this PhD job.

I wish to thank Dr. Hermann Schichl, Prof. Arnold Neumaier, and the other colleagues in the COCONUT project (IST-2000-26063) for their generous help and fruitful collaborations during the time of the project (from December 2000 to February 2004). Much experience I have gained since that time is gracefully acknowledged to the COCONUT members. I would like to thank ILOG for granting us a permission to use ILOG Solver and ILOG CPLEX during the COCONUT project. I also thank the IRIN team of the University of Nantes in France for the open source code of their **HC4** algorithm in the COCONUT platform. I am graceful to Benny Raphael for providing me the beautiful figures and detailed descriptions of civil engineering problems in Appendix C. The precise timer used in all my implementations of the algorithms in this thesis is owning to Nguyen Tuan Viet. I am very thankful to Bui Huu Trung, my office mate, for helping me to print and bind many copies of this thesis at the most critical time of the thesis submission. I also graceful to Bui Hai Thanh, Le Hung Son, Nguyen Vu Hieu, To Huy Cuong, and Do Quang Yen for giving me their comments on this thesis.

I would like to thank Prof. Claude Petitpierre for chairing my thesis committee. I am very thankful to Prof. Arnold Neumaier, Prof. Michel Rueher, and Prof. Mohammad Amin Shokrollahi for being members of my thesis committee. I also thank them for their valuable comments and discussions on my thesis.

I would never forget the friendly help received from Do Ngoc Minh, Nguyen Tuan Anh, Cao Thanh Thuy, Luu Huong Tram Hong, and Marius-Calin Silaghi since the days I was new to the surroundings in Switzerland. Four-year studying life at EPFL has been the source of great pleasure for me. I wish to thank my friends and colleagues there for their friendly help and enthusiastic collaboration, especially Omar Belakhdar, Romaric Besançon, Monique Calisti, Jean-Cédric Chappelier, Cristian Ciressan, Ion Constantinescu, Santiago Macho Gonzalez, Frédéric Goualard, Christophe Jermann, Radu Jurca, Miroslav Melichar, Adrian Petcu, David Portabella, Antoine Rozenknop, Vincent Schickel-Zuber, Florian Seydoux, Paolo Viappiani, Alex Trutnev, and Steven Willmott. I also would like to thank Mrs. Marie Decrauzat for her administrative supports.

I wish to share a big hug with my Vietnamese friends, who have discussed everyday occurrences with me, especially Bui Hai Thanh, Bui Huu Trung, Ho Quoc Bang, Le Dinh Duy, Le Hung Son, Le Lam Son, Luu Vinh Toan, Mai Tuan Anh, Nguyen Ngoc Anh Vu, Nguyen Phi Hung, Nguyen Quang Huy, Nguyen Thanh Tung, Nguyen Tuan Viet, Nguyen Vu Hieu, Nguyen Xuan Hung, Tran Doan Binh, Tran Hien Dat, Vo Duc Duy, and Vu Le Hung. Thanks to them, nearly one thousand stressful hours in writing up this thesis have been harmoniously relaxed by many enjoyable hours.

I want to thank my high school mathematical teacher, Pham Ngoc Quang, for specially inspiring me with the first motivation of my scientific life and for encouraging me throughout the rest.

From the bottom of my heart, I am extremely thankful to my parents and my family for so many priceless things and so much great love they have ever given to me.

List of Algorithms

2.1 The **Solution** algorithm – a general solving process 22
3.1 The **Exclusion** algorithm 69
3.2 The **Inclusion** algorithm 77
3.3 The **Branch&Prune** algorithm – the recursive version 80
3.4 The **Branch&Prune** algorithm – the loop version 80
3.5 Function **Prune&Check**($\mathcal{P}$, SOLUTIONS, ATOMS) 80
3.6 The $\mathbb{F}$-**Hull Consistency** algorithm – $\mathbb{F}$-hull consistency by search 81
3.7 The **HC4revise** algorithm – a forward-backward propagation on a tree 84
3.8 Procedure **RFE**(**in/out:** a compact tree $\mathcal{T}_E$; **in:** $\mathbf{x} \in \mathbb{I}^n$) 84
3.9 Procedure **RBP**(**in/out:** a compact tree $\mathcal{T}_E$, $\mathbf{x} \in \mathbb{I}^n$) 84
3.10 The **HC4** algorithm – hull consistency on primitive constraints 84
3.11 The **BC3Revise** algorithm – box consistency by search 85
3.12 Function **SearchLowerBound**(**in:** an interval form Γ, $\mathbf{x} \in \mathbb{I}^n$, $i \in \{1, \ldots, n\}$) 85
3.13 Function **SearchUpperBound**(**in:** an interval form Γ, $\mathbf{x} \in \mathbb{I}^n$, $i \in \{1, \ldots, n\}$) 85
3.14 The **BC3** algorithm – box(Γ) consistency by bounds search + **AC3** 86
3.15 The **BC4** algorithm – box(Γ) consistency by **BC3** + **HC4revise** 87
3.16 The kB-**Consistency** algorithm – the function $\Phi^{\mathbb{F}}_{k\mathrm{B}}$ 88
3.17 The **Fixed Point Combination** algorithm 89
3.18 The **Quad** algorithm – a propagation based on linear relaxations 91
4.1 The **UCA6** algorithm – a slightly improved version 120
4.2 Function **solveQuickly**($\mathbf{B}$, $\mathcal{C}$, $\{\mathbf{CB}_c \mid c \in \mathcal{C}\}$, ε, FC, WAITINGLIST) 120
4.3 Function **isEpsilonBox**($\mathbf{B}$, $\mathcal{C}$, ε, FC) 121
4.4 The **UCA6**$^+$ algorithm 122
4.5 Function **solveQuickly**$^+$($\mathbf{B}$, $\mathcal{C}$, $\{\mathbf{CB}_c \mid c \in \mathcal{C}'\}$, ε, FC, WAITINGLIST, D_{stop}) 123
4.6 Function **isEpsilonBox**$^+$($\mathbf{B}$, $\mathcal{C}$, ε, FC) 123
5.1 The **SafeChebyshevApprox**$^\uparrow$ algorithm 139
5.2 The **SafeChebyshevApprox**$^\downarrow$ algorithm 140
6.1 Procedure **NodeOccurrences**(**in:** a node $\mathbf{N}$; **in/out:** a vector V_{oc}) 160
6.2 The **FBPD** algorithm – a constraint propagation on DAGs 161
6.3 Procedure **ReForwardEvaluation**(**in**: a node $\mathbf{N}$; **in/out**: a vector V_{ch}, a list $\mathcal{L}_{\mathrm{b}}$) . 162
6.4 Procedure **NodeLevel**(**in:** a node $\mathbf{N}$; **in/out:** a vector V_{lvl}) 163
6.5 The **BnPSearch** algorithm – a generic branch-and-prune search 165
7.1 **CIRD** – a scheme for **c**ombining **i**nclusion **r**epresentations on **D**AGs 178
7.2 Procedure **RecursiveNodeEvaluation**(**in:** a node $\mathbf{N}$) 179
8.1 The **Colonization** algorithm – a naive clustering 192
8.2 The $\mathbb{I}_\circ$-**intersect** algorithm – intersection of two sets of intervals 195
8.3 Procedure **BoxTreeFitting**(**in/out:** a bounding-box tree $\mathcal{T}_0$) 196

8.4 Procedure **SeparatorPulling**(**in/out:** a node **P**) 196
8.5 The max-connected clustering (**MCC**) algorithm 197
8.6 Function **SubtreeSeparation**(**in:** a node **P**, a separator $S \in \texttt{SPT}(\mathbf{P})$; **out:** $l_{\mathbf{P}}$, $u_{\mathbf{P}}$) . 198
8.7 The separator-driven clustering (**SDC**) algorithm 199
8.8 Function **MaxSeparation**(**in:** a tree $\mathcal{T}$; **out:** a list $\mathcal{L}$ of trees decomposed from $\mathcal{T}$) 200

List of Definitions

2.1 Variable Domain . . . 10
2.2 Constraint . . . 11
2.4 Compound Label, Instantiation . . . 11
2.5 Satisfiability . . . 11
2.7 Constraint Satisfaction Problem ≡ CSP . . . 12
2.10 Consistent k-Instantiation . . . 12
2.11 Arity . . . 12
2.12 Solution . . . 13
2.13 Solution Set . . . 13
2.14 Consistency . . . 13
2.17 Constraint Graph ≡ Constraint Network . . . 14
2.18 Constraint Hypergraph . . . 14
2.19 Binary CSP, Strictly Binary CSP . . . 14
2.20 Normalized CSP . . . 14
2.21 Standardized CSP . . . 14
2.22 Regular CSP . . . 14
2.23 Negation . . . 15
2.24 Projection . . . 15
2.27 Section, Cross Section . . . 15
2.30 Slice . . . 16
2.32 Singleton CSP ≡ Section, Cross Section . . . 16
2.33 Slice of CSP . . . 17
2.35 Compact CSP . . . 17
2.36 Globally Solved Form . . . 17
2.37 Problem Equivalence . . . 18
2.38 Problem Relaxation . . . 18
2.39 Equivalence-Preserving . . . 18
2.40 Consistency-Preserving . . . 18
2.41 Relaxing . . . 19
2.42 Consistency-Relaxing . . . 19
2.43 Domain Reduction Operator . . . 19
2.45 Redundant Value/Compound Label . . . 20
2.46 Redundant Constraint ≡ Implied Constraint . . . 20
2.47 Soundness . . . 21
2.48 Completeness, Incompleteness . . . 21
2.49 Asymptotical Completeness . . . 21
2.50 Rigor . . . 21

2.51 Continuous Domain . . . 24
2.53 Continuous Variable . . . 24
2.54 Numerical Constraint . . . 25
2.55 Numerical CSP ≡ NCSP . . . 25
2.59 Constraint Range . . . 25
2.62 Arithmetic Expression . . . 27
2.63 Factorable Expression . . . 28
2.66 Factorable Function . . . 28
2.68 Factorable Constraint . . . 28
2.70 Factorable CSP . . . 29
2.71 Separable Expression . . . 29
2.73 Separable Function . . . 29
2.75 Separable Constraint . . . 30
2.76 Separable CSP . . . 30
2.78 Interval Vector, Box . . . 33
2.80 Interval Form of Functions . . . 35
2.81 Interval Form of Relations . . . 35
2.83 Kolev Affine Form, Kolev Interval Form . . . 39
2.86 Strongness . . . 43
2.87 Monotonicity . . . 43
2.89 Global Consistency . . . 44
2.90 Node Consistency . . . 44
2.93 Arc Consistency . . . 44
2.96 Directional Arc Consistency . . . 45
2.98 Hyper-Arc Consistency . . . 45
2.100 Path Consistency . . . 46
2.102 Strong Path Consistency . . . 46
2.105 Directional Path Consistency . . . 46
2.107 k-Consistency . . . 47
2.108 Strong k-Consistency . . . 47
2.114 Hull Consistency . . . 48
2.115 $\mathbb{F}$-Hull Consistency . . . 48
2.119 kB-Consistency . . . 50
2.123 kB($\mathbb{F}$)-Consistency . . . 51
2.124 Box Consistency . . . 51
2.125 Box(Γ) Consistency . . . 52
2.127 Box$\langle\pm\varphi\rangle$ Consistency . . . 52
2.129 (i,j)-Consistency . . . 53
2.130 Strong (i,j)-Consistency . . . 53
2.132 Relational Consistency . . . 53
2.134 Singleton Consistency . . . 54
3.3 Strong Convergence . . . 56
3.4 Interval Contraction ≡ Narrowing/Contracting Operator . . . 57
3.6 Krawczyk Operator for Linear Equations . . . 58
3.8 Gauss-Seidel Operator . . . 59
3.13 Krawczyk Operator for Nonlinear Equations . . . 60
3.17 Hansen-Sengupta Operator . . . 62

3.20 Interval Newton Operator . . . 63
3.23 Gauss Inverse . . . 65
3.24 Exclusion Test . . . 68
3.29 Existence Test . . . 73
3.30 Uniqueness Test . . . 73
3.33 Generalized Inclusion Test . . . 76
3.43 Kolev Operator . . . 98
4.3 Inner Approximation . . . 105
4.4 Outer Approximation . . . 105
4.5 Inner Union Approximation, $\boxplus^{\mathcal{I}}$ . . . 106
4.6 Outer Union Approximation, $\boxplus^{\mathcal{O}}$ . . . 106
4.7 Boundary Union Approximation, $\boxplus^{\mathcal{B}}$ . . . 106
4.9 Feasibility Checker, FC . . . 107
4.10 Interval-Based Precision . . . 108
4.11 Domain Reduction Operator, DR . . . 108
4.13 Complementary Boxing Operator, CB . . . 110
4.18 Monotonicity . . . 113
4.20 Dichotomous Splitting Operator, DS . . . 113
4.22 Box Splitting Operator, BS . . . 114
4.23 Active/Inactive Variable . . . 115
4.24 Color Function . . . 117
4.25 Extreme Vertex . . . 118
5.1 Multifunction, Values, Fibers . . . 130
5.2 Image, Inverse Image . . . 130
5.3 Extended Function . . . 130
5.9 Interval Form of Multifunctions . . . 132
5.11 Interval Division: $[\div_{\emptyset}]$, $[\div_{\mathbb{R}}]$, $[\div_{\star}]$ . . . 132
5.22 Inclusion Representation . . . 141
5.23 Real Inclusion Representation, Real Representation . . . 141
5.36 Inclusion Function of Functions . . . 146
5.37 Inclusion Function of Multifunctions . . . 146
5.38 Inclusion Relation . . . 146
5.41 Natural Extension . . . 147
5.45 Inclusion Conversion . . . 148
6.1 Directed Multigraph . . . 152
6.2 In-Edges, Out-Edges . . . 152
6.3 Leaf, Root . . . 152
6.4 Directed Path, Cycle, Directed Acyclic Multigraph . . . 152
6.5 Parent, Child, Ancestor, Descendant . . . 153
6.7 Directed Multigraph with Ordered Edges . . . 153
6.10 Forward Evaluation Rule . . . 156
6.16 Backward Propagation Rule . . . 158
7.4 Node Evaluation, NEV . . . 171
7.6 Inclusion Constraint System, ICS . . . 173
7.9 Pruning Constraint System, PCS . . . 174
8.1 Separator, SPT . . . 193
8.2 Extension, EXT . . . 193

8.3 Separating Set, SE . . . 194
A.1 Natural Interval Form . . . 210
A.2 Centered Interval Form . . . 210
A.3 Mixed Centered Interval Form . . . 210
A.4 Taylor Interval Form . . . 211
A.5 Linear, Sublinear Mapping . . . 211
A.6 Spectral Radius . . . 212
A.7 Extended Matrix Inverse . . . 212
A.8 Interval Matrix . . . 212
A.9 Comparison Matrix . . . 212
A.10 M-Matrix . . . 212
A.11 H-Matrix . . . 213
A.12 Linear Interval Equation . . . 213
A.13 Linear Interval Inequalities . . . 213
A.14 Hull Inverse . . . 213
A.15 Fixed Point Inverse . . . 213
A.16 Regular Matrix Set . . . 214
A.17 Strongly Regular Interval Matrix . . . 214
A.18 Interval Matrix Inverse . . . 214
A.20 Lipschitz Set, Lipschitz Matrix . . . 214
A.21 Slope, Slope Matrix . . . 215
A.22 Slope Form . . . 215
A.23 Second Order Slope Matrix . . . 215
A.25 Slope Set, Interval Slope Matrix . . . 216
B.1 Metric Space . . . 217
B.2 Precompact Set . . . 217
B.3 Closed/Open Set . . . 217
B.4 Cover, Subcover . . . 217
B.5 Compact Metric Space, Compact Set . . . 218
B.6 Ring, Field . . . 218
B.7 Vector Space . . . 218
B.8 Norm, Normed Vector Space . . . 218
B.9 Convex Set . . . 219
B.10 Euclidean Distance, Hausdorff Distance . . . 219
B.11 Lipschitz/Nonexpansive/Contractive/Contraction Mapping . . . 219
B.12 Compact Mapping . . . 220
B.13 Fixed Point . . . 220

List of Figures

3.1 The tree representation of a factorable NCSP 83
3.2 The compact tree representation of a factorable NCSP 83

4.1 An example of NCSPs with: (a) isolated solutions; (b) continuums of solutions . . 102
4.2 An example of a poorly informative representation 104
4.3 An example of inner and outer approximations 105
4.4 An example of inner and outer union approximations 106
4.5 An example of a domain reduction (DR) operator 109
4.6 An example of a complementary boxing (CB) operator 110
4.7 Domain reduction (DR) operators and complementary boxing (CB) operators 111
4.8 Examples of box splitting (BS) and dichotomous splitting (DS) operators 114
4.9 Normal domain reductions and restricted-dimensional domain reductions 116
4.10 Examples of a griddy polyhedron and an orthogonal polyhedron 117
4.11 A griddy polyhedron with samples of colors 117
4.12 An example of extreme vertices of union approximations 118
4.13 Logarithmic charts for: (a) the running time; and (b) the total number of boxes . 126

6.1 A node and its computational flows in a DAG representation 154
6.2 The DAG representation of the constraint system (6.2) 155
6.3 Partial DAG representations of the problem (6.2) 159
6.4 The DAG representation of the system (6.2) after a recursive forward evaluation . 162
6.5 The node levels are updated at each call to the **FBPD** algorithm 163

7.1 The DAG representation: (a) before; and (b) after interval evaluations 171
7.2 A pruning constraint system 175
7.3 The distribution of auxiliary variables (ϵ_1, ϵ_2) in a DAG representation 179

8.1 Examples of orthogonal-separable and non-orthogonal-separable trees 191
8.2 Examples of separators and extensions 193
8.3 The fitting process on a bounding-box tree 194
8.4 The separating process: constructing lower-bound and upper-bound trees 197
8.5 Max-Connected Clustering 198
8.6 Separator-Driven Clustering 199
8.7 Examples of the output from **MCC** and **MCC** 201
8.8 Examples of the output from **OSDC** 201
8.9 The average running times (a) in three groups; and (b) for all problems 203

C.1 The geometric design of a truss 223

C.2 The design of a column for combined axial load and moment 225

List of Tables

2.1 IEEE 754 floating-point formats 32
2.2 IEEE 754 special quantities 32

3.1 The general form of a basic tableau in the simplex method 92
3.2 The reduced form of a basic tableau in the simplex method 93
3.3 The optimal feasible tableau for the linear program (3.99) 94
3.4 The optimal tableau for the linear program (3.100) 95

4.1 The running time results for search algorithms. 125
4.2 The numbers of boxes in inner and boundary union approximations 125
4.3 The ratio of the inner volume to the outer volume 126

5.1 Examples of functions $f \in \mathcal{C}^1([a, b])$ satisfying the conditions of Theorem 5.19 . . . 139

6.1 A comparison of three constraint propagation techniques: **FBPD**, **BOX**, **HC4** . . . 167
6.2 The averages of the relative time ratios in each test case 167
6.3 The overrun ratios for the test case T_1 167

7.1 A comparison of three constraint propagation techniques: **CIRD[ai]**, **BOX**, **HC4** . 184
7.2 The averages of the relative time ratios in each test case 184
7.3 A preliminary comparison of **Quad** and **CIRD[ai]** 186

8.1 The average running times and the average number of clusters 202

Chapter 1

Introduction

Everybody has learned, since high school time, to solve particular mathematical problems; for example, has learned to

- solve univariate quadratic equations by using the *completing the square method* (an ancient version of the *quadratic formula* known to the Babylonians,[1] circa 400 BC);
- solve systems of linear equations by using the *Gauss elimination* method;
- find local and global extrema based on the *first order derivative test*, *second order derivative test*, or *extreme test*.

Such theoretical methods are however too limited for practical use. They can only solve problems of specific types or of small sizes. In practice, it is, however, often required to solve larger and more general problems. These problems remain out of the reach of those methods. Numerous approaches have been developed to tackle problems of practical interest. These approaches pertain to various disciplines including *mathematical programming*, *logic programming*, and *constraint programming*.

Mathematical programming mainly focuses on *mathematical computing* aspects, while logic programming leans towards *logic reasoning* aspects. They both have long histories with a vast literature. Constraint programming, which has been officially studied since circa the year 1973, involves both *mathematical computing* and *logic reasoning*.

This thesis is concerned with various aspects of the constraint programming approach for solving numerical nonlinear problems. Before introducing the goal of the thesis, a preview of constraint programming is given to allow better understanding the direction of the thesis.

The reader who is already familiar with basic concepts in constraint programming may skip the next section and start with the goal of the thesis in Section 1.2.

1.1. A Preview of Constraint Programming

Constraint programming is an approach to programming that relies on a combination of *reasoning* and *computing*, with emphasis on the interaction among constraints. It offers a declarative framework for implementing problem solving methods. So far, it has been successfully applied in a number of fields including electrical engineering, design engineering, interactive graphics, natural language processing, numerical analysis, molecular biology, and operations research. See some applications in [Apt 2003].

[1] The ancient Babylonians did not have the quadratic formula of today but they had their own ancient method. The quadratic formula, though, is simply a formalization of this ancient method.

Constraint Programming [Apt 2003]. The central concept in constraint programming is the concept of a constraint. Mathematically, a *constraint* on a sequence of variables is a relation over their domains. It can also be viewed as a requirement stating which tuple of values is admitted. To state multiple requirements that need to be satisfied, the concept of a constraint satisfaction problem has also been introduced. Roughly speaking, a *constraint satisfaction problem* (CSP) consists of a finite set of constraints, each on a subsequence of a given sequence of variables. These are the most fundamental concepts in constraint programming.

First, to solve a given problem with the use of constraint programming, we formulate its requirements as a constraint satisfaction problem. This task is called *problem modeling*, which consists of

- introducing some variables to represent the quantities of interest;
- associating each variable with a domain that contains values of possible interest;
- expressing some constraints on the variables to state the requirements of the problem.

The modeling phase results in a representation of the problem as a constraint satisfaction problem. Along with representing the problem, we need to identify the *target* of the problem, which often amounts to *either*

- determining if the representation has a *solution*[2];
- finding one or more *solutions*;
- finding one or more *optimal solutions* w.r.t. some quality measure, which is often expressed as minimizing (or maximizing) an *objective function*.

While the first is casted into a *satisfiability problem* and the last is formulated as a *constrained optimization problem*, the second defines the constraint satisfaction problem itself.

Second, to solve the chosen representation of the problem by means of constraint programming, we may use general methods, domain-specific methods, or a combination of them:

- The *general methods* in constraint programming are concerned with reducing the *search space* and with specific *search methods*. The algorithms that deal with the search space reduction are usually called *constraint propagation techniques*. These techniques aim at computing an equivalent, but simplified, representation of the considered problem. They achieve various forms of *local consistency*, which is a state achieved by an algorithm such that some tuples of values that do not satisfy some constraints are discarded. This notion approximates the notion of *global consistency*, which is a state that each instantiation of values to variables can be extended to a solution. The general methods often combine various forms of constraint propagation with either the customary *backtracking*, *branch-and-bound*, or *branch-and-prune* frameworks for search, depending on the target of the considered problem.

- The *domain-specific methods* are usually provided in the form of special purpose algorithms, such as the *linear programming* algorithms that only deal with *linear functions*. These methods exploit the specialities of the considered problem in order to solve them more efficiently. In general, if domain-specific methods are applicable, they should be used in place of the general methods. For example, it is far more efficient to use linear programming techniques, such as the *Gauss-Jordan elimination* method and the *simplex method*, than to use the general methods when solving a system of linear equations.

[2] In this thesis, each tuple of values in considered variable domains that satisfies all constraints of a problem is called a *solution* of the problem.

In fact, one of the aims of constraint programming is to look for efficient domain-specific methods that can be used in specific circumstances instead of the general methods and to incorporate them into a general framework seamlessly. Such a framework usually supports

- domain-specific methods, often implemented in specific *constraint solvers*;
- general methods, often in the form of *constraint programming languages*, by facilitating the use of constraint propagation algorithms in various search methods.

Mathematical Rigor. A *solution algorithm* (or a *solution technique*, or a *solution method*) for a problem is a sequence of steps for solving the problem. Following the classification in [Neumaier 2004], the different solution methods can be classified into four classes according to the degree of *mathematical rigor*[3] as follows (see Section 2.1.2.3 for formal definitions):

1. ***Incomplete solution methods*:** which use heuristics to accelerate the solving process, but do not guarantee to find all solutions;

2. ***Asymptotically complete solution methods*:** which find a solution with certainty or at least with probability one;

3. ***Complete solution methods*:** which find a solution with certainty, assuming exact computations and indefinitely long run time;

4. ***Rigorous solution methods*:** which find a solution within prescribed tolerances with certainty, even in the presence of rounding errors.

For problems with discrete domains, the last two categories are identical. A complete solution algorithm is able to compute all approximate solutions systematically. However, *only rigorous solution algorithms can be relied upon to provide all relevant solutions and guarantee that all requirements of the problem are rigorously satisfied. The rigor would thus benefit the quality and reliability of decisions based on the solutions found by the algorithms.*

1.2. The Goal of the Thesis

Many applications involve constraints of which variable domains are continuous (that is, each variable domain is a connected set of real numbers). In this case, the domains and variables are said to be *continuous*. A constraint on continuous variables is called a *numerical constraint*. A constraint satisfaction problem with numerical constraints is called a *numerical constraint satisfaction problem*. In practice, numerical constraints are often expressed in the *factorable form*; that is, each expression of which is recursively composed of *elementary operations* or functions, such as $+$, $-$, $\times$, $\div$, log, sin, and cos. Numerical constraints may be simple or complex, linear or nonlinear, or even may involve logic functions (usually of first order).

A problem is said to be *linear* if all its constraints are linear, or to be *nonlinear* otherwise. If some variable domains of a linear problem (respectively, a nonlinear problem) contain only integers, the problem is called a *mixed integer linear problem* (respectively, a *mixed integer nonlinear problem*). When considering constrained optimization problems, we often replace the term *problem* in these concepts by the term *program*, hence getting the well-known concepts of a *linear program* (LP), a *nonlinear program* (NLP), a *mixed integer linear program* (MILP), and a *mixed integer nonlinear program* (MINLP) in mathematical programming.

[3] The *mathematical rigor* is defined as amenability to algorithmic checking of correctness.

Arising very early in many practical applications, constrained optimization problems have been attracting the attention of numerous researchers around the world. Numerous techniques have been proposed in mathematical programming to solve these problems. However, the only class of optimization problems that can now be completely and efficiently solved for many large scale problem instances (say, with hundred thousands of variables) is the class of MILPs. Although sharing with MILPs the feature that fixing all integer variables may lead to a tractable problem, MINLPs are still much more difficult to solve due to the nonlinearity. In general, the most powerful mathematical programming techniques are usually applicable to restricted classes of constraints only and are sometimes domain-specific.

A (numerical) constraint satisfaction problem can be viewed as a constrained optimization problem with a constant objective function. Hence, it can be theoretically solved by using solution algorithms for constrained optimization problems. However, most efficient solution algorithms for constrained optimization problems are heavily based on the interaction between objective functions and constraints. For example, the first order necessary optimality conditions such as the *Karush-Kuhn-Tucker conditions* and the *second order necessary/sufficient optimality conditions* (see [Nocedal and Wright 2000, Chapter 12], [Neumaier 2004, Section 5]) do not help reduce the search space when the objective function is constant because all solutions are optimal. In this case, more intensive techniques are needed for efficiently solving numerical constraint satisfaction problems.

Since the arising of many practical applications that involve constraint satisfaction problems, a new framework, originally called *constraint satisfaction* and recently called *constraint programming*, emerged thirty years ago to concern with solving constraint satisfaction problems (see Section 2.1.1). At the beginning, constraint satisfaction techniques only focused on the reasoning in solving constraint satisfaction problems with discrete domains. Given such a problem, constraint satisfaction techniques often insisted on taking into account the interaction among constraints of the problem and on exploring the search space by intelligently enumerating solutions. Consequently, the constraint satisfaction techniques could deal with general constraints and handle heterogeneous problems while maintaining the completeness and the rigor. Later on, many techniques for continuous domains in mathematical programming were adapted to the framework of constraint satisfaction to help solve numerical constraint satisfaction problems more efficiently. The gap between continuous and discrete domains has been bridged by using progressive *discretization* or *splitting techniques*. Nowadays, devised techniques are often referred to as *constraint programming techniques*, which imply a combination of *computing* and *reasoning*. A side effect of the combination is that such constraint programming techniques no longer possess the generality that holds in the original framework of constraint satisfaction.

Although the generality and completeness of constraint programming techniques are high, their performance still does not suffice to handle problems of practical size. We have realized that, among the reasons, the inefficiency is caused by the following limitations:

L1 Constraints have been individually represented and the interaction among them has not been sufficiently taken into account during the solving process;

L2 The strengths of domain-specific methods have not been sufficiently explored in the general framework;

L3 Under-constrained problems have been improperly treated as the same as well-constrained problems (i.e., the ones with isolated solutions).

Therefore, we have set the goal for the thesis as follows.

The goal is to develop rigorous solution algorithms for numerical constraint satisfaction problems such that they reduce the limitations **L1**, **L2** and **L3**.

This goal builds on the idea that the strengths of different domain-specific methods, such as linear programming, interval analysis and affine arithmetic, are complementary. They can be integrated into the general framework of constraint programming, which allows exploring the interaction of constraints, to devise powerful solution techniques. The combination techniques will be able to preserve the generality and flexibility of constraint programming techniques while taking advantage of the powerful capabilities of domain-specific methods. Moreover, when under-constrained problems are considered, better splitting strategies can be used to avoid reconsidering some constraints in search regions that completely satisfy the constraints. In other words, the constraints that are implied by the other constraints in subproblems should be removed from further consideration.

1.3. Contributions of the Thesis

The detailed contributions of this thesis are presented at the end of each chapter (cf. the chapters 4, 5, 6, 7, and 8), and are summarized in Section 9.1. The major contributions are:

C1 In Chapter 4, we propose **a new *complete search* technique to solve numerical constraint satisfaction problems**. The proposed search technique is general and hence applicable to most *branch-and-prune* based solution methods.

C2 In Chapter 5, we propose **several improvements to, and an abstraction of, *inclusion techniques***[4]. These proposals are used in our new constraint propagation techniques in Chapter 6 and Chapter 7.

C3 In Chapter 6, we propose **a new *numerical constraint propagation* technique and a method for coordinating it and *search* on directed acyclic graphs**. Our experiments show that the new technique outperforms previously available techniques by 1 to 2 orders of magnitude or more in speed.

C4 In Chapter 7, our contribution is twofold:

(a) We propose **a novel *generic scheme* to combine multiple inclusion techniques in *numerical* constraint propagation**. The scheme potentially allows bringing into the constraint propagation framework the strengths of different techniques coming from different fields.

(b) We base on the generic scheme and devise **several specific *combination strategies* for *numerical* constraint propagation** using *interval constraint propagation*, *interval arithmetic*, *affine arithmetic*, and *linear programming*. Our experiments on a particular strategy show that the new approach outperforms previously available techniques by 1 to 4 orders of magnitude or more in speed.

[4] An *inclusion technique* is to include a set of interest into enclosures. It is also called an *enclosure technique*.

C5 In Chapter 8, we propose **several *post-processing* techniques for the representation of continuums of solutions**. Based on connectedness, they allow grouping each cluster of connected solution subsets into a larger subset. Therefore, they enable *interval-based solution techniques* to be alternatives to bounding-volume techniques in applications such as collision detection and interactive graphics.

Additional Contributions. I present the necessary background in Chapter 2 in my view because there is no standard in numerical constraint programming. I also present in Chapter 3 an overview of major existing methods for solving numerical constraint satisfaction problems, with emphasis on complete methods. The overview is more detailed than as usual because I see that many techniques can be easily integrated into the generic scheme proposed in Chapter 7. Moreover, an one-page summary of each important technique is certainly helpful to whom it is not familiar to. In some cases, I find the original versions ambiguous or even erroneous, hence attempting to make them clear or correct them in Chapter 3. In some other cases, I find that the original presentations of techniques can be made much more concise and easier to understand, hence giving alternative presentations in my view. Note that some remarks on, and some extensions of, techniques therein are my personal opinions; hence, they do not necessarily reflect the ideas of the original authors. It is to provide a better understanding of different techniques in a common view and language.

1.4. Organization of the Thesis

The outline of this thesis is as follows:

- In Chapter 1, we give an introduction to the goal of the thesis, the main contributions and the organization of the thesis.
- In Chapter 2, we present the necessary background. Section 2.1.1 presents the basic concepts in *constraint programming* including the concept of a *constraint satisfaction problem.* Section 2.2 presents the basic concepts involving *floating-point numbers*, *interval arithmetic*, and *affine arithmetic.* Section 2.3 presents some fundamental consistency notions, including some *local consistency* notions for *numerical constraints.*
- In Chapter 3, we present an overview of major existing methods for solving *numerical constraint satisfaction problems*, with emphasis on *complete methods.* Section 3.1 presents an overview of fundamental *mathematical techniques*, including very recent methods. Section 3.2 presents some recent fundamental solution techniques in *constraint programming.* Section 3.3 is dedicated to *relaxation* based methods, mainly *linear relaxation* based methods. One may need some concepts in Appendix A and Appendix B.
- In Chapter 4, we present a new *complete search technique*, called **UCA6**$^+$, for solving numerical constraint satisfaction problems (see Contribution **C1**). Section 4.3 presents the concepts of *inner/outer/boundary union approximations* and the concept of a *feasibility checker.* Section 4.4 presents the concepts of a *domain reduction operator*, a *complementary boxing operator*, and a *splitting operator.* This section also presents the details of the **UCA6**$^+$ algorithm. A summary of our contributions in this chapter, including a conclusion, is given in Section 4.6.

- In Chapter 5, we present several new improvements to, and an abstraction of, inclusion techniques (see Contribution **C2**). Section 5.2 presents the concept of an *extended function* – a special *multifunction*. Section 5.3 presents the revision of *interval arithmetic*. Section 5.4 presents some improvements and proposals to *affine arithmetic*. Section 5.5 present the concept of an *inclusion representation*. A summary of our contributions in this chapter, including a conclusion, is given in Section 5.6.

- In Chapter 6, we present a new *constraint propagation* technique, called **FBPD**, and a method for coordinating constraint propagation and search on *directed acyclic graphs* (DAGs) (see Contribution **C3**). Section 6.2 presents the concept of a *DAG representation*. Section 6.3 defines the concepts of a *forward evaluation* and a *backward propagation* on DAGs. Section 6.4 presents the concept of a *partial DAG representation*. Section 6.5 presents the **FBPD** algorithm – a *constraint propagator* on the partial DAG representation. Section 6.6 presents a way to coordinate constraint propagation and search in the *branch-and-prune* framework. A summary of our contributions in this chapter, including a conclusion, is given in Section 6.8.

- In Chapter 7, we present a novel generic scheme for combining multiple *inclusion techniques* in *numerical constraint propagation*. We also present several specific combination strategies for numerical constraint propagation (see Contribution **C4**). Section 7.2 presents a generic scheme, called **CIRD**, based on the concepts of a *inclusion constraint system* and a *pruning constraint system*. Section 7.3 presents specific combination strategies based on *interval constraint propagation*, *interval arithmetic*, *affine arithmetic*, and *linear programming*. Experiments on a particular instance of these strategies, called **CIRD[ai]**, are presented in Section 7.4. Section 7.5 presents some potential directions for integrating inclusion techniques into **CIRD**. A summary of our contributions in this chapter, including a conclusion, is given in Section 7.6.

- In Chapter 8, we present several new *post-processing techniques* for the representation of *continuums of solutions* (see Contribution **C5**). Section 8.2 presents the goals for *clustering* with several definitions based on *connectedness*. Section 8.3 presents our new techniques. They are called **MCC**, **SDC**, **OMCC**, and **OSDC**. A summary of our contributions in this chapter, including a conclusion, is given in Section 8.5.

- In Chapter 9, we give the conclusions of the thesis, including detailed contributions, limitations, challenges, and potential directions.

- In Appendix A, we give some definitions needed for some techniques in Chapter 3.

- In Appendix B, we give some basic concepts and results of the *fixed point theory*.

- In Appendix C, we give some numerical problems used in some experiments in this thesis.

Chapter 2

Background and Definition

2.1. Basic Concepts in Constraint Programming

2.1.1. A Short History of Constraint Programming

The following history is composed from [Marriott and Stuckey 1998] and [Apt 2003]. The concept of *constraint solving* has a long history in mathematics. Finding solutions to simple equations has been considered since thousands of years ago. For example, the ancient Babylonians found the *completing the square method* to solve univariate *quadratic equations* circa the year 400 BC. They have even solved a variety of equations with two variables. Diophantus wrote his *Arithmetica* around the year 250.

According to Struik [1948], the first general solution in integers to equations of the form $aX + bY = c$ is due to the Indian mathematician Brahmagrupta from the seventh century. In the ninth century, the great Persian astronomer/mathematician al-Khwarizmi, believed to have lived from the year 780 to the year 850, wrote an influential treatise on *equation solving*. The term *algorithm* was derived from al-Khwarizmi's latinized name.

By the end of the eighteen century, solving linear equations by *variable elimination* was a common technique. A form of the popular *Gauss elimination* method has been used since as early as the year 1809. The *algebra* of *propositional logic* is due to Boole [1847]. In the late twentieth century, Frege extended Boole's algebra by developing *predicate logic*. The concepts of a *term*, a *tree*, and *tree constraint solving* are the cornerstone of most automated deduction.

Apart from mathematics, the two areas that have had the greatest impact on a modern theory of constraints and their use in automated problem solving are *operation research* (OR) and *artificial intelligence* (AI). Operation research is concerned with building mathematical models of real world situations to allow the experimental analysis of problems. Artificial intelligence is concerned with intelligently accelerating automated problem solution techniques.

The concept of solving constraints by using *constraint propagation* was already developed independently by a number of researchers to solve *arithmetic* and *Boolean constraints*. The term generally is attributed to Sussman and Steele [1980], who used constraint propagation to solve constraints in the constraint language CONSTRAINTS, while the use of constraint propagation can be traced back to an earlier work by Sutherland [1963] on the interactive drawing system SKETCHPAD.

The concept of a *constraint satisfaction problem* was also formulated in the seventies (of the twentieth century) by researchers in artificial intelligence. They also identified the main forms of *local consistency* and the algorithms that allow us to achieve them. Independently,

various *search methods* were defined. Some of them, like *backtracking* can be traced back to the nineteenth century, while others, such as *branch-and-bound* [Land and Doig 1960; Little *et al.* 1963], were defined in the context of *combinatorial optimization*. The contribution of *constraint programming* was to identify various new forms of search that combine the known techniques with various constraint propagation algorithms. Some specific combinations were already studied in combinatorial optimization.

In the eighties (1980s), the most significant constraint programming languages such as CONSTRAINTS were based on the *logic programming* paradigm. This led to a development of *constraint logic programming*, a combination of logic programming and constraint programming, arose with [Jaffar *et al.* 1986]. However, the first true constraint logic programming language is PROLOG II [Colmerauer 1982], while the first constraint logic programming language with real arithmetic is CLP($\mathcal{R}$) [Jaffar and Michaylov 1987]. Constraint propagation and various forms of search are usually available in these languages in the form of built-ins.

In the late eighties and the nineties, a form of synthesis between the developments of logic programming and constraint programming took place. The researchers found various new applications of constraint programming, most notably in the fields of operations research and numerical analysis. The progress was often achieved by identifying important new types of constraints and new constraint propagation algorithms. One also realized that further progress might depend on a combination of techniques from artificial intelligence, operations research, computer algebra and mathematical logic. This turned constraint programming into an interesting hybrid area, in which theoretical work is often driven by applications and, in turn, applications lead to new challenges concerning implementations of constraint programming.

2.1.2. Constraint Satisfaction

The concept of a constraint satisfaction problem is fundamental in constraint programming. Hence, we need to give a formal definition of it. We first introduce the concepts of a variable domain and a constraint, and then define the concept of a constraint satisfaction problem and related concepts. The class of constraint satisfaction problems is worth studying because it arises in a large number of applications.

2.1.2.1. Constraint Satisfaction Problems

By the term *variable*, we mean a quantity that can assume any of a set of values. In literature, a variable is often represented by a symbol such as x, X, and $\mathbf{X}$. Restricting the interest to a limited set of values leads to the definition of a *domain*.

Definition 2.1 (Variable Domain). The *domain of a variable* is a set of all considered values that can be assigned to the variable.

In practice, the domain of a variable, x, might be continuous or discrete. For instance, the domain of a real variable x given explicitly by a right-open real interval $[1, 5[$ is connected, hence continuous. A domain given explicitly by enumerating its values, such as $\{1, 9, 7, 3\}$, is discrete. Specially, a variable domain can be of a *heterogeneous* form, such as $\{1.2, 1, [-1, 1], \text{blue}, \text{table}, \{\texttt{false}, \texttt{true}\}\}$.

A domain can be a disconnected set, such as the union of two intervals $[-5, -2] \cup [2, 5]$. This domain can also given implicitly, such as $2 \leq |x| \leq 5$, or even more complicated, such as

$(x^2 - 4)(x^2 - 25) \leq 0$. However, in case the domain is not simply expressed, it often implies an implicit requirement on x that needs to be explored. This is generalized to a requirement on multiple variables, which is hereafter called a constraint – the central concept in constraint programming.

Definition 2.2 (Constraint). A *constraint*, C, on a finite sequence of variables $(x_1, \ldots, x_k)$ associated with respective domains $(D_1, \ldots, D_k)$ is a subset of the Cartesian product $D_1 \times \cdots \times D_k$, where $k \in \mathbb{N}$. The variables $x_1, \ldots, x_k$ are called the variables of C or the variables involving C. If $C = D_1 \times \cdots \times D_k$, it is called a *universal constraint*.

Note 2.3. Precisely, a constraint is defined on a **sequence** (or an ordered set) of its variables and not on a **set** of its variables [Apt 2003, p. 10].

The ordering of variables is necessary, otherwise the constraint $c\{x, y\} \equiv (x < y)$ will always be empty since the identity $c\{a, b\} \equiv c\{b, a\}$ leads to $(a < b) \equiv (b < a)$, where the domains of x and y contain both a and b. Another reason is that the Cartesian product $D_1 \times \cdots \times D_k$ is only defined on the sequence $(D_1, \ldots, D_k)$, not on the set $\{D_1, \ldots, D_k\}$.

We now introduce the following concept to facilitate the presentation of more complicated concepts in constraint programming.

Definition 2.4 (Compound Label, Instantiation). A *k-compound label*, also called a *compound label* for short, of a sequence of variables $(x_1, \ldots, x_k)$ with respective domains $(D_1, \ldots, D_k)$ is a tuple of values $(a_1, \ldots, a_k) \in D_1 \times \cdots \times D_k$. $(\langle x_1, a_1 \rangle, \ldots, \langle x_k, a_k \rangle)$ is called a *k-instantiation* of the values $(a_1, \ldots, a_k)$ to the variables $(x_1, \ldots, x_k)$.

Here, we define the *satisfiability* of a constraint.

Definition 2.5 (Satisfiability). A constraint C on a sequence of variables $(x_1, \ldots, x_k)$ is *satisfied* with a compound label $l \equiv (a_1, \ldots, a_k)$ of $(x_1, \ldots, x_k)$ (i.e., an instantiation $(\langle x_1, a_1 \rangle, \ldots, \langle x_k, a_k \rangle)$) if and only if $l \in C$. If the compound label l does not satisfy C, we say that l *violates* C.

Note 2.6. Let $(x_1, \ldots, x_k)$ be a subsequence of a sequence of variables $(v_1, \ldots, v_n)$ and C a constraint on the sequence $(x_1, \ldots, x_k)$. For convenience, we also say that C is satisfied with a compound label $l \equiv (a_1, \ldots, a_n)$ of $(v_1, \ldots, v_n)$ if the label $l_k \equiv (b_1, \ldots, b_k)$ of $(x_1, \ldots, x_k)$ taking respective values from $(a_1, \ldots, a_n)$ satisfies C.

Mathematically, a constraint can be viewed as a relation on its variable domains as in Definition 2.2. In real world applications, a constraint c on a sequence of variables $(x_1, \ldots, x_k)$ with domains $(D_1, \ldots, D_k)$ can also be viewed as a Boolean function

$$b_C : D_1 \times \cdots \times D_k \to \{\texttt{false}, \texttt{true}\},$$
$$b_C(x_1, \ldots, x_k) = \texttt{true} \Leftrightarrow (x_1, \ldots, x_k) \in C.$$

That is, $C = b_C^{-1}(\texttt{true}) \equiv \{(x_1, \ldots, x_k) \in D_1 \times \cdots \times D_k \mid b_C(x_1, \ldots, x_k) = \texttt{true}\}$. It follows that a label l satisfies the constraint C if and only if $b_C(l)$ holds. Consequently, one can use the meaning of a Boolean function for a constraint instead of the meaning of a relation. This also leads to the notation $C(x_1, \ldots, x_k)$, instead of C, when one wishes to emphasize that the constraint is defined on the sequence of variables $(x_1, \ldots, x_k)$. Note that the domain of a variable can always be interpreted as a special constraint.

In constraint programming, multiple requirements that need to be simultaneously satisfied can be represented by the concept of a *constraint satisfaction problem* defined as follows.

Definition 2.7 (Constraint Satisfaction Problem ≡ CSP). A *constraint satisfaction problem*, abbreviated to *CSP*, is a triple $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ in which

- $\mathcal{V}$ is a finite sequence of variables $(v_1, \ldots, v_n)$;
- $\mathcal{D}$ is a finite sequence of respective domains $(D_1, \ldots, D_n)$ of the variables $(v_1, \ldots, v_n)$;
- $\mathcal{C}$ is a finite set of constraints, each on a subsequence of $\mathcal{V}$.

Notation 2.8. For convenience, we use the notation $D_1 \times \cdots \times D_n$ to denote the domains $\mathcal{D}$ in Definition 2.7. We can also use the notation $\langle \mathcal{C};\ \mathcal{V} \in \mathcal{D} \rangle$ to refer to the CSP $(\mathcal{V}, \mathcal{D}, \mathcal{C})$.

Example 2.9. Consider a CSP written in the standard notation $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ with

$$\begin{aligned} &\mathcal{V} \equiv (x, y),\\ &\mathcal{D} \equiv ([-7, 7], [-3, 3]) = [-7, 7] \times [-3, 3],\\ &\mathcal{C} \equiv \{x^2 + y^2 = 1, x + y \le 1\}. \end{aligned}$$

It can be written shortly as

$$\langle x^2 + y^2 = 1,\ x + y \le 1;\ (x, y) \in [-7, 7] \times [-3, 3] \rangle,$$

or

$$\langle x^2 + y^2 = 1,\ x + y \le 1;\ x \in [-7, 7],\ y \in [-3, 3] \rangle.$$

Both variable domains of this CSP are continuous. ♣

Definition 2.10 (Consistent *k*-Instantiation). Given a CSP $\mathcal{P}$. A k-instantiation $(\langle x_1, a_1 \rangle, \ldots, \langle x_k, a_k \rangle)$ is said to be *consistent w.r.t. $\mathcal{P}$ (or the set of constraints of $\mathcal{P}$)* if it satisfies all the constraints on the subsequences of $x_1, \ldots, x_k$.

To measure the number of variables involving a constraint or a CSP, one may use the following terminology.

Definition 2.11 (Arity). The *arity of a constraint* (respectively, the *arity of a CSP*) is the number of variables of the constraint (respectively, of the CSP). A constraint of arity k is called a *k-ary constraint*. A CSP of arity k is called an *k-ary CSP*.

If the arity of a constraint is one, two or three, we say that the constraint is *unary*, *binary* or *ternary*, respectively. For example, the constraint $x^2 + y^2 + z^2 = 1$ is a constraint of second order and of arity three, where x, y, and z are its variables; hence, it is a ternary constraint.

The concept of a constraint satisfaction problem is one of the most important concepts in constraint programming. Once the constraint satisfaction problem is defined, one often requires to obtain one or more tuple of values assigned to the problem's variables. This leads to the concept of a solution defined as follows.

Definition 2.12 (Solution). A *solution* of a constraint satisfaction problem $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ is a compound label of $\mathcal{V}$ (i.e., a tuple of values $(d_1, \ldots, d_n) \in \mathcal{D}$) that satisfies all the constraints in $\mathcal{C}$.

Definition 2.13 (Solution Set). The *solution set* of a constraint satisfaction problem is the set of all solutions of the problem.

To indicate whether a CSP has a solution, we can use the following terminology.

Definition 2.14 (Consistency). If a CSP has at least one solution, we say that it is *consistent* or *satisfiable*, otherwise we say that it is *inconsistent*.

Example 2.15. Consider a CSP with integer variables (and hence called a discrete CSP)

$$\langle x^2 + y^2 = 25;\ x \in \mathbb{Z}_+,\ y \in \mathbb{N} \rangle,$$

where $\mathbb{Z}_+$ is the set of all positive integers and $\mathbb{N}$ is the set of all natural numbers including zero. The solution set of this CSP is the set $\{(5,0),(4,3),(3,4)\} \neq \emptyset$. Hence, this CSP is consistent. If we add a constraint $x + y < 5$, the problem becomes inconsistent. ♣

Example 2.16. Consider the problem

$$\langle x = y,\ x \vdots 3 \Rightarrow y \vdots 2;\ x \in \{1,2,3\},\ y \in \{1,2,3\} \rangle;$$

where the notation $a \vdots b$ means that b divides a. The solution set is $\{(1,1),(2,2)\}$; hence, this problem is consistent. ♣

According to Definition 2.7, the problem in Example 2.16 is a CSP. However, it belongs to a class of special CSPs, called the *constraint logic problems*, because it also contains *logic constraints*. The field that focuses on this class of problems is called *constraint logic programming* (see [Marriott and Stuckey 1998]). The CSPs that contain only logic predications belong to a class of special problems called *logic problems*, which are studied in *logic programming*. The variables in a logic problem only take Boolean values.

When a problem is modeled as a CSP, one may only require to check if the CSP has at least a solution, not to find solutions explicitly. This requirement belongs to a class of *satisfiability problems*, also called *decision problems*. It is known that the class of general satisfiability problems is NP-complete, but the class of general CSPs NP-hard. This suggests that *problem modeling* is important for efficiency. For clarity, we assume that when a CSP is referred to

without explicitly stating the need for solutions, it will be interpreted as finding one or more solutions of the problem.

The relation among the constraints and variables of a CSP can be partially represented by a graph, which is called the *constraint graph*.

Definition 2.17 (Constraint Graph ≡ Constraint Network). The *constraint graph*, also called the *constraint network*, of a CSP $\mathcal{P}$ is the graph in which each variable of $\mathcal{P}$ is represented by a *node* and in which every two nodes are connected by a unique *arc* if the variables represented by them involves a binary constraint of $\mathcal{P}$. All binary constraints on the same two variables are represented by only one arc.

Many types of graphs have been proposed to represent the relation among the constraints and variables of general CSPs (see [Silaghi 2002, Chapter 5]). The most common one is the *constraint hypergraph* defined as follows.

Definition 2.18 (Constraint Hypergraph). The *constraint hypergraph* of a CSP $\mathcal{P}$ is the graph in which each variable of $\mathcal{P}$ is represented by a *node* and each constraint of $\mathcal{P}$ is represented by a *hyper-arc* connecting the variables involving the constraint.

Several reduced forms of CSPs have been introduced, including the following forms.

Definition 2.19 (Binary CSP, Strictly Binary CSP). If the arity of each constraint in a CSP is at most two then this CSP is called a *binary CSP*. If it only contains binary constraints, then it is called a *strictly binary CSP*.

Definition 2.20 (Normalized CSP). A CSP is said to be *normalized* if, for each subsequence (x, y) of the sequence of its variables, there is at most one constraint on the subsequence (x, y).

Definition 2.21 (Standardized CSP). A CSP is said to be *standardized* if, for each subsequence (x, y) of the sequence of its variables, there exists a unique constraint on the subsequence (x, y).

Definition 2.22 (Regular CSP). A CSP is said to be *regular* if, for each subsequence X of the sequence of its variables, there exists a unique constraint on the subsequence X.

The constraint graphs of standardized binary CSPs and regular CSPs are *complete graphs*. Note that every CSP can always be made normalized, standardized, or regular by adding *universal constraints* and/or replacing multiple constraints on the same set of variables with their intersection. The reader can find more examples and concepts in the books on constraints, such as [Tsang 1993], [Marriott and Stuckey 1998], [Dechter 2003], and [Apt 2003].

2.1.2.2. Set Theory Concepts for Constraints

The negation of a constraint is an extension of the negation of a logic term.

Definition 2.23 (Negation). Let $\mathcal{D} \equiv D_1 \times \cdots \times D_n$ be the domains of a sequence $X \equiv (x_1, \ldots, x_n)$ of variables, respectively. The negation of a constraint C on the sequence X is defined and denoted by $\neg C \equiv \mathcal{D} \setminus C$. The *negation* of a set $\mathcal{C}$ of constraints on the sequence X is the negation of the intersection of its constraints and is denoted by $\neg \mathcal{C}$.

For convenience, we recall the definitions of projection, section, cross section, and slice with their notations adopted in relation to the concept of a constraint.

Definition 2.24 (Projection). Let $\mathcal{D} \equiv D_1 \times \cdots \times D_n$ be the domains of a sequence $X \equiv (x_1, \ldots, x_n)$ of variables, respectively.

- Take a compound label $d \equiv (d_1, \ldots, d_n) \in \mathcal{D}$ of X and a subsequence $Y \equiv (x_{i_1}, \ldots, x_{i_k})$ of X. The sequence $(d_{i_1}, \ldots, d_{i_k})$ is called the *projection* of d on Y, denoted by $d[Y]$.
- For any constraint C on X or any subset $C \subseteq \mathcal{D}$, the set $\{d[Y] \mid d \in C\}$ is called the projection of C on Y and is denoted by $C[Y]$.

Example 2.25. Consider the sequence $V \equiv (x, y, z)$ of variables associated with respective domains $D_x \equiv \{1, 2, 3\}, D_y \equiv \{4, 5, 6\}, D_z \equiv \{7, 8, 9\}$, the compound label $l \equiv (1, 4, 7)$, and the constraint $C(x, y, z) \equiv \{(1, 4, 7), (1, 5, 8), (3, 4, 9)\}$. We have

$$\begin{aligned} l[(x, z)] &= (1, 7), \\ C[y] &= \{4, 5\}, \\ C[(x, y)] &= \{(1, 4), (1, 5), (3, 4)\}. \end{aligned}$$

♣

Example 2.26. Consider the sequence $V \equiv (x, y, z)$ of real variables associated with respective domains $D_x = D_y = D_z \equiv [-2, 2]$ and the constraint $C(x, y, z) \equiv \{(a, b, c) \in \mathbb{R}^3 \mid a^2 + b^2 + c^2 = 1\}$. We have

$$\begin{gathered} C[x] = C[y] = C[z] = [-1, 1], \\ C[(x, y)] = C[(x, z)] = C[(y, z)] = \{(a, b) \in \mathbb{R}^2 \mid a^2 + b^2 \leq 1\}. \end{gathered}$$

The projection of a continuum set is often a continuum set in a lower-dimensional space. ♣

Definition 2.27 (Section, Cross Section). Let $X \equiv (x_1, \ldots, x_n)$ be a sequence of variables, C a constraint on X, and $\mathcal{C}$ a finite set of constraints on X. For all $i \in \{1, \ldots, n\}$, an arbitrary value p of x_i. We define that

- The set $C|_{x_i=p} \equiv \{(c_1, \ldots, c_n) \in C \mid c_i = p\}$ is called the *section of* C *at* $x_i = p$;
- The set $\mathcal{C}|_{x_i=p} \equiv \{S|_{x_i=p} \mid S \in \mathcal{C}\}$ is called the *section of* $\mathcal{C}$ *at* $x_i = p$;
- The set $C|^*_{x_i=p} \equiv \{(c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n) \mid (c_1, \ldots, c_{i-1}, p, c_{i+1}, \ldots, c_n) \in C\}$ is called the *cross section of* C *at* $x_i = p$;
- The set $\mathcal{C}|^*_{x_i=p} \equiv \{S|^*_{x_i=p} \mid S \in \mathcal{C}\}$ is called the *cross section of* $\mathcal{C}$ *at* $x_i = p$.

Example 2.28. Consider the sequence V and the constraint C in Example 2.25. We have

$$\begin{aligned} C|_{x=1} &= \{(1,4,7),(1,5,8)\},\\ C|^*_{x=1} &= \{(4,7),(5,8)\}.\\ C|_{x=2} = C|_{x=2004} &= \emptyset. \end{aligned}$$

♣

Example 2.29. Consider the sequence V and the constraint C in Example 2.26. We have

$$\begin{aligned} C|_{x=0} &= \{(0,a,b) \mid a^2+b^2=1\},\\ C|_{y=0} &= \{(a,0,b) \mid a^2+b^2=1\},\\ C|_{z=1} &= \{(0,0,1)\},\\ C|^*_{x=1} = C|^*_{y=1} = C|^*_{z=1} &= \{(a,b) \mid a^2+b^2=1\}. \end{aligned}$$

The section of a constraint might be empty, a singleton, or even a continuum. ♣

Definition 2.30 (Slice). Let $X \equiv (x_1,\ldots,x_n)$ be a sequence of variables, C a constraint on X, and $\mathcal{C}$ a finite set of constraints on X. For all $i \in \{1,\ldots,n\}$, an arbitrary set S of values of x_i. We define that

- The set $C|_{x_i \in S} \equiv \{(c_1,\ldots,c_n) \in C \mid c_i \in S\}$ is called the *slice of C in the slot $x_i \in S$*;
- The set $\mathcal{C}|_{x_i \in S} \equiv \{A|_{x_i \in S} \mid A \in \mathcal{C}\}$ is called the *slice of $\mathcal{C}$ in the slot $x_i \in S$*.

Example 2.31. Consider the sequence V and the constraint C in Example 2.25. We have

$$\begin{aligned} C|_{x\in[1,2]} = C|_{x\in\{1,2\}} &= \{(1,4,7),(1,5,8)\},\\ C|_{x\in[0,5]} = C|_{x\in\{0,5\}} &= \{(1,4,7),(1,5,8),(3,4,9)\},\\ C|_{x\in\emptyset} = C|_{x=\{4,5,6\}} &= \emptyset. \end{aligned}$$

Note that a slice of a constraint can be empty. ♣

Inspired by the concepts of a section and a cross section (Definition 2.27), and the approaches of *singleton consistency* (Section 2.3.4.3) and kB-*consistency* (Section 2.3.3.2), we define the following notation for an existing concept.[1]

Definition 2.32 (Singleton CSP ≡ Section, Cross Section). Consider a CSP $\mathcal{P} \equiv (\mathcal{V},\mathcal{D},\mathcal{C})$. Let D_x be the domain of a variable x in $\mathcal{V}$. The *singleton CSP of $\mathcal{P}$ w.r.t. $x = a$*, also called the *section of $\mathcal{P}$ at $x = a$*, is defined as

$$\mathcal{P}|_{x=a} \equiv (\mathcal{V},\ \mathcal{D}|_{x=a},\ \mathcal{C}|_{x=a}). \tag{2.1}$$

The *cross section of $\mathcal{P}$ at $x = a$* is defined as

$$\mathcal{P}|^*_{x=a} \equiv (\mathcal{V}\setminus\{x\},\ \mathcal{D}\setminus\{D_x\},\ \mathcal{C}|^*_{x=a}), \tag{2.2}$$

where $\mathcal{D}\setminus\{D_x\}$ denotes the sequence of domains obtained from $\mathcal{D}$ by removing D_x.

[1] It is only for convenience because there has been no standard terminology for this concept so far.

Definition 2.33 (Slice of CSP). Consider a CSP $\mathcal{P} \equiv (\mathcal{V}, \mathcal{D}, \mathcal{C})$. Let D_x be the domain of a variable x in $\mathcal{V}$. The *slice of* $\mathcal{P}$ *in the slot* $x \in S$, where S is an arbitrary set of values of x, is defined as

$$\mathcal{P}|_{x \in S} \equiv (\mathcal{V},\ \mathcal{D}|_{x \in S},\ \mathcal{C}|_{x \in S}). \tag{2.3}$$

Example 2.34. Consider the CSP $\mathcal{P} \equiv \langle x^2 + y^2 + z^2 = 1;\ (x, y, z) \in [-2, 2]^3 \rangle$. We have

$$\begin{aligned}
\mathcal{P}|_{x=0} &= \langle y^2 + z^2 = 1;\ x \in \{0\},\ (y, z) \in [-2, 2] \times [-2, 2] \rangle, \\
\mathcal{P}|^*_{x=0} &= \langle y^2 + z^2 = 1;\ (y, z) \in [-2, 2] \times [-2, 2] \rangle, \\
\mathcal{P}|_{x \in [0,1]} &= \langle x^2 + y^2 + z^2 = 1;\ x \in [0, 1],\ (y, z) \in [-2, 2] \times [-2, 2] \rangle, \\
\mathcal{P}|_{x \in \{0,1,3\}} &= \langle x^2 + y^2 + z^2 = 1;\ x \in \{0, 1\},\ (y, z) \in [-2, 2] \times [-2, 2] \rangle, \\
\mathcal{P}|_{x \in \{0,3\}} &= \langle y^2 + z^2 = 1;\ x \in \{0\},\ (y, z) \in [-2, 2] \times [-2, 2] \rangle.
\end{aligned}$$

Note that the number of variables of a CSP is constant under slicing. ♣

We also define the concept of a *compact CSP*.

Definition 2.35 (Compact CSP). A CSP $\mathcal{P}$ is said to be *compact* if every constraint and every domain of $\mathcal{P}$ is a compact set (see Definition B.5).

2.1.2.3. Basic Concepts of Problem Solving

In Section 1.1, we have already got a general feeling about what constraint programming is. In the previous section, we have also studied some formal definitions of constraints, constraint satisfaction problems and related basic concepts. This section presents an introduction to basic concepts of CSP solving in constraint programming, where solving a CSP means finding one ore more solutions of the CSP. To be precise, here we recall the definition of when a CSP is said to be globally solved [Dechter and van Beek 1997].

Definition 2.36 (Globally Solved Form). A CSP is said to be in *globally solved form* if either it has been proved to be inconsistent or there is a known ordering of the variables along which solutions can be assembled without encountering deadends.

A globally solved form of a CSP is a useful representation of all solutions whenever such a representation is more compact than the explicit representation of set of all solutions. For example, CSPs with inequality constraints on continuous domains often have continuums of solutions. Therefore, finding a globally solved form is more practical than representing the whole solution set explicitly.

To solve CSPs, one often transforms them in some way until one or more solutions have been found or it has been proved that no solution exists. Ideally, the transformations of CSPs need to preserve the equivalence of CSPs in some sense. This leads to the following formal definition of the equivalence.

Definition 2.37 (Problem Equivalence). A CSP $\mathcal{P}_0$ and the union of k CSPs $\mathcal{P}_1, \ldots, \mathcal{P}_k$ are said to be *equivalent w.r.t. a sequence X of variables* if

(i) for every $i \in \{0, 1, \ldots, k\}$, X is a subsequence of the sequence of variables of $\mathcal{P}_i$;

(ii) for every solution s of $\mathcal{P}_0$, there exists a solution s' of some $\mathcal{P}_i$, where $i \in \{1, \ldots, k\}$, such that $s[X] = s'[X]$;

(iii) for every solution s' of every $\mathcal{P}_i$, where $i \in \{1, \ldots, k\}$, there exists a solution s of $\mathcal{P}_0$ such that $s[X] = s'[X]$.

In addition to that, if the sequence of variables of $\mathcal{P}_i$ is equal to X for all $i \in \{0, 1, \ldots, k\}$, we just say that $\mathcal{P}_0$ and the union of $\mathcal{P}_1, \ldots, \mathcal{P}_k$ are *equivalent*.

If the problem equivalence is difficult to achieve, one may also relax the problem to get problems that have all solutions of the initial one and that are hopefully easier to solve. The following property defines this concept.

Definition 2.38 (Problem Relaxation). We say that a CSP $\mathcal{P}_0$ is *relaxed to* the union of k CSPs $\mathcal{P}_1, \ldots, \mathcal{P}_k$ or that the union of k CSPs $\mathcal{P}_1, \ldots, \mathcal{P}_k$ is a *relaxation system* (hence, each constraint in it is called a *relaxation*) of $\mathcal{P}_0$ w.r.t. a sequence X of variables, if

(i) for every $i \in \{0, 1, \ldots, k\}$, X is a subsequence of the sequence of variables of $\mathcal{P}_i$;

(ii) for every solution s of $\mathcal{P}_0$, there exists a solution s' of some $\mathcal{P}_i$, where $i \in \{1, \ldots, k\}$, such that $s[X] = s'[X]$.

In addition to that, if the sequence of variables of $\mathcal{P}_i$ is equal to X for all $i \in \{0, 1, \ldots, k\}$, we just say that $\mathcal{P}_0$ is *relaxed to* the union of $\mathcal{P}_1, \ldots, \mathcal{P}_k$ or that the union of $\mathcal{P}_1, \ldots, \mathcal{P}_k$ is a *relaxation system* of $\mathcal{P}_0$.

A transformation of a CSP $\mathcal{P}_0$ into the union of k CSPs $\mathcal{P}_1, \ldots, \mathcal{P}_k$ written as

$$\mathcal{P}_0 \mapsto \mathcal{P}_1 \cup \cdots \cup \mathcal{P}_k. \tag{2.4}$$

Definition 2.39 (Equivalence-Preserving). Consider the transformation (2.4). It is said to be *equivalence-preserving* (respectively, *equivalence-preserving w.r.t. a sequence X of the variables*) if $\mathcal{P}_0$ and the union of $\mathcal{P}_1, \ldots, \mathcal{P}_k$ are equivalent (respectively, equivalent w.r.t. X).

The equivalence-preserving property is desired for finding solutions of CSPs. It is easy to see that if a transformation is equivalence-preserving w.r.t. a sequence X of variables then it is also equivalence-preserving w.r.t. any subsequence of X. In some solving processes, one may only need to verify if CSPs are consistent, not to find solutions explicitly. For this purpose, the equivalence-preserving property relaxes to the following.

Definition 2.40 (Consistency-Preserving). Consider the transformation in (2.4). It is said to be *consistency-preserving* if the following holds

$$\mathcal{P}_0 \text{ is consistent } \Leftrightarrow \exists i \in \{1, \ldots, k\} : \mathcal{P}_i \text{ is consistent.}$$

The equivalence-preserving property clearly implies the consistency-preserving property. If preserving the equivalence or consistency is difficult to be obtained, one may consider the following properties, respectively.

Definition 2.41 (Relaxing). Consider the transformation in (2.4). It is said to be *relaxing* (respectively, *relaxing w.r.t.* a sequence X of variables) if the union of $\mathcal{P}_1, \ldots, \mathcal{P}_k$ is a relaxation system of $\mathcal{P}_0$ (respectively, w.r.t. X).

Definition 2.42 (Consistency-Relaxing). Consider the transformation in (2.4). It is said to be *consistency-relaxing* (respectively, *consistency-relaxing w.r.t.* a sequence X of variables) if the following holds

$$\mathcal{P}_0 \text{ is consistent} \Rightarrow \exists i \in \{1, \ldots, k\} : \mathcal{P}_i \text{ is consistent.}$$

During solving a CSP, one may also need to eliminate a *singleton variable* or a value or a variable by substituting the variable with its value. We formalize the *variable elimination* under the transformation notation

$$\mathcal{P} \equiv \langle \mathcal{C};\ \mathcal{V} \in \mathcal{D} \rangle \mapsto \mathcal{P}|^*_{x=a} \equiv \langle \mathcal{C}|^*_{x=a};\ \mathcal{V} \setminus \{x\} \in \mathcal{D} \setminus \{D_x\} \rangle, \tag{2.5}$$

where $\mathcal{P}|^*_{x=a}$ denotes the cross section of $\mathcal{P}$ at $x = a$ (see Definition 2.32). The variable elimination (2.5) is equivalence-preserving w.r.t. the sequence of variables $\mathcal{V} \setminus \{x\}$, provided that the domain of x is $\{a\}$. We can also see that the singleton CSP $\mathcal{P}|_{x=a}$ and the cross section $\mathcal{P}|^*_{x=a}$ are equivalent w.r.t. $\mathcal{V} \setminus \{x\}$.

One of the most fundamental techniques is called the *domain reduction*. With the transformation notation, the domain reduction can be written as

$$\langle \mathcal{C};\ \mathcal{V} \in \mathcal{D} \rangle \mapsto \langle \mathcal{C};\ \mathcal{V} \in \mathcal{D}' \rangle, \tag{2.6}$$

where $\mathcal{D}' \subseteq D$. This transformation is equivalence-preserving if no compound label of $\mathcal{V}$ in $\mathcal{D} \setminus \mathcal{D}'$ satisfies $\mathcal{C}$. Let denote $\mathcal{C}_\cap = \bigcap_{C \in \mathcal{C}} C$, then it is the solution set of $\mathcal{P}$.

Definition 2.43 (Domain Reduction Operator). Let ϕ be the function that maps $\mathcal{D}$ to $\mathcal{D}' = \phi(\mathcal{D})$ in (2.6), then it is called a *domain reduction operator*.

Property 2.44. *Four properties are often desired for a domain reduction operator ϕ:*

(*contractiveness*)	$\phi(\mathcal{D}) \subseteq \mathcal{D}$,
(*correctness*)	$\phi(\mathcal{D}) \supseteq \mathcal{D} \cap \mathcal{C}_\cap$,
(*monotonicity*)	$\mathcal{D}_1 \subseteq \mathcal{D}_2 \Rightarrow \phi(\mathcal{D}_1) \subseteq \phi(\mathcal{D}_2)$,
(*idempotence*)	$\phi(\phi(\mathcal{D})) = \phi(\mathcal{D})$. ✠

The correctness is sometimes called the *completeness*. Here we give a simple example on the domain reduction rules for inequality constraints on real numbers:

$$\langle x \le y;\ x \in [\underline{x}, \overline{x}],\ y \in [\underline{y}, \overline{y}] \rangle \mapsto \langle x \le y;\ x \in [\underline{x}, \min\{\overline{x}, \overline{y}\}],\ y \in [\max\{\underline{y}, \underline{x}\}, \overline{y}] \rangle.$$

In general, domain reduction techniques attempt to remove redundant values. More generally, problem reduction techniques attempt to remove compound labels from variable domains. Where values or compound labels are considered redundant if they conform to the following.

Definition 2.45 (Redundant Value/Compound Label). A value in a variable domain of a CSP is said to be *redundant* if it is not the projection of any solution of the CSP. Similarly, a compound label of a subset of variables is said to be *redundant* if it is not the projection of any solution of the CSP.

The solving process, especially in the domain reduction techniques, may also leads to the case that some constraints are always satisfied with all combinations of values remaining in variable domains. In this case, it is not necessary to take these constraints into account. In other words, they are redundant in the following sense.

Definition 2.46 (Redundant Constraint ≡ Implied Constraint). A constraint C of a CSP $\mathcal{P}$ is said to be *redundant w.r.t.* (or *implied by*) the other constraints of $\mathcal{P}$ if the CSP obtained from $\mathcal{P}$ by removing C is equivalent to $\mathcal{P}$.

One may introduce auxiliary variables to simplify the form of constraints. This transformation is called a *variable introduction*. For example,

$$\langle \mathcal{C},\ e_1 \diamond e_2;\ \mathcal{V} \in \mathcal{D} \rangle \mapsto \langle \mathcal{C},\ x_{\text{new}} = e_1,\ x_{\text{new}} \diamond e_2;\ \mathcal{V} \in \mathcal{D},\ x_{\text{new}} \in D_x \rangle, \tag{2.7}$$

where $\diamond$ can be any relation (e.g., $<$, $\leq$, $>$, $\geq$, $=$, $\neq$); e_1 and e_2 are suitable expressions; x_{new} is an *introduced variable* that differs to any variable in $\mathcal{D}$; and the domain D_x of x_{new} is a set containing all possible values of e_1. This variable introduction is equivalence-preserving w.r.t. $\mathcal{V}$, provided that $x_{\text{new}} \diamond e_2$ is defined for all $x_{\text{new}} \in D_x$. Many variable introductions can also be defined in a similar way. For example, adding a *slack variable* converts a real inequality constraint into a real equality constraint with the equivalence-preserving property:

$$\langle \mathcal{C},\ e_1 \leq e_2;\ \mathcal{V} \in \mathcal{D} \rangle \mapsto \langle \mathcal{C},\ e_1 + x_{\text{new}} = e_2;\ \mathcal{V} \in \mathcal{D},\ x_{\text{new}} \in [0, +\infty] \rangle. \tag{2.8}$$

In this thesis, the rule (2.7) is used to construct *DAG representations* of CSPs (see Section 6.2). The rule (2.8) is used to convert between several forms of CSPs (see Section 2.1.3.2).

One may also add relaxations to a CSP to obtain an equivalent CSP. For example,

$$\mathcal{P} \equiv \langle \mathcal{C};\ \mathcal{V} \in \mathcal{D} \rangle \mapsto \langle \mathcal{C},\ \mathcal{C}';\ (\mathcal{V}, \mathcal{V}') \in (\mathcal{D}, \mathcal{D}') \rangle, \tag{2.9}$$

where $\mathcal{R} \equiv \langle \mathcal{C}';\ (\mathcal{V}, \mathcal{V}') \in (\mathcal{D}, \mathcal{D}') \rangle$ is a relaxation system of $\mathcal{P}$. One often aims at producing $\mathcal{C}'$ such that it is as simple as possible and the solution set of $\mathcal{R}$ is close to that of $\mathcal{P}$, and then applies efficient techniques to $\mathcal{R}$ in order to get a hopefully tight enclosure of the solution set of $\mathcal{P}$. See Section 3.3 for more details on relaxation methods.

To measure the power of solution algorithms, it is helpful to give formal definitions on the rigor of solution algorithms.

Definition 2.47 (Soundness). A solution algorithm is called *sound* if every result returned by it is a solution. In this case, the returned solution is said to be *sound*.

Intuitively, an algorithm is said to be *complete* if it can find every solution [Tsang 1993, Definition 2.2]. Apt [2003, p. 82] pointed out that this definition is imprecise and gave a definition based on the proof theory, which depends on type of constraints. His definition is based on globally solved forms (Definition 2.36). For simplicity, we accept the following definition on the completeness of algorithms for solving CSPs, including the ones with continuous domains (see also [Neumaier 2004, p. 5]).

Definition 2.48 (Completeness, Incompleteness). An algorithm is said to be *complete* if every solution can be found by it in the infinite time and approximated by it within an arbitrarily small positive tolerance after a finite time, provided that the underlying arithmetic is exact. An algorithm is said to be *incomplete* if it is not complete.

Nearly complete algorithms are also of interest; hence, this leads to the following definition.

Definition 2.49 (Asymptotical Completeness). An algorithm is said to be *asymptotically complete* if it is not complete and every solution can be found by it with certainty or probability one in the infinity time, provided that the underlying arithmetic is exact.

The arithmetics implemented in present computers are finite precision and their arithmetic results are rounded, and hence are not as exact as expected in the theory. Therefore, it is useful to take this fact into account. That leads to the definition of rigorous algorithms.

Definition 2.50 (Rigor). An algorithm is said to be *rigorous* if it is complete and every solution can be approximated, after a finite time, by it within the smallest computer-representable positive tolerance containing the exact solution, provided that the underlying arithmetic is finite precision and allows controlling rounding errors.

For CSPs with integer domains, the tolerances and rounding errors are usually not allowed; therefore, the rigor resembles to the completeness. In general, the algorithms can be categorized into four classes with increasing degrees of mathematical rigor: incomplete algorithms, asymptotical complete algorithms, complete algorithms, and rigorous algorithms.

The Solving Process. Apt [2003] described a generic process of problem solving as depicted in Algorithm 2.1, where a solution algorithm contains five elementary procedures:

1. Preprocess: This is to transform the considered CSP into a desired syntactic form. The transformation should be equivalence-preserving w.r.t. the initial sequence of variables. Usually, this process is applied only once, at the top level of the solving process.
2. Happy: This means the goal set for initial CSP has been achieved. For example, that often holds when:

Algorithm 2.1: The **Solution** algorithm – a general solving process

```
CONTINUE := true;
while CONTINUE and not Happy do
    Preprocess;
    Problem Reduction;
    if not Happy then
        if Atomic then
            CONTINUE := false;
        else
            Split;
            Proceed by Cases;
        end
    end
end
```

- a solution has been found;
- all solutions have been found;
- a globally solved form has been obtained;
- the inconsistency has been detected.

3. Atomic: This procedure checks if it is amenable for splitting the current CSP into smaller CSPs, often amounting to checking the sizes of domains.

4. Split: This procedure transforms the current CSP $\mathcal{P}$ into multiple CSPs. The equivalence-preserving property is required for completeness. In practice, it often amounts to splitting a domain or a constraint. Some typical instances of Split are as follows:

   - *Enumeration*: a value of a variable is taken into account. That is, the domain of the considered variable is split in two parts. The first considered part contains only one value. This procedure is often used for discrete domains.
   - *Labeling*: multiple values are taken into account at the same time. This procedure is often used for discrete domains.
   - *Bisection*: a domain is split into two subdomains. This is often used for continuous domains. Multiple bisections can be sequenced into a more complicated split.
   - *Cell subdivision*: every domain is split into subdomains simultaneously. This is equivalent to multiple bisections.

5. Proceed by Cases: Since Split yields multiple CSPs, we have to select each yielded CSP for further considerations.

This framework for solving CSPs is not the only one, but the most typical one, especially for complete algorithms. See [Apt 2003, Section 3.2] for more details.

2.1.2.4. Major Solution Approaches

There are several general approaches to solve CSPs, including two major ones: *problem reduction* and *search*. Although the problem reduction techniques alone do not produce solutions, in general, it can be extremely useful when used together with other methods, such as search techniques. The reader can find in [Tsang 1993] more approaches, such as *solution synthesis*.

Problem Reduction. *Problem reduction* plays a very significant role in CSP solving. Basically, it transforms a CSP $\mathcal{P}$ into CSPs, of which the union is equivalent to $\mathcal{P}$ and hopefully easier to process, by reducing the size of domains and by changing the set of constraints. The most popular class of problem reduction techniques is *domain reduction*, which can be represented by a transformation (2.6). Removing redundant compound labels or redundant constraints is called *redundant constraint elimination* or *redundant constraint removal*. It is often done by *constraint propagation*, which is to propagate the impact of constraints through some structure representing the considered problem, for reducing the problem. Upon termination, a constraint propagation technique usually achieves a property called a *local consistency* (see Section 2.3.2). Constraint propagation plays a central role in constraint programming.

Note that problem reduction does not necessarily reduce the number of constraints (as in Chapter 4) because adding *relaxations* sometimes leads to a significant reduction of domains after resorting to specialized techniques for relaxations. See Chapter 6 and Chapter 7 for new methods for *interval relaxations* and mixed *interval/linear relaxations*, respectively; see also Section 3.3 for other relaxation methods. All the above mentioned transformations can be combined to make a better effect of problem reduction. There has also been a numerous number of problem reduction techniques devised in other fields (see Section 3.1).

For more details on domain reduction methods for CSPs, see the sections 2.3, 3.2.2.2, 3.2.2.5, 3.2.2.4, 3.2.2.3. See also the chapters 4–7 in [Tsang 1993] and [Apt 2003, Chapter 7] for problem reduction methods, including redundant constraint removal. The reader can also find in [Neumaier 2002] a much more comprehensive survey on the use of Taylor's formula for reducing the domains of optimization problems and CSPs with continuous domains.

Complete Search. Probably, *complete search* is the general approach that attracts most attentions in constraint programming. Traditionally, *complete search algorithms* often attempt, in some sense, to enumerate combinations of compound labels to find solutions. It often used in combination with other methods such as problem reduction techniques. In general, a complete search algorithm repeatedly generates several cases and then consider each case, one by one, until the requirement is satisfied. Hence, a tree structure is generated. This tree is called the *search tree*, each node is called a *search node*. The set of all states at which a search algorithm could possibly arrive is called the *search space*.

Essentially, most complete search techniques for solving CSPs are *backtracking*. A backtracking search starts with the original CSP (corresponding to the root of the search tree), and repeatedly proceeds by descending to a child of the current node of the search tree. This process continues as long as a node is not a leaf. If a leaf is encountered, the search proceeds by moving back to the parent of the leaf. Then the next child, if any, of this parent is selected. If we are interested in finding just one solution, the backtracking search terminates as soon as a solution is found; hence, a leaf is generated. To find all solutions, this process continues until the control is back at the root node and all of its children have been processed. Note that the search tree to be traversed is not given in advance, but it is generated on the fly. A search algorithm is said to be *backtrack-free* if it can always find a solution or prove the inconsistence of the problem without encountering *deadends* (i.e., without the need for moving back to *ancestors*) during the search.

The reader can find *complete search methods* in Section 3.2.1 (for both CSPs with discrete and continuous domains), and in Section 3.1.3 and Section 3.1.6 (for CSPs with continuous domains). The reader can also find in [Neumaier 2004] a comprehensive survey on complete search methods for optimization problems and CSPs with continuous domains.

Incomplete Search. The soundness and completeness are desirable properties of algorithms. However, some real life problems are intractable and cannot be solved by existing complete algorithms in due time. If the response time is crucial, then one may be willing to sacrifice completeness for speed. When this is the case, incomplete search algorithms such as *stochastic search* could be useful. Stochastic search is a class of the incomplete search methods, which includes heuristics and an element of nondeterminism in traversing the *search space* (i.e., the set of states at which it could possibly arrive). Unlike the complete search algorithms, a stochastic search algorithm moves from one point to another in the search space in a nondeterministic manner, guided by heuristics. The next move is partly determined by the outcome of the previous one. In general, stochastic search algorithms are incomplete.

Another important class of the incomplete search methods is *local search*, including popular methods such as *Newton-like methods* and *homotopy/continuation methods*. Local search is more deterministic than stochastic search. In most cases, there is a determined formula to generate the next move in the search space. It is however not complete because it cannot guarantee to find a solution in the general case. Unlike stochastic search, local search may be complete under some assumptions, such as the convexity and monotonicity. In many applications, these assumptions are acceptable; hence, the local search is the best choice. For more details on incomplete search methods, see Section 3.4.

2.1.3. Numerical Constraint Satisfaction Problems

CSPs with continuous domains (e.g., systems of equations and inequalities) are ubiquitous in real world applications. However, most classical methods for solving CSPs with discrete domains (see Section 3.2.1) are not efficient for solving CSPs with continuous domains. Therefore, the class of CSPs with continuous domains needs specialized algorithms. In this thesis, we study solution algorithms for the class of CSPs with continuous domains in the framework of constraint programming, that is, the algorithms that take the interaction of constraints into account explicitly.

2.1.3.1. Numerical Constraints

To be precise, we give some formal definitions related to the class of CSPs with continuous domains (they can be extended to other sets than $\mathbb{R}$).

Definition 2.51 (Continuous Domain). A *continuous domain* is a connected set of real numbers. In other words, a continuous domain is a real interval.

Example 2.52. The domain $D_1 \equiv \{1, 3, 5, 7\}$ is a discrete domain, hence not a continuous one. The domains $D_2 \equiv [-10, 20]$, $D_3 \equiv]3, 25[$, $D_3 \equiv]-\infty, 10]$, $D_4 \equiv [10, +\infty[$ are connected sets of real numbers, hence continuous domains. Conversely, the domain $D_5 \equiv [2, 4] \cup [5, 7]$ is not a connected set of real numbers, hence not a continuous domain. However, it is the union of two continuous domains. ♣

Next, we present the concepts of a variable and a numerical constraint.

Definition 2.53 (Continuous Variable). A *continuous variable* is a variable that is associated with a continuous domain.

Definition 2.54 (Numerical Constraint). A *numerical constraint* is a constraint on a sequence of continuous variables.

In literature, numerical constraints are sometimes referred to as *continuous constraints*. We avoid using this term since it may be confused with a constraint that is defined by using a *continuous function*. We also use the term *numerical constraint satisfaction problem* defined in the next definition, instead of the term *continuous constraint satisfaction problem*.

Definition 2.55 (Numerical CSP ≡ NCSP). A *numerical constraint satisfaction problem*, abbreviated to *NCSP* or *numerical CSP*, is a constraint satisfaction problem $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ where all domains in $\mathcal{D}$ are continuous.

Example 2.56. Consider the CSP $\langle x^2 + y^2 \leq 1;\ x \in [-2, 4],\ y \in [-4, 2]\rangle$. It is a numerical CSP. Its solution set is the unit circle centered at the origin of the coordinate system, including the interior. ♣

Example 2.57. Consider the CSP $\langle x^2 + y^2 = 1;\ x \in [-2, 4],\ y \in [-4, 2]\rangle$. It is a numerical CSP. Its solution set is the unit circle centered at the origin of the coordinate system, without the interior. ♣

Example 2.58. Consider the CSP $\langle x^2 + y^2 = 1,\ y + a = x^2;\ x \in [-2, 4],\ y \in [-4, 2]\rangle$. It is a numerical CSP. If $a = 1$, the solution set is the set of three points $(-1, 0)$, $(0, -1)$, and $(1, 0)$. If $a = 4$, the solution set is empty. ♣

The above examples show that, in general, the solution set of a numerical CSP can be empty, isolated points, a surface, a shape with a nonzero (full rank) volume, or even more complicated. We study the representation of the solution set of a numerical CSP in Chapter 4. Since the variable domains in numerical CSPs are continuums, the classic techniques for solving CSPs with discrete domains cannot apply to numerical CSPs directly, in general. Solving a numerical CSP by simply discretizing its variable domains and then using classic techniques for CSPs with discrete domains are usually inefficient.

2.1.3.2. Problem Formulation

In practice, a numerical constraint satisfaction problem (NCSP) can often be represented in the form

$$f(x) \in \mathbf{b}, \tag{2.10}$$

where x is a vector of n real variables in $\mathbf{x} \in \mathbb{I}^n$, $\mathbf{b} \equiv (\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_m)^{\mathrm{T}}$ is an interval vector in $\mathbb{I}^m$, and $f = (f_1, f_2, \ldots, f_m)^{\mathrm{T}}$ is a function from $D \subseteq \mathbb{R}^n$ to $\mathbb{R}^m$. In practice, each component function f_j is often expressed in factorable form (see Section 2.1.3.3).

Definition 2.59 (Constraint Range). Considered the constraint system of the form (2.10). We call $\mathbf{b}_j$ the *constraint range* of the j-th constraint (i.e., the constraint $f_j(x) \in \mathbf{b}_j$) for $j \in \{1, \ldots, m\}$, and $\mathbf{b}$ the *vector of constraint ranges*.

Let $\mathbf{b}_j = [\underline{b}_j, \overline{b}_j]$ $(j = 1, \ldots, m)$. The constraint system (2.10) consists of m constraints of the form

$$f_j(x) \in [\underline{b}_j, \overline{b}_j]. \tag{2.11}$$

There are four possible cases for such a constraint:

1. If $\mathbf{b}_j$ is thin (i.e., $\underline{b}_j = \overline{b}_j = b_j$) for some $j \in \{1, \ldots, m\}$, the constraint (2.11) is indeed an *equality constraint* (also called an *equation*) of the form

$$f_j(x) = b_j; \tag{2.12}$$

2. If $\underline{b}_j = -\infty$ for some $j \in \{1, \ldots, m\}$, the constraint (2.11) is indeed an *inequality constraint* (also called an *inequality*) of the form

$$f_j(x) \geq \underline{b}_j; \tag{2.13}$$

3. If $\overline{b}_j = +\infty$ for some $j \in \{1, \ldots, m\}$, the constraint (2.11) is indeed an *inequality constraint* (also called an *inequality*) of the form

$$f_j(x) \leq \overline{b}_j; \tag{2.14}$$

4. If $-\infty < \underline{b}_j < \overline{b}_j < +\infty$, the constraint (2.11) is a *two-sided inequality constraint* (also called a *two-sided inequality*) of the form

$$\underline{b}_j \leq f_j(x) \leq \overline{b}_j. \tag{2.15}$$

Example 2.60. Consider the parametric constraint system

$$\langle \sqrt{x_1} + 2\sqrt{x_1 x_2} + 2\sqrt{x_2} \leq 7,\ p \leq x_1^2\sqrt{x_2} - 2x_1x_2 + 3\sqrt{x_2} \leq q;\ x_1 \in [1, 16],\ x_2 \in [1, 16]\rangle.$$

The first constraint is an inequality with the constraint range $[-\infty, 7]$. The second constraint can be either an equality or an inequality, with the constraint range $[p, q]$, depending on the parameters (p, q). For instance, the second constraint is an equality if $(p, q) = (0, 0)$, and it is a two-sided inequality if $(p, q) = (0, 2)$. This CSP can be written as follows

$$\begin{cases} \sqrt{x_1} + 2\sqrt{x_1 x_2} + 2\sqrt{x_2} & \in & [-\infty, 7], \\ x_1^2\sqrt{x_2} - 2x_1x_2 + 3\sqrt{x_2} & \in & [p, q], \end{cases}$$

where $x \equiv (x_1, x_2) \in \mathbf{x} = ([1, 16], [1, 16]) \in \mathbb{I} \times \mathbb{I}$. ♣

It easy to see that the form (2.12) can be rewritten as

$$g_j(x) = 0 \tag{2.16}$$

by defining $g_j(x) = f_j(x) - b_j$. Analogously, the forms (2.13) and (2.14) can also be rewritten in the form

$$g_j(x) \leq 0. \tag{2.17}$$

Moreover, the inequality form (2.17) can be converted into the equality form (2.16) by adding a new *slack variable* $s_j \in [0, +\infty]$:

$$g_j(x) + s_j = 0. \tag{2.18}$$

Inversely, the equality form (2.16) can also be converted into the inequality form (2.17) by replacing it with two inequalities

$$\begin{cases} g_j(x) & \leq & 0, \\ -g_j(x) & \leq & 0. \end{cases} \tag{2.19}$$

Consequently, we can assume, without loss of generality, that the input problem has either the equality form

$$f(x) = 0 \tag{2.20}$$

or the inequality form

$$f(x) \leq 0, \tag{2.21}$$

where f is a function from $D \subseteq \mathbb{R}^n$ to $\mathbb{R}^m$ and x is a vector of n real variables.

Although the equality form (2.20) and the inequality form (2.21) are equivalent in theory, they usually have different meanings in practice. A system of equations of the form (2.20) arising in real world applications usually has, but not always, isolated solutions if $n = m$. A system of the form (2.21) or of the form (2.20) with $m < n$ often has continuums of solutions, in practice. The former are called a *well-constrained problem* and the latter are called an *under-constrained problem*.

2.1.3.3. Factorable Form

In practice, most explicit functions used for modeling real world applications can be composed of elementary operations/functions such as $+$, $-$, $*$, $/$, sqr, exp, ln, and sin. These function are called *factorable functions*. They play a very significant role in algorithms for solving not only numerical CSPs but also other numerical problems, such as optimization problems and automatic differentiation computations. For this reason, we recall in this section the concepts of *factorability*. For convenience, we define two sets of standard elementary operations.

Notation 2.61 (Elementary Operations). Denote by $\mathbb{E}_1$ the set of standard *elementary unary functions*, namely, $\mathbb{E}_1 = \{\text{abs}, \text{sqr}, \text{sqrt}, \exp, \ln, \sin, \cos, \arctan\}$. Denote by $\mathbb{E}_2$ the set of standard *elementary binary operation*, namely, $\mathbb{E}_2 = \{+, -, *, /, \hat{}\}$.

If an expression is recursively composed of the standard elementary operations and functions, it is called an *arithmetic expression* [Neumaier 1990, p. 13] (it is originally called a *factorable expression* in [McCormick 1976, 1983]).

Definition 2.62 (Arithmetic Expression). Let $\mathbb{E}_1$ be the set of standard elementary unary functions and $\mathbb{E}_2$ the set of standard elementary binary operations. An *arithmetic expression* in the (formal) variables $x_1, \ldots, x_n$ is a member of the set $\mathcal{E} \equiv \mathcal{E}(x_1, \ldots, x_n)$ satisfying the *composition* rules:

(*i*) $\mathbb{R} \subseteq \mathcal{E}$;
(*ii*) $x_i \in \mathcal{E}$ for $i = 1, \ldots, n$;
(*iii*) $e_1, e_2 \in \mathcal{E} \Rightarrow e_1 \diamond e_2 \in \mathcal{E}$ for all $\diamond \in \mathbb{E}_2$;
(*iv*) $e \in \mathcal{E} \Rightarrow f(e) \in \mathcal{E}$ for all $f \in \mathbb{E}_1$.

We extend the concept of an arithmetic expression to include other elementary operations.

Definition 2.63 (Factorable Expression). Let R be a nonempty set; $x_1, \dots, x_n$ the variables taking values in R; F a finite set of *elementary operations* of the form $f : R^k \to R$. An expression is said to be *factorable* in the (formal) variables $x_1, \dots, x_n$ by using operations in F if it is a member of the set $\mathcal{F} \equiv \mathcal{F}(R, F; x_1, \dots, x_n)$ satisfying the *composition* rules:

(i) $R \subseteq \mathcal{F}$;
(ii) $x_i \in \mathcal{F}$ for $i = 1, \dots, n$;
(iii) If $f : R^k \to R$ is an operation in F and $e_1, \dots, e_k \in \mathcal{F}$; then $f(e_1, \dots, e_k) \in \mathcal{F}$.

We have that $\mathcal{E}(x_1, \dots, x_n) \equiv \mathcal{F}(\mathbb{R}, \mathbb{E}_1 \cup \mathbb{E}_2; x_1, \dots, x_n)$, where $\mathbb{E}_1$ and $\mathbb{E}_2$ are the sets of standard elementary operations as defined in Notation 2.61.

Note 2.64. If an expression E is factorable in the variables $X \equiv \{x_1, \dots, x_n\}$ by using operations in F as defined in Definition 2.63 and if either $F = \mathbb{E}_1 \cup \mathbb{E}_2 \wedge R = \mathbb{R}$ holds or we do not care about F for the moment, then we say that E is factorable (in X), for short.

Example 2.65. The expression given by $f(x, y) = 2x^y + \sin x$ is factorable by using elementary operations in $\{+, *, \hat{}, \sin\}$. The recursive composition is given as follows:

$$\begin{aligned} f_1 &\equiv x \hat{} y \; (\equiv x^y), \\ f_2 &\equiv 2 * f_1, \\ f_3 &\equiv \sin(x), \\ f &\equiv f_2 + f_3. \end{aligned}$$

The expression $f(x, y)$ is also an arithmetic expression, namely, in $\mathcal{E}(x, y)$. ♣

Definition 2.66 (Factorable Function). A function f is said to be *factorable* in the variables $x_1, \dots, x_n$ by using the operations in a finite set F of elementary operations if it can be expressed by an expression that is factorable in the variables $x_1, \dots, x_n$ by using elementary operations in F. If $F = \mathbb{E}_1 \cup \mathbb{E}_2$ or we do not care about F for the moment, we just say f is factorable, for short.

Example 2.67. The function given by the expression $f(x, y) = 2x^y + \sin x$ is factorable using operations in $\{+, *, \hat{}, \sin\}$. In other words, the function $f : \mathbb{R}^2 \to \mathbb{R}$ given by the rule $(x, y) \mapsto 2x^y + \sin x$ is factorable (in the variables x, y) by using operations in $\{+, *, \hat{}, \sin\}$. However, this function is not factorable by using only the operations in $\{+, *, \hat{}\}$. ♣

For convenience, we also define the concept of a *factorable constraint* in a similar way. In the next definition, we only consider the standard relations $\leq, <, \geq, >, =, \neq$, its spirit be easily extended to other relations under

Definition 2.68 (Factorable Constraint). A constraint is said to be *factorable* in the variables $x_1, \dots, x_n$ by using a finite set F of elementary operations if it can be expressed by a standard relation in $\{\leq, <, \geq, >, =, \neq\}$ of two expressions each of which is factorable in the variables $x_1, \dots, x_n$ by using operations in F. In the composition of a factorable constraint, each constraint representing an elementary operation is called a *primitive constraint*.

Example 2.69. The constraint $2x^y + \sin x \leq 0$ is factorable (in the variables x, y) by using operations in $\{+, *, \hat{}, \sin\}$. Its primitive constraints are

$$f_1 = x\hat{}y \ (\equiv x^y),$$
$$f_2 = 2 * f_1,$$
$$f_3 = \sin(x),$$
$$f_2 + f_3 \leq 0.$$

Similarly, the constraint $2x^y + \sin x \leq 2y^x + \cos z$ is also factorable (in the variables x, y, z) by using operations in $\{+, *, \hat{}, \sin, \cos\}$. ♣

Finally, we define the concept of a factorable CSP for convenience in terminology.

Definition 2.70 (Factorable CSP). A CSP is said to be *factorable* (by using a set F of elementary operations) if all its constraints are factorable (by using operations in F).

2.1.3.4. Separable Form

Many techniques have been developed for functions that can be expressed as a sum of univariate functions. These functions, called *separable functions*, play an important role in numerical algorithms because the computational complexity of the algorithms which require the separability is often better than that of the general algorithms which require the factorability only. Kolmogorov [1963] proved a theorem, which is well-known as a complete solution to the thirteen problem of Hilbert, stating that any function can be represented by a superposition of functions of one variable and addition. This proof is however non-constructive. Recently, Yamamura [1993, 1996] proposed a practical way to transform factorable functions into equivalent separable functions by introducing new auxiliary variables when composing the functions. This makes it possible to solve factorable systems by resorting to specialized techniques for separable functions (see some examples in [Kolev 1998, 1999, 2001, 2002] or in Section 3.3.3). For this reason, we recall hereafter the concepts of *separability*.

Definition 2.71 (Separable Expression). An expression f in the (formal) variable $x_1, \ldots, x_n$ is said to be *separable* if it can be expressed as $f = \sum_{k=1}^{n} f_k$, where f_k is an expression in the only variable x_k, for $k = 1, \ldots, n$.

Example 2.72. The expression $(x - y)^2 + e^y + (x + y)^2 + \sin x$ is separable because it can be expressed by the sum of two univariate expressions: $2x^2 + \sin x$, $2y^2 + e^y$. ♣

The concept of a separable function is defined similarly to that of a separable expression.

Definition 2.73 (Separable Function). A function f in the (formal) variable $x_1, \ldots, x_n$ is said to be *separable* if it can be expressed as $f(x_1, \ldots, x_n) = \sum_{k=1}^{n} f_k(x_k)$, where f_k is a function in the only variable x_k, for $k = 1, \ldots, n$.

Example 2.74. The function $(x, y) \mapsto (x - y)^2 + e^y + (x + y)^2 + \sin x$ is separable because it can be expressed by the sum of two univariate functions: $x \mapsto 2x^2 + \sin x$ and $y \mapsto 2y^2 + e^y$. ♣

For convenience, we also define the concepts of a separable constraint and a separable CSP in a similar way. In this thesis, we only consider the standard relations in $\{\leq, <, \geq, >, =, \neq\}$. However, these two concepts can be easily extended for other relations.

Definition 2.75 (Separable Constraint). A constraint is said to be *separable* if it can be expressed by a standard relation in $\{\leq, <, \geq, >, =, \neq\}$ of two separable expressions.

Definition 2.76 (Separable CSP). A CSP is said to be *separable* if all its constraint is factorable.

2.1.3.5. Ternary Form

Every factorable expression can be converted into ternary expressions by repeatedly replacing each binary arithmetic subexpression with an auxiliary variable until the arity of all the resulting expressions is at most three. An expression of arity less than three are made ternary by adding one or two dummy variables, if needed.

Example 2.77. The expression $E = (x^5 + xy^3)/y^2\sqrt{z}$ can be converted into the ternary form as follows:

$$u = xy^3, \qquad v = y^2\sqrt{z},$$
$$t = x^5 + u, \qquad E = t/v.$$

The resulting system contains four new expressions on three original variables (x, y, z) and three auxiliary variables (u, v, t). ♣

The domains of auxiliary variables can be obtained by evaluating the ranges of expressions defining the variables. Each resulting expression in the ternarization can be made simpler by introducing more auxiliary variables. The worse case is to factorize an expression into primitive operations that define the factorable form, as shown in Example 2.65. Since factorable constraints are made of factorable expressions, every factorable CSPs can be converted into ternary factorable CSPs by introducing some auxiliary variables. The generated ternary constraints are often small in the number of elementary operations.

Decomposing general factorable constraints into constraints of the ternary form makes it possible to represent the continuum solution set of an NCSP concisely, provided that the *ternarized constraints* are *convex* (see Section 2.3.4.2). The reader can find more details of *ternarization*-based methods in [Sam-Haroud 1995, Chapter 5], [Sam-Haroud and Faltings 1996], [Faltings and Gelle 1997; Gelle 1998; Gelle and Faltings 2001], [Lottaz 1999, 2000].

2.2. Common Arithmetics for Numerical Computations

2.2.1. Floating-Point Numbers and IEEE 754 Standard

In this section, we only give a few important properties of the *floating-point number system*. The reader can find much more details in [Goldberg 1991] and references therein.

2.2.1.1. Number Representation

The floating-point representation is the most widely used among the representations proposed so far in order to represent real numbers on computers approximately. Some among the representations of the real numbers are: the *signed-digit number system* [Avizienis 1961], the *sign/logarithm number system* [Swartzlander and Alexopoulos 1975], and the *slash number system* [Matula and Kornerup 1985]. The last one was recently improved by several variants of *rational arithmetic* [Mencer 2000].

A number in the *floating-point representation*, which has a base β and a precision p, can be written as

$$\pm d_0.d_1 d_2 \ldots d_{p-1} \times \beta^e, \tag{2.22}$$

where $0 \leq d_i < \beta$ (for $i = 0, \ldots, p-1$) and e is an integer within some interval $[e_{\min}, e_{\max}]$. This is called a *floating-point number*, which exactly represents the real number

$$\pm \left(d_0 + d_1 \beta^{-1} + d_2 \beta^{-2} + \cdots + d_{p-1} \beta^{-(p-1)} \right) \beta^e.$$

The part $d_0.d_1 d_2 \ldots d_{p-1}$ is called the *significand* (which was previously called the *mantissa*) and e is called the *exponent*. Throughout this thesis, the term *floating-point number* is used to mean a real number that can be exactly represented in the format under discussion. A floating-point number can be encoded in $\lceil \log_2(e_{\max} - e_{\min} + 1) \rceil + \lceil p \log_2 \beta \rceil + 1$ bits, where a bit is dedicated to the sign.

The floating-point representation is not necessarily unique. For example, both 0.1×10^{-1} and 1.0×10^{-2} represent the same real number 0.01. Therefore, the practical implementations often require the leading digit being nonzero, that is, $d_0 \neq 0$ in the form (2.22). If this is the case, the representation is said to be *normalized*. Requiring that a floating-point representation be normalized makes the representation unique, but makes it impossible to represent zero. Alternatively, zero can be naturally represented by $1.0 \times \beta^{e_{\min}-1}$ because this preserves the fact that the numerical ordering of nonnegative real numbers corresponds to the lexicographic ordering of their floating-point representations. When using k bits to store the exponent, only $2^k - 1$ values are available for use as exponents because one bit must be reserved to represent zero. The smallest (respectively, the greatest) floating-point number greater than x (respectively, less than x) is denoted by denoted by x^+ (respectively, x^-).

2.2.1.2. IEEE 754 Standard and Conventions

There are two different IEEE standards for floating-point computation: IEEE 754 and IEEE 854. IEEE 754 (1985) governs binary *floating-point arithmetic*. It specifies number formats, basic operations, conversions, and exceptional conditions. The related standard IEEE 854 (1987) generalizes IEEE 754 to cover decimal arithmetic as well as binary. Unlike IEEE 745, it does not specify how floating-point numbers are encoded into bits. It also does not require a particular value for p, but instead it specifies constraints on the allowable values of p for single and double precisions. Characteristics of IEEE 754 is given in Table 2.1.

Special quantities have been defined in IEEE 754 to allow correct handling of exceptional situations, such as a division by zero or an evaluation of the square root of a negative number. Table 2.2 gives those special quantities. Unlike the real zero, the floating-point zeros have a sign. Since the leading digit in the significand of a normalized binary floating-point number is equal to 1, the representation of 0 is problematic, as described in Section 2.2.1.1. IEEE 754 defines that $+0 = 1.0 \times \beta^{e_{\min}-1}$ and $-0 = -1.0 \times \beta^{e_{\min}-1}$. The standard imposes that any

Table 2.1. IEEE 754 floating-point formats

Precision	p	$e_{\min}$	$e_{\max}$	Exponent width	Format width
single	24	-126	127	8	32
single ext.	32	≤ -1022	≥ 1023	≥ 11	≥ 43
double	53	-1022	1023	11	43
double ext.	64	≤ -16382	≥ 16383	≥ 15	≥ 79

Table 2.2. IEEE 754 special quantities

Number	Sign	Exponent	Significand
$\pm\infty$	$\pm$	$e_{\max}+1$	$1.00\ldots0$
±0	$\pm$	$e_{\min}-1$	$1.00\ldots0$
denormalized	$\pm$	$e_{\min}-1$	$\neq 1.00\ldots0$
`NaN`	any	$e_{\max}+1$	$\neq 1.00\ldots0$

check on $-0 = +0$ returns `true`. However, these two zeros are distinct. *denormalized numbers* are introduced to allow the representation of smaller numbers than possible with normalized representation at the cost of a reduction in the number of significand digits. For denormalized numbers, the leading digit is assumed to be 0. `NaN` stands for *Not a Number*. It is used to indicate that a result is invalid. For example, the operations $1/0$, $\sqrt{-2}$, $0.0/0.0$, $0 * \infty$, and $((-\infty) + (+\infty))$ are invalid. Any operation involving `NaN` will produce `NaN`.

2.2.2. Interval Arithmetic

In this section, we give a very short introduction to interval arithmetic. The reader can find much more details in Appendix A, especially in the following fundamental books: introduction [Alefeld and Herzberger 1983; Moore 1966, 1979], fundamental interval methods for systems of equations [Neumaier 1990], some recently added applications [Jaulin *et al.* 2001], extended interval methods for optimization problems [Hansen and Walster 2004].

2.2.2.1. Real Intervals

A *real interval* is a connected subset of $\mathbb{R}$.[2] For simplicity, we will call it an *interval* when no confusion may arise. For convenience, we consider the empty set as a special interval, called the *empty interval*. It is to sure that the set of intervals is closed under the set intersection.

Let $\mathbb{R}_\infty \equiv \mathbb{R} \cup \{-\infty, +\infty\}$. The *lower bound* of an interval $\mathbf{x}$ is defined as $\inf(\mathbf{x})$. Similarly, the *upper bound* of $\mathbf{x}$ is defined as $\sup(\mathbf{x})$. Let denote $\underline{x} = \inf(\mathbf{x}) \in \mathbb{R}_\infty$ and $\overline{x} = \sup(\mathbf{x}) \in \mathbb{R}_\infty$. There are four possible intervals $\mathbf{x}$ with these bounds:

- the *closed interval* defined as $\mathbf{x} \equiv [\underline{x}, \overline{x}] \equiv \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\}$;
- the *open interval* defined as $\mathbf{x} \equiv]\underline{x}, \overline{x}[\equiv \{x \in \mathbb{R} \mid \underline{x} < x < \overline{x}\}$;
- the *left-open interval* defined as $\mathbf{x} \equiv]\underline{x}, \overline{x}] \equiv \{x \in \mathbb{R} \mid \underline{x} < x \leq \overline{x}\}$;
- the *right-open interval* defined as $\mathbf{x} \equiv [\underline{x}, \overline{x}[\equiv \{x \in \mathbb{R} \mid \underline{x} \leq x < \overline{x}\}$.

[2] A set is said to be *disconnected* if it can be partitioned into two nonempty subsets such that each subset has no points in common with the *set closure* of the other, otherwise it is called a *connected set*.

The set of all closed intervals is denoted by $\mathbb{I}$ and the set of all intervals is denoted by $\mathbb{I}_\circ$. The *interval hull* of a subset S of $\mathbb{R}$, denoted by $\square S$, is the smallest interval (w.r.t. the set inclusion) that contains S. For example, $\square(]1,3] \cup \{2,4\}) =]1,4]$. Given a nonempty interval $\mathbf{x}$, we can define that

- the *midpoint* of $\mathbf{x}$ is $\operatorname{mid}(\mathbf{x}) \equiv (\inf(\mathbf{x}) + \sup(\mathbf{x}))/2$;
- the *radius* of $\mathbf{x}$ is $\operatorname{rad}(\mathbf{x}) \equiv (\sup(\mathbf{x}) - \inf(\mathbf{x}))/2$;
- the *width* of $\mathbf{x}$ is $\operatorname{w}(\mathbf{x}) \equiv \sup(\mathbf{x}) - \inf(\mathbf{x})$.

For convenience, we define that $\operatorname{rad}(\emptyset) = \operatorname{w}(\emptyset) = 0$ and $\operatorname{mid}(\emptyset) = \emptyset$. The set $\mathbb{I}_\circ$ of all intervals, and hence also the set $\mathbb{I}$, admits the usual *partial orders* (also called *partial ordering*) $\diamond \in \{<, \leq, >, \geq\}$ in the set theory; namely, for every $\mathbf{x}, \mathbf{y} \in \mathbb{I}_\circ$, we define that

$$\mathbf{x} \diamond \mathbf{y} \Leftrightarrow \forall x \in \mathbf{x}, y \in \mathbf{y} : x \diamond y.$$

Definition 2.78 (Interval Vector, Box). An *interval vector* is a vector of intervals. Equivalently, an *interval box* (or a *box* for short) is the Cartesian product of intervals.

If not specified, a vector will be interpreted as a *column vector*. The concept of an *interval box* is important in interval analysis. The above concepts (e.g., the midpoint, radius, and width) are extended to *interval vectors/matrices* in a componentwise manner.

2.2.2.2. Exact Interval Arithmetic

Fundamental Operations. Fundamentally, if $\mathbf{x}$ and $\mathbf{y}$ are two real intervals, then the four elementary operations for *idealized interval arithmetic* obey the rule

$$\mathbf{x} \diamond \mathbf{y} = \{x \diamond y \mid x \in \mathbf{x}, y \in \mathbf{y}\}, \quad \forall \diamond \in \{+, -, *, \div\}. \tag{2.23}$$

Thus, the results of the four elementary interval arithmetic operations are exactly the ranges of their real-valued counterparts. This is the *spirit of interval arithmetic*. Although the rule (2.23) characterizes these operations mathematically, the usefulness of interval arithmetic is due to the *operational definitions* based on interval bounds (a short description can be found in [Hickey *et al.* 2001]). For example, let $\mathbf{x} = [\underline{x}, \overline{x}]$ and $\mathbf{y} = [\underline{y}, \overline{y}]$ be two closed intervals, (standard) interval arithmetic shows that

$$\mathbf{x} + \mathbf{y} \equiv [\underline{x} + \underline{y}, \overline{x} + \overline{y}]; \tag{2.24a}$$
$$\mathbf{x} - \mathbf{y} \equiv [\underline{x} - \overline{y}, \overline{x} - \underline{y}]; \tag{2.24b}$$
$$\mathbf{x} * \mathbf{y} \equiv [\min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}, \max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}]; \tag{2.24c}$$
$$\mathbf{x} \div \mathbf{y} \equiv \mathbf{x} * 1/\mathbf{y} \text{ if } 0 \notin \mathbf{y}, \text{ where } 1/\mathbf{y} \equiv [1/\overline{y}, 1/\underline{y}]. \tag{2.24d}$$

Note that the division is not defined in standard interval arithmetic when denominator contains zero. In this case, one often assumes the result is the *universal interval* $[-\infty, +\infty]$, for convenience and safety. Simple arithmetic expressions are composed of these four fundamental operations. Every operation with an empty set should return an empty set. Note that interval arithmetic suffers from the problem of *dependency*; for example, we have $\mathbf{x} - \mathbf{x} \not\equiv [0, 0]$. Moreover, it has only the *subdistributivity* property: $(\mathbf{x} + \mathbf{y}) * \mathbf{z} \subseteq \mathbf{x} * \mathbf{z} + \mathbf{y} * \mathbf{z}$.

Example 2.79. If given a *real-valued function/expression*, $f(x) \equiv x * (x-1)$. The *natural extension* of this function to interval arithmetic is $\mathbf{f}(\mathbf{x}) = \mathbf{x} * (\mathbf{x} - \mathbf{1})$, where $\mathbf{1} \equiv [1,1] = 1$. For example, this expression can be evaluated at $[0,1]$ as follows

$$\mathbf{f}([0,1]) = [0,1] * ([0,1] - \mathbf{1}) = [0,1] * [-1,0] = [-1,0].$$

Note that this results contains the exact range $f([0,1]) = [-0.25, 0]$. ♣

Unary Operations. Other elementary operations of the form $\psi : D \subseteq \mathbb{R} \to \mathbb{R}$ can also be extended to intervals. This is usually done by attempting to follow the spirit of interval arithmetic. Namely, it would be ideal if we could define

$$\psi(\mathbf{x}) \equiv \Box\{\psi(x) \mid x \in \mathbf{x}\},$$

whenever $\mathbf{x} = [\underline{x}, \overline{x}] \subseteq D$. For example, following the *spirit of interval arithmetic*, we can define the *square* operation in interval arithmetic:

$$\mathrm{sqr}(\mathbf{x}) \equiv \mathbf{x}^2 \equiv \begin{cases} [\min\{\underline{x}^2, \overline{x}^2\}, \max\{\underline{x}^2, \overline{x}^2\}] & \text{if } 0 \notin [\underline{x}, \overline{x}]; \\ [0, \max\{\underline{x}^2, \overline{x}^2\}] & \text{otherwise.} \end{cases} \tag{2.25}$$

Similarly, the *square root* operation in interval arithmetic is defined as

$$\mathrm{sqrt}(\mathbf{x}) \equiv \sqrt{\mathbf{x}} \equiv \begin{cases} \emptyset & \text{if } \overline{x} < 0; \\ [\sqrt{\max\{\underline{x}, 0\}}, \sqrt{\overline{x}}] & \text{otherwise.} \end{cases} \tag{2.26}$$

For a deeper discussion of elementary functions defined on subset of $\mathbb{R}$, see Chapter 5.

2.2.2.3. Rounded Interval Arithmetic

The finite nature of computers precludes an exact representation of the real numbers. In practice, the real set, $\mathbb{R}$, is therefore approximated by a finite set $\mathbb{F}_\infty \equiv \mathbb{F} \cup \{-\infty, +\infty\}$, where $\mathbb{F}$ is the set of floating-point numbers (see Section 2.2.1). The set of real intervals is then approximated by the set, $\mathbb{I}_\diamond$, of closed *floating-point intervals* with bounds in $\mathbb{F}_\infty$. The concept of an interval hull is extend to that of an $\mathbb{F}$-hull, the smallest interval box in $\mathbb{I}_\diamond^n$ containing $S \subseteq \mathbb{R}^n$, and is denoted by $\Box^{\mathbb{F}} S$. The other concepts are also extended w.r.t. the inclusion property. For example, in $\mathbb{I}_\diamond$, we do not have the identity $\mathbf{x} = [\mathrm{mid}(\mathbf{x}) - \mathrm{rad}(\mathbf{x}), \mathrm{mid}(\mathbf{x}) + \mathrm{rad}(\mathbf{x})]$ but only maintain the inclusion $\mathbf{x} \subseteq [\mathrm{mid}(\mathbf{x}) - \mathrm{rad}(\mathbf{x}), \mathrm{mid}(\mathbf{x}) + \mathrm{rad}(\mathbf{x})]$. The power of interval arithmetic lies in its implementation on computers. In particular, *outwardly rounded* interval arithmetic allows computing *rigorous enclosures* for the ranges of operations/functions. For example, the operation (2.24) can be made rigorous by adding suitable *rounding controls*:

$$\mathbf{x} + \mathbf{y} \equiv [\lfloor \underline{x} + \underline{y} \rfloor, \lceil \overline{x} + \overline{y} \rceil]; \tag{2.27a}$$

$$\mathbf{x} - \mathbf{y} \equiv [\lfloor \underline{x} - \overline{y} \rfloor, \lceil \overline{x} - \underline{y} \rceil]; \tag{2.27b}$$

$$\mathbf{x} * \mathbf{y} \equiv [\min\{\lfloor \underline{x}\underline{y} \rfloor, \lfloor \underline{x}\overline{y} \rfloor, \lfloor \overline{x}\underline{y} \rfloor, \lfloor \overline{x}\overline{y} \rfloor\}, \max\{\lceil \underline{x}\underline{y} \rceil, \lceil \underline{x}\overline{y} \rceil, \lceil \overline{x}\underline{y} \rceil, \lceil \overline{x}\overline{y} \rceil\}]; \tag{2.27c}$$

$$\mathbf{x} \div \mathbf{y} \equiv \mathbf{x} * 1/\mathbf{y} \text{ if } 0 \notin \mathbf{y}, \text{ where } 1/\mathbf{y} \equiv [\lfloor 1/\overline{y} \rfloor, \lceil 1/\underline{y} \rceil]. \tag{2.27d}$$

This makes a qualitative difference in scientific computations because the results are now intervals in which the exact result must lie. Interval arithmetic can be carried out for virtually any arithmetic expression that can be evaluated with floating-point arithmetic. However, expressions that are equivalent in real arithmetic differ in interval arithmetic because interval arithmetic is only *subdistributive*. Therefore, computations should be arranged so that the overestimation of the ranges of functions is minimized.

2.2.2.4. Interval Functions

An *interval function* is a function from $\mathbb{I}^n$ to $\mathbb{I}^m$ (or from $\mathbb{I}_\circ^n$ to $\mathbb{I}_\circ^m$, if we want to extend to include open intervals). There are guidelines to follow when defining interval functions for practical use. It is desirable to follow the spirit of interval arithmetic:

- An interval represents any real number in its range;
- The result of an interval function represents all possible real results over an interval.

The *inclusion property* formally states this spirit for two cases: functions and relations.

Definition 2.80 (Interval Form of Functions). Let $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ be a function. Any interval function $[f] : \mathbb{I}^n \to \mathbb{I}^m$ satisfying the *inclusion property*

$$\forall x \in D, \forall \mathbf{x} \in \mathbb{I}^n : x \in \mathbf{x} \Rightarrow f(x) \in [f](\mathbf{x}) \tag{2.28}$$

is called an *interval form* of f (or for f).

Definition 2.81 (Interval Form of Relations). Let $R \subseteq \mathbb{R}^n$ be a relation. An interval relation $[R] \subseteq \mathbb{I}^n$ is called an *interval form* of R (or for R) if it satisfies the *inclusion property*:

$$\forall r \in R, \forall \mathbf{x} \in \mathbb{I}^n : r \in \mathbf{x} \Rightarrow \mathbf{x} \in [R]. \tag{2.29}$$

An interval form $[f]$ of f on a box $\mathbf{x}$ of maximum width ε is said to have the *approximation property of order k* if $[f](\mathbf{x}) \subseteq (1 + \text{const.}\varepsilon^k)\square f(\mathbf{x})$.

An interval form of a function is also referred to as an *inclusion function* in [Jaulin *et al.* 2001, p. 27]. It is also called an *interval extension* in literature. In Section 5.3, we study the concept of an interval form of a *multifunction*. In Section 5.5.2, we also study about an abstraction of the interval form concept. Hereafter we give a relationship between the concept of an interval form of a function and that of an interval form of a relation.

Theorem 2.82. Let $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ be a function and $\mathbf{b} \in \mathbb{I}^m$. If $[f]$ is an interval form of f. Then interval relation $[R] \equiv \{\mathbf{x} \in \mathbb{I}^n \mid [f](\mathbf{x}) \cap \mathbf{b} \neq \emptyset\}$ is an interval form of the real relation $R \equiv \{x \in D \mid f(x) \in \mathbf{b}\}$.

Proof. For all $r \in R$ and $\mathbf{x} \in \mathbb{I}^n$, if $r \in \mathbf{x}$ then $f(r) \in \mathbf{b}$, thus $[f](\mathbf{x}) \cap \mathbf{b} \ni f(r)$. That is, $\mathbf{x} \in [R]$. The proof is hence completed. ∎

2.2.2.5. From Closed Intervals to Open Intervals

Notations for Algorithms. Precisely, a general real interval is not only represented by the values of its bounds, but also the openness/closedness of the two bounds. In Section 2.2.2.1, the openness and closedness of bounds are represented by bracket symbols: ']' $\equiv 3$ and '[' $\equiv 1$ are for lower bounds, and '[' $\equiv 2$ and ']' $\equiv 0$ are for upper bounds.

In some cases (see Chapter 8 for an example), accessing the meaning of the above bracket symbols in algorithms is needed. Hence, we define here another syntax for general real intervals. The lower and upper brackets of intervals are taken in $\mathcal{L} = \{1, 3\}$ and $\mathcal{U} = \{0, 2\}$, respectively.

The set of (symbolic) bounds is $\mathbb{R}_\diamond = \mathbb{R}_\triangleleft \cup \mathbb{R}_\triangleright$, where $\mathbb{R}_\triangleleft = \mathbb{R} \times \mathcal{L} \cup \{(-\infty, 3)\}$ and $\mathbb{R}_\triangleright = \mathbb{R} \times \mathcal{U} \cup \{(+\infty, 2)\}$. The set $\mathbb{R}_\diamond$ of (symbolic) bounds is totally ordered by the ordering relation $\leq$ defined as follows: for every two bounds $\beta_1 \equiv (x_1, b_1)$ and $\beta_2 \equiv (x_2, b_2)$ in $\mathbb{R}_\diamond$, we have

$$\beta_1 = \beta_2 \quad \Leftrightarrow \quad x_1 = x_2 \wedge b_1 = b_2, \tag{2.30a}$$

$$\beta_1 \leq \beta_2 \quad \Leftrightarrow \quad x_1 < x_2 \vee (x_1 = x_2 \wedge b_1 \leq b_2). \tag{2.30b}$$

If $\beta_1 \neq \beta_2$ holds in addition to (2.30), we write $\beta_1 < \beta_2$. The reverse relations, such as $\beta_2 > \beta_1$ and $\beta_1 \geq \beta_2$, can also defined as usual. The bounds in $\mathbb{R}_\diamond$ are used to construct the set of (general) intervals:

$$\mathbb{I}_\circ \equiv \{\langle \beta_1, \beta_2 \rangle \mid \beta_1 < \beta_2,\ \beta_1 \in \mathbb{R}_\triangleleft,\ \beta_2 \in \mathbb{R}_\triangleright\} \cup \{\emptyset\}.$$

The relation between the usual notations of intervals and the new notations is:

$$\begin{array}{rcccl} [x, y] & \equiv & \langle (x,1), (y,0) \rangle & \equiv & \{r \in \mathbb{R} \mid x \leq r \leq y\}; \\ ]x, y] & \equiv & \langle (x,3), (y,0) \rangle & \equiv & \{r \in \mathbb{R} \mid x < r \leq y\}; \\ [x, y[& \equiv & \langle (x,1), (y,2) \rangle & \equiv & \{r \in \mathbb{R} \mid x \leq r < y\}; \\ ]x, y[& \equiv & \langle (x,3), (y,2) \rangle & \equiv & \{r \in \mathbb{R} \mid x < r < y\}. \end{array}$$

For convenience, we define, for each bound $\beta = (x, b) \in \mathbb{R}_\diamond$, the negation of β as $\neg\beta = (x, 3-b)$. Thus, $\neg(\neg\beta) = \beta$ holds and if $\beta \in \mathbb{R}_\triangleright$ then $\neg\beta \in \mathbb{R}_\triangleleft$, and vice versa, if $\beta \in \mathbb{R}_\triangleleft$ then $\neg\beta \in \mathbb{R}_\triangleright$.

Computations with Open/Closed Intervals. Most interval arithmetic libraries have been implemented only for closed intervals. One might use these libraries to perform computations on open/closed intervals. Indeed, every general interval (vector) $\beta \equiv \langle (\underline{x}, \underline{b}), (\overline{x}, \overline{b}) \rangle \in \mathbb{I}_\circ$ is contained in the respective closed interval (vector) $\mathbf{x} \equiv [\underline{x}, \overline{x}] \in \mathbb{I}$. Therefore, the computations can be performed on $\mathbf{x}$, in place of β, to get an enclosure of the result. If the computations are domain reduction techniques, we can perform them on respective closed intervals, and then take the set intersection of the computed intervals with the initial general intervals. For example, after performing a domain reduction technique on the closed interval $[\underline{x}, \overline{x}]$, we get a closed interval $[\underline{y}, \overline{y}] \subseteq [\underline{x}, \overline{x}]$. The result to be obtained is thus the set intersection $\beta \cap [\underline{y}, \overline{y}]$.

The above procedure is useful in the context that we have only interval arithmetic libraries for closed intervals, but the solving process needs to perform computations on open/closed intervals, for example, when performing a bisection that requires two resulting boxes to be *disjoint* (hence, they are in $\mathbb{I}_\circ$). Two boxes resulting from a bisection of a box are required to be disjoint in some cases, for example, as in Chapter 8. If the two resulting boxes are not disjoint, this may contribute to the *cluster effect* [Kearfott and Du 1992, 1994; Van Hentenryck *et al.* 1997b], because both boxes may contain the same solutions on their common facets.

2.2.3. Affine Arithmetic

Affine arithmetic [Comba and Stolfi 1993] is an extension of interval arithmetic that keeps track of correlations between input and computed quantities. Therefore, it is resistant to the catastrophic loss of accuracy often observed in long-running interval computations.

Affine arithmetic is somewhat similar to Hansen's generalized interval arithmetic [Hansen 1975], but differs in several important details. For example, in Hansen's model the internal approximation errors are combined with the input uncertainties, whereas in affine arithmetic they are represented separately, which makes it possible for the approximation error introduced

at one step to be canceled out at a later step. Furthermore, in Hansen's model, but not in affine arithmetic, the joint range of two variables may be a nonconvex region.

The ranges of functions obtained with affine arithmetic may be substantially more accurate than those obtained with interval arithmetic. However, the operations of affine arithmetic are often more expensive than those of interval arithmetic. Some comparisons on interval methods and affine arithmetic methods can be found in [Stolfi and de Figueiredo 1997], [Messine 2002], and [Martin *et al.* 2002].

2.2.3.1. Affine Form

In particular, a real-valued quantity x is represented by an *affine form* defined as follows

$$\hat{x} \equiv x_0 + x_1\epsilon_1 + \cdots + x_n\epsilon_n, \tag{2.31}$$

where $x_0, \ldots, x_n$ are real *coefficients* and $\epsilon_1, \ldots, \epsilon_n$ are *noise variables* (originally called *noise symbols*) taking values in $[-1, 1]$. Similarly to interval arithmetic, affine arithmetic also allows using rounded floating-point arithmetic to construct *rigorous enclosures* for the ranges of operations and functions [Stolfi and de Figueiredo 1997]. In long-running computations, the number of noise variables may be very high, but their distribution is often sparse. Therefore, we only need to store the *nonzero coefficients* and the indices of the respective noise variables of the considered affine form (2.31).

2.2.3.2. Affine Operations

In affine arithmetic, a general *affine operation* of the form $\alpha\hat{x} + \beta\hat{y} + \gamma$ ($\alpha, \beta, \gamma \in \mathbb{R}$) can be obtained exactly, except the rounding errors, by the following formula:

$$\alpha\hat{x} + \beta\hat{y} + \gamma \equiv (\alpha x_0 + \beta y_0 + \gamma) + \sum_{i=1}^{n} (\alpha x_i + \beta y_i)\epsilon_i. \tag{2.32}$$

In the computations on floating-point arithmetic, if the rounding error is enclosed by $[-c, c]$, a new term $z_{\text{new}}\epsilon_{\text{new}}$ is added to represent this error, where $z_{\text{new}} = c$. The rigorous result is

$$\alpha\hat{x} + \beta\hat{y} + \gamma \equiv (\alpha x_0 + \beta y_0 + \gamma) + \sum_{i=1}^{n} (\alpha x_i + \beta y_i)\epsilon_i + z_{\text{new}}\epsilon_{\text{new}}. \tag{2.33}$$

Note that the *length* of the result (2.33) is increased by one.

2.2.3.3. Non-Affine Operations

Unlike the affine operations, non-affine operations such as $f(\hat{x}, \hat{y})$ can only be computed by approximations. In general, the exact result of a non-affine operation has form $f^*(\epsilon_1, \ldots, \epsilon_n)$, where f^* is a *nonlinear function* corresponding to f. In general, this result is then approximated by an *affine function* $f^{\text{a}}(\epsilon_1, \ldots, \epsilon_n) = z_0 + z_1\epsilon_1 + \cdots + z_n\epsilon_n$. A new term $z_{\text{new}}\epsilon_{\text{new}}$ is used to represent the difference between f^* and f^{a}, namely

$$z_{\text{new}}\epsilon_{\text{new}} = f^*(\epsilon_1, \ldots, \epsilon_n) - f^{\text{a}}(\epsilon_1, \ldots, \epsilon_n). \tag{2.34}$$

Hence, the result is an affine form

$$z \equiv z_0 + z_1\epsilon_1 + \cdots + z_n\epsilon_n + z_{\text{new}}\epsilon_{\text{new}}, \tag{2.35}$$

where $\epsilon_{\rm new} \in [-1, 1]$ and $z_{\rm new}$ must not be less than the *maximum absolute error*; that is,

$$z_{\rm new} \geq \sup\{|f^*(\epsilon_1, \dots, \epsilon_n) - f^{\rm a}(\epsilon_1, \dots, \epsilon_n)| : \forall(\epsilon_1, \dots, \epsilon_n) \in [-1, 1]^n\}.$$

An important goal is to find $f^{\rm a}$ such that the maximum absolute error is as small as possible or can be bounded by a value $z_{\rm new}$ that is as small as possible. This is a subject of *Chebyshev approximation theory*, which is a well-developed field with a vast literature. In fact, a sub-theory of affine approximations is enough for affine arithmetic because we only need to construct the elementary operations in affine arithmetic. Factorable expressions/functions can be recursively composed of these elementary operations.

The reader can find in [Stolfi and de Figueiredo 1997] some detailed rigorous procedures for constructing elementary operations, such as $1/\hat{x}$, $\hat{x}/\hat{y}$, $\hat{x}^2$, $\sqrt{\hat{x}}$, $e^{\hat{x}}$ and $\ln \hat{x}$, in affine arithmetic. Hereafter, we recall briefly two most special operations: the multiplication and the division.

Multiplication. In affine arithmetic, the multiplication of two affine forms $\hat{x} = x_0 + \sum_{i=1}^n x_i\epsilon_i$ and $\hat{y} = y_0 + \sum_{i=1}^n y_i\epsilon_i$ is another affine form $\hat{z} = z_0 + \sum_{i=1}^n z_i\epsilon_i + z_{\rm new}\epsilon_{\rm new}$ defined as

$$z_0 \equiv x_0 y_0, \tag{2.36a}$$

$$z_i \equiv x_0 y_i + y_0 x_i \quad (\text{for } i = 1, \dots, n), \tag{2.36b}$$

$$z_{\rm new} \equiv \left(\sum_{i=1}^n |x_i|\right)\left(\sum_{i=1}^n |y_i|\right). \tag{2.36c}$$

The number of real additions in (2.36) is $3n - 2$. The number of real multiplications (2.36) is $2n + 2$. Hence, the total number of real operations for the multiplication defined by (2.36) is $5n$. The multiplication defined by (2.36) is however not tight. Therefore, we can use the following tighter one at higher cost (it can be viewed as a special case of (2.43)):

$$z_0 \equiv x_0 y_0 + \frac{1}{2}\sum_{i=1}^n x_i y_i, \tag{2.37a}$$

$$z_i \equiv x_0 y_i + y_0 x_i \quad (\text{for } i = 1, \dots, n), \tag{2.37b}$$

$$z_{\rm new} \equiv \frac{1}{2}\sum_{i=1}^n |x_i y_i| + \sum_{1 \leq i,j \leq n;\ i \neq j} |x_i y_j|. \tag{2.37c}$$

The number of *real additions* in (2.37) is $n^2 + 2n - 1$. The number of *real multiplications* in (2.37) is $n^2 + 2n + 3$. Hence, the total number of real operations for the multiplication defined by (2.37) is $2(n + 1)^2$. This cost is much higher than the cost, $5n$, of (2.36). Proving the inclusion property of the multiplications (2.36) and (2.37) is easy, and hence is omitted.

Miyajima [2000, p. 22] proposed to replace the multiplication (2.37) with the following:

$$z_0 \equiv x_0 y_0 + \frac{1}{2}\sum_{i=1}^n x_i y_i, \tag{2.38a}$$

$$z_i \equiv x_0 y_i + y_0 x_i \quad (\text{for } i = 1, \dots, n), \tag{2.38b}$$

$$z_{\rm new} \equiv \frac{1}{2}\sum_{i=1}^n |x_i y_i| + \sum_{1 \leq i < j \leq n} |x_i y_j + x_j y_i|. \tag{2.38c}$$

The multiplication (2.38) provides a tighter enclosure than the multiplication (2.37) does. They both have a number of similar real operations. However, the cost of each term $|x_i y_j + x_j y_i|$ is more expensive than the cost of $|x_i y_j| + |x_j y_i|$ when they need to be rounded upwards.

Division. The division $\hat{x}/\hat{y}$ can be written as $\hat{x} * (1/\hat{y})$; hence it can be computed by one reciprocal and one multiplication. Kolev [2002] proposed an improvement for computing the reciprocal $1/\hat{y}$, hence for computing $\hat{x}/\hat{y} := \hat{x} * (1/\hat{y})$. This has an interesting property that $\hat{x}/\hat{x} = 1$, which does not hold for interval arithmetic. Miyajima *et al.* [2003] also proposed new methods to compute $\hat{x}/\hat{y}$. However, these methods are too complicated to be presented here. Hence, the reader should find the details in [Kolev 2002; Miyajima *et al.* 2003].

2.2.3.4. Variants of Affine Arithmetic

Kolev [1998] showed that, under some assumptions, it is possible to enclose a (piecewise) continuously differentiable separable function $f : \mathbf{x} \in \mathbb{I}^n \to \mathbb{R}^m$ in a linear enclosure (see Section 3.3.3.1). Namely, let

$$f(x) = \sum_{j=1}^{n} f_j(x_j),$$

where $x \equiv (x_1, \ldots, x_n)^{\mathrm{T}}$ is a vector of n real variables. It is possible to compute a linear enclosure of f of the form

$$f(x) \in \sum_{j=1}^{n} a_j x_j + \mathbf{d}_j, \tag{2.39}$$

where $\mathbf{d}_j \in \mathbb{I}^m$ and $a_j \in \mathbb{R}^m$. Probably inspired by the similarity between (2.39) and Hansen's *generalized interval* [Hansen 1975], Kolev [2001] proposed a modified form of Hansen interval. However, Kolev's arithmetic, which is defined on Kolev's forms, is much similar to affine arithmetic (see Section 2.2.3), but not similar to *Hansen interval arithmetic*. We recall here the formal definition of Kolev's form.

Definition 2.83 (Kolev Affine Form, Kolev Interval Form). A *Kolev affine form* is a *semi-affine function* on *n noise variables* $\kappa_1, \ldots, \kappa_n$ of the form

$$\tilde{x} = c_x + \sum_{i=1}^{n} x_i \kappa_i + \mathbf{v}_x, \quad \kappa_i \in \mathbf{v}_i, \tag{2.40}$$

where $\mathbf{v}_i \equiv [-v_i, v_i]$ (for $i = 1, \ldots, n$) and $\mathbf{v}_x \equiv [-v_x, v_x]$ are symmetric intervals; $x_1, \ldots, x_n$ are real coefficients; and $c_x \in \mathbb{R}$. It can also be written in the interval form:

$$\tilde{\mathbf{x}} = c_x + \sum_{i=1}^{n} x_i \mathbf{v}_i + \mathbf{v}_x, \tag{2.41}$$

which is called a *Kolev generalized interval* associated with the above Kolev affine form.

A Kolev affine form (and also its associated generalized interval) was originally called a *generalized interval* or a *G interval* by Kolev [2001]. The arithmetic that is defined on Kolev generalized intervals follows the spirit of affine arithmetic rather than the spirit of interval arithmetic; thus, we call it *Kolev generalized affine arithmetic*. Indeed, one can see hereafter that it can be viewed as a generalization of (standard) affine arithmetic.

Kolev Generalized Affine Arithmetic. An *affine operation* $\alpha x+\beta y+\gamma$, where $\alpha, \beta, \gamma \in \mathbb{R}$, of two Kolev affine forms, $\tilde{x} \equiv c_x + \sum_{i=1}^n x_i\kappa_i + \mathbf{v}_x$ and $\tilde{y} \equiv c_y + \sum_{i=1}^n y_i\kappa_i + \mathbf{v}_y$, is another Kolev affine form $\tilde{z} \equiv c_z + \sum_{i=1}^n z_i\kappa_i + \mathbf{v}_z$, where

$$\begin{aligned} c_z &\equiv \alpha c_x + \beta c_y + \gamma, && (2.42a)\\ z_i &\equiv \alpha x_i + \beta y_i \quad (\text{for } i = 1, \ldots, n), && (2.42b)\\ v_z &\equiv |\alpha| v_x + |\beta| v_y. && (2.42c) \end{aligned}$$

The product $\tilde{z}$ of two Kolev affine forms, $\tilde{x} \equiv c_x + \sum_{i=1}^n x_i\kappa_i + \mathbf{v}_x$ and $\tilde{y} \equiv c_y + \sum_{i=1}^n y_i\kappa_i + \mathbf{v}_y$, is another Kolev affine form $\tilde{z} \equiv c_z + \sum_{i=1}^n z_i\kappa_i + \mathbf{v}_z$ defined as follows:

$$\begin{aligned} c_z &\equiv c_x c_y + \frac{1}{2}\sum_{i=1}^n x_i y_i v_i^2, && (2.43a)\\ z_i &\equiv c_x y_i + c_y x_i \quad (\text{for } i = 1, \ldots, n), && (2.43b)\\ v_z &\equiv v_x v_y + |c_x| v_y + |c_y| v_x + \sum_{1\le i,j\le n;\ i\ne j} |x_i y_j| v_i v_j + \\ &\quad v_x \sum_{i=1}^n |y_i| v_i + v_y \sum_{i=1}^n |x_i| v_i + \frac{1}{2}\sum_{i=1}^n |x_i y_i| v_i^2. && (2.43c) \end{aligned}$$

The number of *real additions* in (2.43) is $n^2 + 4n + 2$. The number of *real multiplications* in these formulas is $3n^2 + 4n + 8$. In present computers, the cost of a floating-point addition is quite the same as that of a floating-point multiplication, then the complexity of (2.43) is $4n^2 + 8n + 10$ real operations.

Kolev [2001] also extended the above arithmetic for continuously differentiable elementary operations $\psi : D \subseteq \mathbb{R} \to \mathbb{R}$ (on which (3.105) is defined) by using the linear relaxation technique in Section 3.3.3.1. In particular, given an interval $\mathbf{x} \subseteq D$, for all $x \in \mathbf{x}$ we have

$$\psi(x) \in ax + \mathbf{d},$$

where $a \in \mathbb{R}$, $\mathbf{d} \in \mathbb{I}$. Now let consider a Kolev affine form $\tilde{x} \equiv c_x + \sum_{i=1}^n x_i\kappa_i + \mathbf{v}_x$ such that $\tilde{x} \subseteq \mathbf{x}$, we then have

$$\psi(\tilde{x}) \subseteq a\left(c_x + \sum_{i=1}^n x_i\kappa_i + \mathbf{v}_x\right) + \mathbf{d}.$$

Hence, a Kolev affine form $\tilde{z} \equiv c_z + \sum_{i=1}^n z_i\kappa_i + \mathbf{v}_z$ can be obtained for $\psi(\tilde{x})$, w.r.t. the inclusion property, by defining that

$$\begin{aligned} \mathbf{d}' &\equiv \mathbf{d} + a\left(\sum_{i=1}^n \mathbf{v}_x\right), && (2.44a)\\ c_z &\equiv a c_x + \mathrm{mid}(\mathbf{d}'), && (2.44b)\\ z_i &\equiv a x_i \quad (\text{for } i = 1, \ldots, n), && (2.44c)\\ v_z &\equiv \mathrm{rad}(\mathbf{d}'). && (2.44d) \end{aligned}$$

We can obtain a Kolev affine form (and its associated Kolev generalized interval) for any *factorable expression/function*, with respect to the inclusion property, by composing the expression/function using the above-defined elementary operations. Since most elementary operations are continuously differentiable and (3.105) is defined, a Kolev affine form can be obtained for any factorable expression built on these elementary operations.

Messine Affine Arithmetic. Historically, Messine [1999] proposed a simpler version of Kolev affine form/arithmetic before Kolev [2001] did propose the above generalized version. Affine forms of Messine [1999, 2000, 2002] nearly resemble to Kolev affine forms in Definition 2.83 when fixing $v_i = 1$ for all i, and the idea of Messine affine arithmetic is similar to that of Kolev affine arithmetic in (2.42) and (2.43). In particular, a *Messine affine form* has the form

$$\breve{x} \equiv x_0 + \sum_{i=1}^{n} x_i \epsilon_i + x_{n+1}[-1,1] + x_{n+2}[0,1] + x_{n+3}[-1,0], \tag{2.45}$$

where $\epsilon_i, \dots, \epsilon_n$ are the noise variables taking values in the interval $[-1,1]$, as in affine forms, and the coefficients x_{n+1}, x_{n+2} and x_{n+3} are nonnegative. Suppose the Messine affine form of $\breve{y}$ written similarly to that of $\breve{x}$. For all real numbers $\alpha = -\beta \geq 0$ and γ, *Messine affine arithmetic* defines that

$$\begin{aligned} \breve{x} + \breve{y} \;&\equiv\; (x_0 + y_0) + \sum_{i=1}^{n} (x_i + y_i)\epsilon_i + (x_{n+1} + y_{n+1})[-1,1] + \\ &\quad (x_{n+2} + y_{n+2})[0,1] + (x_{n+3} + y_{n+3})[-1,0]; \end{aligned} \tag{2.46}$$

$$\begin{aligned} \breve{x} - \breve{y} \;&\equiv\; (x_0 - y_0) + \sum_{i=1}^{n} (x_i - y_i)\epsilon_i + (x_{n+1} + y_{n+1})[-1,1] + \\ &\quad (x_{n+2} + y_{n+3})[0,1] + (x_{n+3} + y_{n+2})[-1,0]; \end{aligned} \tag{2.47}$$

$$\alpha * \breve{x} \;\equiv\; \alpha x_0 + \sum_{i=1}^{n} \alpha x_i \epsilon_i + \alpha x_{n+1}[-1,1] + \alpha x_{n+2}[0,1] + \alpha x_{n+3}[-1,0]; \tag{2.48}$$

$$\beta * \breve{x} \;\equiv\; \beta x_0 + \sum_{i=1}^{n} \beta x_i \epsilon_i + \alpha x_{n+1}[-1,1] + \alpha x_{n+3}[0,1] + \alpha x_{n+2}[-1,0]; \tag{2.49}$$

$$\gamma + \breve{x} \;\equiv\; (\gamma + x_0) + \sum_{i=1}^{n} x_i \epsilon_i + x_{n+1}[-1,1] + x_{n+2}[0,1] + x_{n+3}[-1,0]. \tag{2.50}$$

The multiplication in Messine affine arithmetic is defined as follows:

$$\breve{x} * \breve{y} \;\equiv\; x_0 y_0 + \sum_{i=1}^{n} (x_0 y_i + x_i y_0)\epsilon_i + K_1[-1,1] + K_2[0,1] + K_3[-1,0], \tag{2.51}$$

where K_1, K_2, and K_3 are computed by:[3]

$$K_1 \;\equiv\; |x_0| y_{n+1} + |y_0| x_{n+1} + x_{n+1} y_{n+1} + \sum_{1 \leq i,j \leq n+3;\; i \neq j} |x_i y_j|, \tag{2.52a}$$

$$K_2 \;\equiv\; K_2^{\langle 0 \rangle} + x_{n+2} y_{n+2} + x_{n+3} y_{n+3} + \sum_{i=1;\; x_i y_i > 0}^{n} x_i y_i, \tag{2.52b}$$

$$K_3 \;\equiv\; K_3^{\langle 0 \rangle} + \sum_{i=1;\; x_i y_i < 0}^{n} |x_i y_i|, \tag{2.52c}$$

[3] There is a minor error in the multiplication of Messine affine arithmetic in [Messine 1999, 2000, 2002]. To be correct, a term $x_{n+1} y_{n+1}$ must be added to K_1 and removed from K_2 therein. We correct this error in the version presented here.

where $K_2^{\langle 0\rangle}$ and $K_3^{\langle 0\rangle}$ are, in turn, defined as follows:

$$K_2^{\langle 0\rangle} = \begin{cases} x_0 y_{n+2} + y_0 x_{n+2} & \text{if } x_0 \geq 0 \text{ and } y_0 \geq 0, \\ x_0 y_{n+2} - y_0 x_{n+3} & \text{if } x_0 \geq 0 \text{ and } y_0 < 0, \\ -x_0 y_{n+3} + y_0 x_{n+2} & \text{if } x_0 < 0 \text{ and } y_0 \geq 0, \\ -x_0 y_{n+3} - y_0 x_{n+3} & \text{if } x_0 < 0 \text{ and } y_0 < 0; \end{cases} \tag{2.53a}$$

$$K_3^{\langle 0\rangle} = \begin{cases} x_0 y_{n+3} + y_0 x_{n+3} & \text{if } x_0 \geq 0 \text{ and } y_0 \geq 0, \\ x_0 y_{n+3} - y_0 x_{n+2} & \text{if } x_0 \geq 0 \text{ and } y_0 < 0, \\ -x_0 y_{n+2} + y_0 x_{n+3} & \text{if } x_0 < 0 \text{ and } y_0 \geq 0, \\ -x_0 y_{n+2} - y_0 x_{n+2} & \text{if } x_0 < 0 \text{ and } y_0 < 0. \end{cases} \tag{2.53b}$$

The numbers of real additions and multiplications are $n^2+7n+13$ and $n^2+8n+16$, respectively. Hence, the total is $2n^2+15n+29$. One can easily see that Messine's multiplication is a special case of Kolev's definition in (2.43). Notice that this multiplication is not as tight as the one in (2.38) when adapted to Messine affine forms or vice versa.

Kolev Affine Arithmetic. Fortunately, Kolev [2002] improved the multiplication in affine arithmetic by revising the formulas in (2.43) and (2.51) for computing an affine form of the product.

$$\hat{z} \equiv c_z + \sum_{i=1}^{n} z_i \kappa_i + z_{\text{new}} \kappa_{\text{new}}, \tag{2.54}$$

where κ_{new} is a new noise variable taking its value in $[-1, 1]$, and all the other noise variables are also in $[-1, 1]$. The new formulas are:

$$S_x \equiv \sum_{i=1}^{n} |x_i|, \quad S_y \equiv \sum_{i=1}^{n} |y_i|, \quad P \equiv \frac{1}{2}\sum_{i=1}^{n} x_i y_i, \tag{2.55a}$$

$$c_z \equiv c_x c_y + P, \tag{2.55b}$$

$$z_i \equiv c_x y_i + c_y x_i \quad (\text{for } i = 1, \ldots, n), \tag{2.55c}$$

$$z_{\text{new}} \equiv v_x v_y + v_y(|c_x| + S_x) + v_x(|c_y| + S_y) + S_x S_y - |P|. \tag{2.55d}$$

In computations with *rigorous rounding controls*, I propose to replace P in (2.55a) with $\langle P\rangle \pm e_P$, where $\langle Z\rangle \pm z$ denotes some floating-point number such that $\langle Z\rangle - z \leq Z \leq \langle Z\rangle + z$ and $z \in \mathbb{F}$, and then replace (2.55d) with $z_{\text{new}} \equiv \lceil v_x v_y + v_y(|c_x| + S_x) + v_x(|c_y| + S_y) + S_x S_y + 2e_P - \langle P\rangle \rceil$. By this way, we avoid computing P and $|P|$ as if they were completely different expressions in order to reduce the computation cost. The other parts of (2.55) are rounded in the same way. In (2.55), the number of real additions is $4n+5$ and the number of real multiplications is $3n+7$. The total number of real operations is $7n+12$. Therefore, the new multiplication (2.55) is much faster than the previous multiplication in (2.37) and (2.43), when n is big.

Note 2.84. By substituting the term $z_{\text{new}}\kappa_{\text{new}}$ in the (standard) affine form $\hat{z}$ in (2.54) with $\mathbf{v}_z \equiv [-z_{\text{new}}, z_{\text{new}}]$, we get a Kolev affine form $\tilde{z}' \equiv c_z + \sum_{i=1}^{n} z_i\kappa_i + \mathbf{v}_z$, which is equivalent to $\hat{z}$. However, it is not tighter than the Kolev affine form $\tilde{z}$ obtained by (2.43), namely,

$$\tilde{z}' \equiv \{\hat{z} \mid \kappa_{\text{new}} \in [-1, 1]\} \supseteq \tilde{z}. \tag{2.56}$$

Remark 2.85. All affine forms presented here easily converted to (standard) affine form by replacing each interval of them with a new noise variable. All the above improvements to variants of affine arithmetic can also be easily adapted to (standard) affine arithmetic.

2.3. Fundamental Consistency Notions

In this section and its subsections, we recall some fundamental consistency notions for a quick reference. The reader can find algorithms for achieving local consistency states of CSPs in the books on foundations of constraint satisfaction [Tsang 1993], on principles of constraint programming [Apt 2003], and the references mentioned in this section.

To compare the consistency notions, one often uses the *strongness* that is defined as follows.

Definition 2.86 (Strongness). We define a partial order on consistency notions:

- X-consistency is said to be *stronger* than Y-consistency if X-consistency implies Y-consistency. In this case, with respect to the strongness, we write:

$$X\text{-consistency} \geq Y\text{-consistency and } Y\text{-consistency} \leq X\text{-consistency}.$$

- X-consistency is said to be *strictly stronger* than Y-consistency if X-consistency implies Y-consistency but Y-consistency does not implies X-consistency. In this case, with respect to the strongness, we write:

$$X\text{-consistency} > Y\text{-consistency and } Y\text{-consistency} < X\text{-consistency}.$$

The following is a property that is often enjoyed by consistency notions.

Definition 2.87 (Monotonicity). A $\mathcal{ANY}$-consistency is said to be *monotonic* if, for every CSP $\mathcal{P}$ on which $\mathcal{ANY}$-consistency is defined, every singleton CSP of $\mathcal{P}$ is $\mathcal{ANY}$-consistent provided that $\mathcal{P}$ is $\mathcal{ANY}$-consistent.

Notation 2.88. Consider a CSP $\mathcal{P} \equiv (\mathcal{V}, \mathcal{D}, \mathcal{C})$. Let X be a subset of $\mathcal{V}$. Throughout this section, we use the notation $\mathbf{C}_X$ (called the *conjunction constraint*) to denote ***either***

- the intersection of all the constraints on all the variables X (sorted in the order in the sequence of variables of $\mathcal{P}$), if they exist; ***or***
- the Cartesian product of the domains of all the variables in X (sorted in the order in the sequence of variables of $\mathcal{P}$), otherwise.

2.3.1. Global Consistency

It is well-known that the global consistency is a sufficient condition for backtrack-free search; that is, any instantiation of a subsequence of variables can be extended to a solution without any backtracking in search. Here is the definition.

Definition 2.89 (Global Consistency). A CSP is said to be *globally consistent* if every instantiation of subsequences of variables can be extended to a solution without backtracking. Equivalently, a CSP with n variables is said to be *globally consistent* if it is strongly n-consistent (see Definition 2.108),

Global consistency is very difficult to be obtained, in general, but still weaker than the requirement of globally solved form (Definition 2.36). Obtaining global consistency is usually restricted to some special types of constraints and/or special structures of problems. The reader can find a survey on conditions for guaranteeing global consistency in [van Beek 1992], with updates in [van Beek and Dechter 1995] and [Sam-Haroud 1995, Chapter 5].

2.3.2. Classical Local Consistency Notions

In this section and its subsections, we recall the most common local consistency notions with illustration examples, as described in [Apt 2003, Chapter 5].

2.3.2.1. Node Consistency

A form of node consistency was first studied by Montanari [1974]. The version presented here is due to Mackworth [1977].

Definition 2.90 (Node Consistency). A CSP is said to be *node consistent* if, for every variable x, every unary constraint on x coincides with the domain of x.

Note 2.91. Note that node consistency imposes no condition on non-unary constraints and that node consistency neither implies nor is implied by the consistency (Definition 2.14). Note also that a CSP with all empty domains is node consistent.

Example 2.92. Consider the CSP $\langle x = 0,\ y = 0;\ x \in \mathbb{N},\ y \in \mathbb{N}\rangle$. This CSP is clearly consistent, but not node consistent. ♣

2.3.2.2. Arc Consistency

A form of arc consistency was first studied independently by Waltz [1972, 1975] and Montanari [1974]. The version presented here is due to Mackworth [1977].

Definition 2.93 (Arc Consistency). A binary constraint C on the variables (x, y); which are associated with the domains (D_x, D_y), respectively; is called *arc consistent* if

$$(i) \qquad \forall a \in D_x, \exists b \in D_y : (a, b) \in C,$$
$$(ii) \qquad \forall b \in D_y, \exists a \in D_y : (a, b) \in C.$$

A CSP is said to be *arc consistent* if all its binary constraint are arc consistent.

Note 2.94. Note that a CSP with no binary constraints is arc consistent and that arc consistency does not imply the consistency of the CSP. Note also that a CSP with all empty domains is arc consistent.

Example 2.95. The CSP $\langle x = y,\ x \neq y;\ x \in \{0,1\},\ y \in \{0,1\}\rangle$ is obviously arc consistent and also inconsistent. ♣

Arc consistency and its variants constitute a class of important consistency notations. Faltings [1994] showed that arc consistency can be efficiently achieved for binary NCSPs by exploring only extreme points of constraints. A variant of arc consistency is directional arc consistency which is devised for directed network of constraints. The idea of maintaining directional arc consistency has long been proposed and studied in search (see [Haralick and Elliot 1980]), the terminology was never formally defined until Dechter and Pearl [1988] did.

Definition 2.96 (Directional Arc Consistency). Consider a total order $\prec$ on the considered variables. A binary constraint C on the variables (x, y); which are associated with the domains (D_x, D_y), respectively; is called *directionally arc consistent w.r.t* $\prec$ if

$$(i) \qquad x \prec y \Rightarrow (\forall a \in D_x, \exists b \in D_y : (a,b) \in C)\,,$$
$$(ii) \qquad y \prec x \Rightarrow (\forall b \in D_y, \exists a \in D_y : (a,b) \in C)\,.$$

A CSP is said to be *directionally arc consistent w.r.t.* $\prec$ if all its binary constraint are directionally arc consistent w.r.t. $\prec$.

Example 2.97. The CSP $\langle x < y;\ x \in \{1,2,3\},\ y \in \{2,3\}\rangle$ is not arc consistent but directionally arc consistent w.r.t. the ordering $y \prec x$. ♣

The concept of hyper-arc consistency was introduced by Mohr and Masini [1988] under the name *arc consistency.* A similar concept also implicitly appears in [Davis 1987]. The adopted terminology follows Marriott and Stuckey [1998].

Definition 2.98 (Hyper-Arc Consistency). A k-ary constraint C on the variables $x_1, \ldots, x_k$; which are associated with the domains $D_1, \ldots, D_k$, respectively; is called *hyper-arc consistent* if each domain D_i is the projection of C on x_i; that is,

$$\forall i \in \{1, \ldots, k\} : D_i = C[x_i].$$

A CSP is said to be *hyper-arc consistent* if all its constraint are hyper-arc consistent.

Example 2.99. The CSP $\langle x \wedge y = z;\ x \in \{1\},\ y \in \{0,1\},\ z \in \{0,1\}\rangle$ is hyper-arc consistent. The CSP $\langle x \wedge y = z;\ x \in \{0,1\},\ y \in \{0,1\},\ z \in \{1\}\rangle$ is not hyper-arc consistent because there are no values for x and z when $y = 0$. ♣

2.3.2.3. Path Consistency

A form of path consistency was first studied by Montanari [1974]. The version presented here is due to Mackworth [1977].

Definition 2.100 (Path Consistency). A CSP is said to be *path consistent* if, for each subset $\{x, y, z\}$ of its variables, we have

$$\forall (a, c) \in \mathbf{C}_{\{x,z\}}, \exists b : (a, b) \in \mathbf{C}_{\{x,y\}} \wedge (b, c) \in \mathbf{C}_{\{y,z\}}.$$

Note 2.101. Note that a CSP with nonempty domains and with no binary constraints is path consistent. Note also that a CSP with all empty domains is path consistent.

Definition 2.102 (Strong Path Consistency). A CSP is said to be *strongly path consistent* if it is arc and path consistent.

A trivial variant of path consistency is k-path consistency, where every path connecting two nodes is considered. However, this variant is equivalent to the above definition of path consistency, that result is due to Montanari [1974].

Theorem 2.103 (k-Path Consistency). Let $1 < k \in \mathbb{N}$. Then a CSP is *path consistent* if and only if, for each subset $\{x_0, \dots, x_k\}$ of its variables, we have

$$\forall (a_0, a_k) \in \mathbf{C}_{\{x_0,x_k\}}, \exists a_1, \dots, a_{k-1} : \left(\forall i \in \{0, \dots, k-1\} : (a_i, a_{i+1}) \in \mathbf{C}_{\{x_i,x_{i+1}\}}\right).$$

Proof. See the proof of Theorem 5.23 in [Apt 2003, p. 154]. ■

Example 2.104. The CSP $\langle x < y,\ y < z;\ x \in \{0, 1\},\ y \in \{1, 2\},\ z \in \{3, 4\}\rangle$ is path consistency. The CSP $\langle x < y,\ y < z;\ x \in \{0, 1\},\ y \in \{1, 2\},\ z \in \{2, 3, 4\}\rangle$ is not path consistent because there is no value for y when $(x, z) = (1, 2)$. ♣

Theorem 2.103 says, in other words, that path consistent is equivalent to k-path consistency for the *completion of constraint graphs*[4]. Bliek and Sam-Haroud [1999] found that this result also holds for triangulated graphs. Moreover, they proved that strong path consistency on the completion of a constraint graph is equivalent to strong path consistency on the *triangulation of a constraint graph*[5] for *convex CSPs*. Their empirical results show that there is only a little difference between two cases for nonconvex CSPs.

Another variant of path consistency is directional arc consistency, which is devised for directed network of constraints. It is due to Dechter and Pearl [1988].

Definition 2.105 (Directional Path Consistency). Consider a total order $\prec$ on the considered variables. A CSP is said to be *directionally path consistent* if for each subset $\{x, y, z\}$ of its variables, we have

$$x \prec y \wedge z \prec y \Rightarrow \left(\forall (a, c) \in \mathbf{C}_{\{x,z\}}, \exists b : (a, b) \in \mathbf{C}_{\{x,y\}} \wedge (b, c) \in \mathbf{C}_{\{y,z\}}\right).$$

[4] The completion of a constraint graph is to make it complete by adding a universal arc in place of a missing arc.
[5] The triangulation of a constraint graph is to make the graph triangulated by adding a universal arc in place of each missing arc.

Example 2.106. The CSP $\langle x < y,\ y < z;\ x \in \{0,1\},\ y \in \{1,2\},\ z \in \{2,3,4\}\rangle$ is not path consistent but directionally path consistent w.r.t. the ordering $x \prec y \prec z$. It is not directionally arc consistent w.r.t. the ordering $x \prec z \prec y$ because there is no value for y when $(x,z) = (1,2)$. ♣

2.3.2.4. *k*-Consistency

The concepts of k-consistency and strong k-consistency were first introduced by Freuder [1978].

Definition 2.107 (*k*-Consistency). A CSP is said to be 1-*consistent* if it is node consistent. A CSP $\mathcal{P} \equiv (\mathcal{V}, \mathcal{D}, \mathcal{C})$ is said to be k-*consistent*, where $1 < k \in \mathbb{N}$, if
(i) for every consistent $(k-1)$-instantiation l of any subsequence X of $\mathcal{V}$;
(ii) for every variable x in $\mathcal{V} \setminus X$;
there exists a value v in the domain of x such that the k-instantiation composed of l and $\langle x, v\rangle$ is consistent w.r.t. $\mathcal{P}$ (see Definition 2.10).

Definition 2.108 (Strong *k*-Consistency). A CSP is said to be *strong* k-*consistent*, where $k \in \mathbb{Z}_+$, if it is i-consistent for $i = 1, \ldots, k$.

Note 2.109. If the assumption (i) in Definition 2.107 does not hold (that is, there is no consistent $(k-1)$-instantiation), then the CSP is k-consistent.

Note 2.110. Note that every CSP with all empty domains is k-consistent. If a CSP with n variables is n-consistent, then it is k-consistent for all $k \geq n$, because the assumption (ii) in Definition 2.107 does not hold.

Example 2.111. The CSP $\langle x < y,\ x + y = z;\ x \in \{0,1\},\ y \in \{1,2\},\ z \in \{2,3,4\}\rangle$ is 1-consistent and 2-consistent. Hence, it is strong 2-consistent. It is however neither 3-consistent nor strong 3-consistent because the 2-compound label $(1,4)$ of (x,z) cannot be extended. ♣

Theorem 2.112. If a CSP with n variables, where $n \geq 1$, is strong n-consistent and at least one domain is nonempty, then it is consistent.

Proof. See the proof of Theorem 5.38 in [Apt 2003, p. 165]. ■

The concept of k-consistency generalizes the concepts of node, arc, and path consistency in the following sense.

Theorem 2.113. We have that
- 1-consistency is equivalent to node consistency;
- 2-consistency is equivalent to arc consistency;
- 3-consistency is equivalent to path consistency, for binary CSPs;
- k-consistency implies $(k-1)$-consistency.

2.3.3. Local Consistency Notions for Numerical Constraints

In general, we cannot achieve arc/hyper-arc consistency exactly for numerical constraints under a finite number system such as the floating-point number system. As mentioned in [Granvilliers 1998], for a constraint C on a sequence of variables $(x_1, \ldots, x_k)$ associated with the domains $(D_1, \ldots, D_k)$, it would be ideal if each domain D_i can be reduced to the smallest (w.r.t. the point set inclusion) union of intervals that covers the projection of C on x_i instead of being reduced to the exact projection itself (as required by hyper-arc consistency). The property that D_i is the smallest union $\mathcal{U}_i$ of a finite number of intervals containing the projection $C[x_i]$ can be called *interval hyper-arc consistency* or *interval consistency* for short. However, interval hyper-arc consistency is still intractable, in general. Therefore, one often replaces it by a weaker property, for example, that D_i is an interval containing $C[x_i]$. Two fundamental local consistency notions for numerical constraints follow this approach. They are *hull consistency* and *box consistency*. Other local consistency notions are more complicated. However, they still attempt to approximate the projection of a constraint coarsely by an interval at each step. For this purpose, mathematical tools such as interval arithmetic and its variants (see Section 2.2) are very useful in providing cheap estimations of the ranges of functions, hence in providing cheap estimations of the projections of constraints.

2.3.3.1. Hull Consistency

Instead of achieving interval (arc/hyper-arc) consistency, one may just wish to reduce a domain to the smallest interval contains the projection of constraints. This is the idea introduced by Benhamou and Older [1992, 1997], and called *hull consistency* afterwards. The concept of *hull consistency* is defined as follows (see Definition 2.24 for related notations).

Definition 2.114 (Hull Consistency). Let $\mathcal{P} \equiv (\mathcal{V}, \mathcal{D}, \mathcal{C})$ be a CSP, where $\mathcal{V} \equiv (x_1, \ldots, x_n)$ and $\mathcal{D} \subseteq \mathbb{R}^n$; $\mathcal{D}' \equiv D_1 \times \cdots \times D_n \subseteq \mathbb{R}^n$ a subset of $\mathcal{D}$; and C a constraint on $\mathcal{V}$. We define that

- C is said to be *hull consistent in* $\mathcal{D}'$ *on* x_i if $D_i = \square(D_i \cap C[x_i])$;
- C is said to be *hull consistent in* $\mathcal{D}'$ if it is hull consistent in $\mathcal{D}'$ on x_i for all $i = 1, \ldots, n$;
- $\mathcal{P}$ is said to be *hull consistent in* $\mathcal{D}'$ if all its constraints are hull consistent in $\mathcal{D}'$;
- $\mathcal{P}$ is said to be *globally hull consistent in* $\mathcal{D}'$ if the intersection of all its constraints is hull consistent in $\mathcal{D}'$.

In general, we cannot compute the real interval hull of a subset of $\mathbb{R}$ exactly, especially on computers. In the floating-point number system $\mathbb{F}$, one can replace the real interval hull consistency by the following variant.

Definition 2.115 ($\mathbb{F}$-Hull Consistency). Using the same notations as in Definition 2.114, we define that

- C is said to be $\mathbb{F}$-*hull consistent in* $\mathcal{D}'$ *on* x_i if $D_i = \square^{\mathbb{F}}(D_i \cap C[x_i])$;
- C is said to be $\mathbb{F}$-*hull consistent in* $\mathcal{D}'$ if it is $\mathbb{F}$-hull consistent in $\mathcal{D}'$ on x_i for all $i = 1, \ldots, n$;
- $\mathcal{P}$ is said to be $\mathbb{F}$-*hull consistent in* $\mathcal{D}'$ if all its constraints are $\mathbb{F}$-hull consistent in $\mathcal{D}'$;
- $\mathcal{P}$ is said to be *globally* $\mathbb{F}$-*hull consistent in* $\mathcal{D}'$ if the intersection of all its constraints is $\mathbb{F}$-hull consistent in $\mathcal{D}'$.

Note 2.116. In Definition 2.114 and Definition 2.115, the suffix "... *hull consistent in* $\mathcal{D}'$" can be reduced to "... *hull consistent*" when $\mathcal{D}' = \mathcal{D}$, provided that the context is clear.

Remark 2.117. In practice, when referring to the concept of hull consistency, one often refers to the $\mathbb{F}$-hull consistency. However, in numerical analysis, one often refers to the real hull consistency.

Since the $\mathbb{F}$-interval hull of a real set is a closed connected set, every variable domain of an $\mathbb{F}$-hull consistent constraint is also a closed connected set. In contrast, the variable domains of a hull consistent constraint are connected, but not necessarily closed. Let us take some examples to distinguish hull consistency and $\mathbb{F}$-hull consistency.

Example 2.118. It is easy to verify that

- The CSP $\langle x^2 + y^2 = 1;\ x \in [-2, 2],\ y \in [-1, 1]\rangle$ is neither hull consistent nor $\mathbb{F}$-hull consistent because its unique constraint is neither hull nor $\mathbb{F}$-hull consistent on x;
- The CSP $\langle x^2 + y^2 \leq 1;\ x \in [-1, 1],\ y \in [-1, 1]\rangle$ is both hull consistent and $\mathbb{F}$-hull consistent;
- The CSP $\langle x^2 + y^2 < 1;\ x \in [-1, 1],\ y \in [-1, 1]\rangle$ is $\mathbb{F}$-hull consistent, but not hull consistent;
- The CSP $\langle x^2 + y^2 < 1;\ x \in \,]-1, 1[,\ y \in \,]-1, 1[\rangle$ is hull consistent, but not $\mathbb{F}$-hull consistent. Note that one cannot reduce the domains of this problem to achieve $\mathbb{F}$-hull consistency without loss of solutions. ♣

From these examples, We see that hull consistency neither implies nor is implied by $\mathbb{F}$-hull consistency. They are however nearly resemble.

In practice, the simplest way of achieving $\mathbb{F}$-hull consistency for a constraint C is to use a bisection search for the outermost canonical intervals of the projection $C[x_i]$ (see Section 3.2.2.1). More efficient solution techniques do not enforce hull consistency on initial (supposed to be factorable) constraints but often decompose factorable constraints into *primitive constraints* by introducing auxiliary variables, and then enforces $\mathbb{F}$-hull consistency on obtained primitive constraints rather than directly on initial constraints. Since the primitive constraints are very simple, it is possible to obtain $\mathbb{F}$-hull consistency for them. Later, a constraint propagation procedure is performed on a virtual network connecting all primitive constraints in order to propagate the effect of domain reduction. See Section 3.2.2.2 for a recent method of achieving hull consistency on the system of primitive constraints. The reader can find in [Cruz and Barahona 2003] a complicated search method for achieving global $\mathbb{F}$-hull consistency, that is, for computing the $\mathbb{F}$-interval hull of the solution set of an NCSP.

2.3.3.2. *k*B-Consistency

When solving numerical CSPs by using domain reduction techniques adapted from interval arc/hyper-arc consistency, the *early quiescence* problem [Davis 1987] may occurs. In order to tackle this problem, Lhomme [1993] introduced the concept of consistency, called *k*B-*consistency*. It is stronger than hull consistency. 2B-*consistency* is called *arc*-B *consistency*

[Lhomme 1993]. The idea of kB-consistency is to recursively reduce the domains of a CSP. Ideally, a domain of a CSP $\mathcal{P}$ is wished to be gradually reduced to the smallest interval containing the solution set of $\mathcal{P}$. In practice, a facet σ of the box $\mathbf{x} \in \mathbb{I}^n$ defining the domains of a CSP of n variables should be inwardly moved whenever it has been certified that σ contains no solution. Checking if σ contains a solution is a hard work. Alternatively, every facet σ' of σ is inwardly moved whenever it has been certified that σ' contains no solution. Checking if σ' contain a solution is hard too; therefore, it is replaced by an inward movement, and so forth. Here is the formal definition of kB-consistency.

Definition 2.119 (kB-Consistency). Recursively, we define that

- An NCSP $\mathcal{P} \equiv (\mathcal{V}, \mathcal{D}, \mathcal{C})$ is said to be 2B-*consistent in* $\mathcal{D}'$, where $\mathcal{D}' \subseteq \mathcal{D}$, if, for every variable x in $\mathcal{V}$ and every $C \in \mathcal{C}$, the domain D_x of x in $\mathcal{D}'$ is a closed interval $[l, u] \in \mathbb{I}$ such that $l, u \in C[x]$. In addition, if $\mathcal{D} = \mathcal{D}'$ we just say that $\mathcal{P}$ is 2B-*consistent*.
- For $2 \leq k \in \mathbb{N}$, let $\Phi_{k\mathrm{B}}$ be a function that maps every NCSP $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ to a kB-consistent NCSP $\mathcal{P}^* = (\mathcal{V}, \mathcal{D}^*, \mathcal{C})$ which is equivalent to $\mathcal{P}$ such that $\mathcal{D}^* \subseteq \mathcal{D}$ and that there is no kB-consistent NCSP $\mathcal{P}' = (\mathcal{V}, \mathcal{D}', \mathcal{C})$ which is equivalent to $\mathcal{P}$ with $\mathcal{D}^* \subsetneq \mathcal{D}' \subseteq \mathcal{D}$. In case there is no such $\mathcal{P}^*$, we just let $\mathcal{P}^* = \emptyset$.
- An NCSP $\mathcal{P} \equiv (\mathcal{V}, \mathcal{D}, \mathcal{C})$ is said to be kB-*consistent*, where $3 \leq k \in \mathbb{N}$, if, for every variable x in $\mathcal{V}$, the domain D_x of x is a closed interval $[l, u]$ satisfying

$$\Phi_{(k-1)\mathrm{B}}(\mathcal{P}|_{x=l}) \neq \emptyset \wedge \Phi_{(k-1)\mathrm{B}}(\mathcal{P}|_{x=u}) \neq \emptyset. \tag{2.57}$$

  Similarly, $\mathcal{P}$ is said to be kB-*consistent in* $\mathcal{D}'$, where $\mathcal{D}' \subseteq \mathcal{D}$, if the CSP $(\mathcal{V}, \mathcal{D}', \mathcal{C})$ is kB-consistent.

Note 2.120. For a compact NCSP $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ (i.e., $\mathcal{D}$ and $\mathcal{C}$ consist of compact sets), there always exists a unique compact kB-consistent NCSP $\mathcal{P}^* = (\mathcal{V}, \mathcal{D}^*, \mathcal{C})$ as described in Definition 2.119. That is, the function $\Phi_{k\mathrm{B}}$ is uniquely defined.

Example 2.121. Consider three compact NCSPs:

$$\begin{aligned}
\mathcal{P}_1 &= \langle x + y = 1,\ y \leq x + 1,\ y \geq x;\ x \in [0, 10],\ y \in [0, 10] \rangle, \\
\mathcal{P}_2 &= \langle x + y = 1,\ y \leq x + 1,\ y \geq x;\ x \in [0, 2],\ y \in [0, 2] \rangle, \\
\mathcal{P}_3 &= \langle x + y = 1,\ y \leq x + 1,\ y \geq x;\ x \in [0.5, 1],\ y \in [1, 1.5] \rangle.
\end{aligned}$$

One can check that $\mathcal{P}_1$ is not 2B-consistency, that $\mathcal{P}_2$ is not 2B-consistency but not 3B-consistency, and that $\mathcal{P}_3$ is 3B-consistency. Moreover, $\mathcal{P}_3 = \Phi_{2\mathrm{B}}(\mathcal{P}_2)$. ♣

Example 2.122. Consider the CSP

$$\mathcal{P} = \langle x^2 + y^2 < 1;\ x \in [-1, 1],\ y \in [-1, 1] \rangle.$$

The problem $\mathcal{P}$ is neither 2B-consistency nor 3B-consistency because $C[x] = C[y] =]-1, 1[$. Note that we cannot reduce its domains without loss of solutions. ♣

One can replace the condition (2.57) by an equivalent condition:

$$\Phi_{(k-1)\mathrm{B}}(\mathcal{P}|^*_{x=l}) \neq \emptyset \wedge \Phi_{(k-1)\mathrm{B}}(\mathcal{P}|^*_{x=u}) \neq \emptyset, \tag{2.58}$$

where the notation $\mathcal{P}|^*_{x=a}$ denotes the cross section of $\mathcal{P}$ (see Definition 2.32); hence, it has less variables than the singleton CSP $\mathcal{P}|_{x=a}$. They are however defined on real numbers, not on floating-point numbers. For practical use, one may replace the 2B-consistency with $\mathbb{F}$-hull consistency and (2.57) with the following

$$\Phi_{(k-1)\mathrm{B}}(\mathcal{P}|_{x=l^+}) \neq \emptyset \wedge \Phi_{(k-1)\mathrm{B}}(\mathcal{P}|_{x=u^-}) \neq \emptyset, \tag{2.59}$$

or more precisely, we define a variant of kB-consistency as follows.

Definition 2.123 (kB($\mathbb{F}$)-Consistency). Recursively, we define that

- An NCSP $\mathcal{P} \equiv (\mathcal{V}, \mathcal{D}, \mathcal{C})$ is said to be 2B($\mathbb{F}$)*-consistent in* $\mathcal{D}'$, where $\mathcal{D}' \subseteq \mathcal{D}$, if it is $\mathbb{F}$-hull consistent in $\mathcal{D}'$. In addition, if $\mathcal{D} = \mathcal{D}'$ we just say that $\mathcal{P}$ is 2B($\mathbb{F}$)*-consistent*.
- For $2 \leq k \in \mathbb{N}$, let $\Phi^{\mathbb{F}}_{k\mathrm{B}}$ be an function that maps an NCSP $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ to a kB($\mathbb{F}$)-consistent NCSP $\mathcal{P}^* = (\mathcal{V}, \mathcal{D}^*, \mathcal{C})$ which is equivalent to $\mathcal{P}$ such that $\mathcal{D}^* \subseteq \mathcal{D}$ and that there is no kB($\mathbb{F}$)-consistent NCSP $\mathcal{P}' = (\mathcal{V}, \mathcal{D}', \mathcal{C})$ which is equivalent to $\mathcal{P}$ with $\mathcal{D}^* \subsetneq \mathcal{D}' \subseteq \mathcal{D}$. In case there is no such $\mathcal{P}^*$, we just let $\mathcal{P}^* = \emptyset$.
- An NCSP $\mathcal{P} \equiv (\mathcal{V}, \mathcal{D}, \mathcal{C})$ is said to be kB($\mathbb{F}$)*-consistent*, where $3 \leq k \in \mathbb{N}$, if, for every variable x in $\mathcal{V}$, the domain D_x of x is a closed interval $[l, u]$ satisfying

$$\Phi^{\mathbb{F}}_{(k-1)\mathrm{B}}(\mathcal{P}|_{x\in[l,l^+]}) \neq \emptyset \wedge \Phi^{\mathbb{F}}_{(k-1)\mathrm{B}}(\mathcal{P}|_{x\in[u^-,u]}) \neq \emptyset. \tag{2.60}$$

  Similarly, $\mathcal{P}$ is said to be kB($\mathbb{F}$)*-consistent in* $\mathcal{D}'$, where $\mathcal{D}' \subseteq \mathcal{D}$, if the CSP $(\mathcal{V}, \mathcal{D}', \mathcal{C})$ is kB($\mathbb{F}$)-consistent.

The notation $\mathcal{P}|_{x\in[a,b]}$ in (2.60) denotes the slice of $\mathcal{P}$ in the slot $x \in [a, b]$ (see Definition 2.33). The reader can find in Section 3.2.2.5 a short description of a very recent method for maintaining kB($\mathbb{F}$)-consistency.

2.3.3.3. Box Consistency

The concept of *box consistency* was first introduced by Benhamou *et al.* [1994] to address the problem that multiple occurrences of a variables in an expression may lead to unacceptable overestimates of the ranges of functions. Here is the formal definition. (See Definition 2.81 for the concept of an interval form of a relation.)

Definition 2.124 (Box Consistency). Let $\mathcal{P} \equiv (\mathcal{V}, \mathcal{D}, \mathcal{C})$ be a CSP, where $\mathcal{V} \equiv (x_1, \ldots, x_n)$ and $\mathcal{D} \subseteq \mathbb{R}^n$; $\mathbf{x} \equiv (\mathbf{x}_1, \ldots, \mathbf{x}_n) \in \mathbb{I}^n$ a subset of $\mathcal{D}$; C a constraint on $\mathcal{V}$; $[C]$ an interval form of C; and $[\mathcal{P}]$ an interval form of all the constraints of $\mathcal{P}$. We define that

- C is said to be *box consistent in* $\mathbf{x}$ *on* x_i *w.r.t.* $[C]$, where $i \in \{1, \ldots, n\}$, if

$$\mathbf{x}_i = \square\{a \in \mathbf{x}_i \mid (\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, \square\{a\}, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_k) \in [C]\}; \tag{2.61}$$

- C is said to be *box consistent in* $\mathbf{x}$ *w.r.t.* $[C]$ if it is box consistent in $\mathbf{x}$ on x_i w.r.t. $[C]$ for all $i = 1, \ldots, n$;
- $\mathcal{P}$ is said to be *box consistent in* $\mathbf{x}$ *w.r.t.* $[\mathcal{P}]$ if every constraint of $\mathcal{P}$ is box consistent in $\mathbf{x}$ w.r.t. $[\mathcal{P}]$; in addition, if $\mathcal{D} = \mathbf{x}$ we just say that $\mathcal{P}$ is *box consistent w.r.t.* $[\mathcal{P}]$.

Let denotes $\mathbf{x}_i = [\underline{x}_i, \overline{x}_i]$, where $i = 1, \ldots, k$. The condition (2.61) can be replaced by an equivalent condition:

$$(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, \mathbf{x}_i \cap [\underline{x}_i, \underline{x}_i^+], \mathbf{x}_{i+1}, \ldots, \mathbf{x}_k) \in [C] \tag{2.62a}$$
$$\wedge \quad (\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, \mathbf{x}_i \cap [\overline{x}_i^-, \overline{x}_i], \mathbf{x}_{i+1}, \ldots, \mathbf{x}_k) \in [C]. \tag{2.62b}$$

Benhamou *et al.* [1999]; Granvilliers *et al.* [1999] introduced two variants of box consistency: *box*(Γ) *consistency* and *box*$\langle\pm\varphi\rangle$ *consistency*, which can be viewed as an extended version and a weak version of box consistency, respectively. Hereafter, we recall these concepts.

Definition 2.125 (Box(Γ) Consistency). Let $\mathcal{P} \equiv (\mathcal{V}, \mathcal{D}, \mathcal{C})$ be a CSP, where $\mathcal{V} \equiv (x_1, \ldots, x_n)$ and $\mathcal{D} \subseteq \mathbb{R}^n$; $\mathbf{x} \equiv (\mathbf{x}_1, \ldots, \mathbf{x}_n) \in \mathbb{I}^n$ a subset of $\mathcal{D}$; C a constraint in $\mathcal{C}$; and $\Gamma = (\Gamma_1, \ldots, \Gamma_n)$ be a sequence of interval forms of all the constraints of $\mathcal{P}$. We define that

- C is said to be *box*(Γ) *consistent in* $\mathbf{x}$ *on* x_i, where $i \in \{1, \ldots, n\}$, if

$$\mathbf{x}_i = \square\{a \in \mathbf{x}_i \mid (\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, \square\{a\}, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_k) \in \Gamma_i\}; \tag{2.63}$$

- C is said to be *box*(Γ) *consistent in* $\mathbf{x}$ if it is box(Γ) consistent in $\mathbf{x}$ on x_i for all $i = 1, \ldots, n$;
- $\mathcal{P}$ is said to be *box*(Γ) *consistent in* $\mathbf{x}$ if every constraint of $\mathcal{P}$ is box(Γ) consistent in $\mathbf{x}$; in addition, if $\mathcal{D} = \mathbf{x}$ we just say that $\mathcal{P}$ is *box*(Γ) *consistent*.

If $\Gamma = ([C], \ldots, [C])$, then *box*$(\Gamma)$ *consistency* is equivalent to *box consistency w.r.t.* $[C]$.

Notation 2.126. We can replace the notation "box$(([C]_1, \ldots, [C]_n))$ consistent" with "box$([C]_1, \ldots, [C]_n)$ consistent" for short, also with "box consistent w.r.t. $([C]_1, \ldots, [C]_n)$".

Definition 2.127 (Box⟨±φ⟩ Consistency). Let φ be a positive parameter. Using the same notations as in Definition 2.124. We define that

- C is said to be *box*$\langle\pm\varphi\rangle$ *consistent in* $\mathbf{x}$ *on* x_i *w.r.t.* $[C]$, where $i \in \{1, \ldots, n\}$, if

$$\mathbf{x}_i = \square\{a \in \mathbf{x}_i \mid (\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, \square\{a - \varphi, a + \varphi\}, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_k) \in [C]\}; \tag{2.64}$$

- C is said to be *box*$\langle\pm\varphi\rangle$ *consistent in* $\mathbf{x}$ *w.r.t.* $[C]$ if it is box$\langle\pm\varphi\rangle$ consistent in $\mathbf{x}$ on x_i w.r.t. $[C]$ for all $i = 1, \ldots, n$;
- $\mathcal{P}$ is said to be *box*$\langle\pm\varphi\rangle$ *consistent in* $\mathbf{x}$ *w.r.t.* $[\mathcal{P}]$ if every constraint of $\mathcal{P}$ is box$\langle\pm\varphi\rangle$ consistent in $\mathbf{x}$ w.r.t. $[\mathcal{P}]$; in addition, if $\mathcal{D} = \mathbf{x}$ we just say that $\mathcal{P}$ is *box*$\langle\pm\varphi\rangle$ *consistent w.r.t.* $[\mathcal{P}]$.

Note 2.128. In the original definition in [Granvilliers *et al.* 1999] the term $\square\{a - \varphi, a + \varphi\}$ in (2.64) is replaced with $\square\{a, a + \varphi\}$. We use $\square\{a - \varphi, a + \varphi\}$ since it is symmetric.

When maintained on the same constraint C, box consistency and its variants differ from hull consistency on the fact that

- box consistency and its variants are implied by hull consistency;
- box consistency and its variants depend on interval forms of constraints, while hull consistency does not.

To achieve box consistency for a constraint C, one often performs a bisection search until the outermost thick canonical intervals that are not excluded by $[C]$ are found. The reader can find in Section 3.2.2.3 and Section 3.2.2.4 two typical methods for achieving box consistency by search and propagation, respectively.

2.3.4. Extended Local Consistency Notions

2.3.4.1. (i, j)-Consistency

Freuder [1985] generalized the concepts of k-consistency and strong k-consistency (see Section 2.3.2.4) to the concepts of (i, j)-*consistency* and *strong* (i, j)-*consistency*, respectively.

Definition 2.129 ((i, j)-Consistency). A CSP $\mathcal{P} \equiv (\mathcal{V}, \mathcal{D}, \mathcal{C})$ is (i, j)-*consistent* iff
(i) for every consistent i-instantiation l_X of any subsequence X of $\mathcal{V}$; and
(ii) for every subsequence Y of at most j variables of $\mathcal{V}$ such that $X \cap Y = \emptyset$;
there exists an instantiation l_Y of Y such that the combination of l_X and l_Y (sorted in the order of $\mathcal{V}$) is consistent w.r.t. $\mathcal{P}$.

Definition 2.130 (Strong (i, j)-Consistency). A CSP is said to be *strong* (i, j)-*consistent*; where $i, j \in \mathbb{Z}_+$; if it is (k, j)-consistent for $k = 1, \ldots, i$.

The concepts of (i, j)-consistency and strong (i, j)-consistency generalize the concepts of k-consistency and strong k-consistency, respectively, in the following sense.

Theorem 2.131. We have that
- $(k - 1, 1)$-consistency is equivalent to k-consistency;
- $(k - 1, j)$-consistency implies k-consistency for $j > 1$;
- strong $(k - 1, 1)$-consistency is equivalent to strong k-consistency;
- strong $(k - 1, 1)$-consistency implies strong k-consistency for $j > 1$;
- strong (i, j)-consistency implies strong (i', j)-consistency for $i > i'$.

2.3.4.2. Relational Consistency

Dechter and van Beek [1995, 1997] introduced the concept of *relational* (n_v, n_c)-*consistency*, which is a reminiscence of the (i, j)-consistency. Here is the formal definition.

Definition 2.132 (Relational Consistency). A CSP $\mathcal{P}$ is said to be *relationally* (n_v, n_c)-*consistent* if, for every subset $\mathcal{C}$ of size n_c of its constraints and every subsequence X of length n_v of the variables involving $\mathcal{C}$, we have that every consistent (w.r.t. $\mathcal{P}$) n_v-instantiation of X can be extended to a solution of the CSP obtained from $\mathcal{P}$ by removing all constraints that are not in $\mathcal{C}$ and removing all domains and variables not involving in any constraint in $\mathcal{C}$.

To characterize the relation between the relational consistency notion and other local consistency notions, we recall here some straightforward properties [Apt 2003, Note 5.41].

Theorem 2.133. We have that
- A node consistent binary CSP is arc consistent iff it is relationally $(1,1)$-consistent;
- A node consistent CSP is hyper-arc consistent iff it is relationally $(1,1)$-consistent;
- Every node consistent normalized relationally $(2,3)$-consistent CSP is path consistent;
- Every strictly binary relationally $(k-1,k)$-consistent CSP is k-consistent;
- A CSP with m constraints is consistent if and only if it is relationally $(0,m)$-consistent.

Relational $(k, k-1)$-Consistency for NCSPs. Sam-Haroud and Faltings [1996] found that relational $(3,2)$-consistency on convex ternary NCSPs is a sufficient condition for obtaining global consistency. The result is based on Helly's theorem (see [Eckhoff 1993]). This also holds for relational $(k,k-1)$-consistency for k-ary numerical constraints. It is however not efficient to maintain $(k,k-1)$-consistency on numerical constraints of high arity because the time complexity is $\mathcal{O}(n^{2k-1})$, where n is the number of variables of the problem. Note that every factorable NCSP can be easily made ternary by repeatedly replacing each binary arithmetic subexpression in constraints with an auxiliary variable until the arity of every constraint is at most three (see Section 2.1.3.5). The above sufficient condition also holds for convex 2^k-*tree representation* of numerical constraints. The reader can find more details on *ternarization*-based methods in [Sam-Haroud 1995; Sam-Haroud and Faltings 1996], [Faltings and Gelle 1997; Gelle 1998; Gelle and Faltings 2001], and [Lottaz 1999, 2000].

2.3.4.3. Singleton Consistency

The *singleton consistency* introduced by Prosser *et al.* [2000] can be viewed as an extension of the approach of kB-consistency (see Section 2.3.3.2) to CSPs with discrete domains.

Definition 2.134 (Singleton Consistency). Consider an arbitrary $\mathcal{ANY}$-consistency. A CSP $\mathcal{P}$ is said to be *singleton $\mathcal{ANY}$-consistent* if, for every value a in the domain of each variable x, the singleton CSP $\mathcal{P}|_{x=a}$ is $\mathcal{ANY}$-consistent.

Theoretically, this concept can apply to any consistency notion, including the consistency notions in Section 2.3. However, it might be not relevant for CSPs with continuous domains. The following theorem, which is due to Prosser *et al.* [2000], characterizes some properties of singleton consistency.

Theorem 2.135. With respect to the strongness, we have
- X-consistency is monotonic $\Rightarrow$ singleton X-consistency $\geq$ X-consistency;
- X-consistency $\geq$ Y-consistency $\Rightarrow$ singleton X-consistency $\geq$ singleton Y-consistency;
- strong $(i+1,j)$-consistency $>$ singleton (i,j)-consistency $>$ $(i,j+1)$-consistency.

The reader can find in [Prosser *et al.* 2000] some empirical comparisons.

Chapter 3

An Overview of Solution Methods

As mentioned in the previous chapter, there are some alternative approaches for solving a problem, including, but not limited to, the followings:

1. The *domain-specific* methods;
2. The *stochastic* and *genetic* methods;
3. The *homotopy/continuation* methods;
4. The *local* methods;
5. The *complete* methods, also called the *global* methods.

This chapter almost only focuses on complete methods for solving numerical CSPs. However, in Section 3.2.1 and Section 3.4, for completeness, we also provides pointers to classical complete methods for solving classic CSPs (with discrete domains) and incomplete methods for solving CSPs (including numerical CSPs), respectively.

Notation 3.1. Throughout this chapter, we use the notation I to denote the identity matrix of suitable size, when no confusion may arise.

3.1. Mathematical Solution Methods

The mathematical methods for solving constraint satisfaction problems are ubiquitous and plentiful (see a list of methods in [McNamee 1993, 1997, 1999, 2002, 2003]). Therefore, a detailed overview of them is certainly far out of the scope of a PhD thesis chapter, even a handbook. Alternatively, we present an overview with a search-based categorization of the most fundamental methods and some recent methods. A solution method will be classified into a category to which it closely relates. In case a solution method relates to several categories, it will be described in the most related category and is also linked to others.

3.1.1. Fundamental Interval Fixed Point Methods

Without loss of generality, we now consider the problem of the form (2.10):

$$f(x) \in \mathbf{b},\ x \in \mathbf{x}, \tag{3.1}$$

where f is a function from $D \subseteq \mathbb{R}^n$ to $\mathbb{R}^m$, $\mathbf{b} \in \mathbb{I}^m$, and $\mathbf{x} \in \mathbb{I}^n$. Basically, classic *iterative methods*, such as the standard *Newton iteration* and hundreds of *Newton-like methods* [McNamee

1993, 2002], aim at constructing a sequence $(x^{\langle k\rangle})_{k\geq 0}$ that locally converges to a solution of (3.1). Inspired by this idea, (global) *fixed point methods* aim at constructing a sequence of boxes $(\mathbf{x}^{\langle k\rangle})_{k\geq 0}$, where $\mathbf{x}^{\langle 0\rangle} \equiv \mathbf{x}$, that enjoys two desirable properties:[1]

$$\forall k \in \mathbb{Z}_+ : \mathbf{x}^{\langle k-1\rangle} \supseteq \mathbf{x}^{\langle k\rangle}, \tag{3.2}$$

$$x \in \mathbf{x}, f(x) \in \mathbf{b} \Rightarrow \forall k \geq 0 : x \in \mathbf{x}^{\langle k\rangle}. \tag{3.3}$$

If the condition (3.2) holds, then the sequence $(\mathbf{x}^{\langle k\rangle})_{k\geq 0}$ converges to a box $\mathbf{x}^{\langle\infty\rangle}$ that is contained in all boxes $\mathbf{x}^{\langle k\rangle}$. In addition to that, if the condition (3.3) also holds, then the box $\mathbf{x}^{\langle\infty\rangle}$ contains all solutions of (3.1); that is,

$$\mathbf{x} \cap f^{-1}(\mathbf{b}) \subseteq \mathbf{x}^{\langle\infty\rangle}. \tag{3.4}$$

In practice, one often aims at constructing a function $[g] : \mathbb{I}^n \to \mathbb{I}^n$ that satisfies the following condition:

$$x \in \mathbf{x}, f(x) \in \mathbf{b} \Rightarrow x \in [g](\mathbf{x}), \tag{3.5}$$

and then define a sequence

$$\mathbf{x}^{\langle 0\rangle} \equiv \mathbf{x},\ \mathbf{x}^{\langle k\rangle} \equiv [g](\mathbf{x}^{\langle k-1\rangle}) \quad (k = 1, 2, 3, \dots). \tag{3.6}$$

For convenience, we use the notation

$$[g]^{\langle k\rangle}(\mathbf{x}) \equiv \mathbf{x}^{\langle k\rangle} \tag{3.7}$$

for the sequence in (3.6). If the condition (3.2) holds for the sequence (3.6), then the condition (3.3) also holds.

Example 3.2. For any interval form $[f]$ of f, the *interval function*

$$[g](\mathbf{x}) \equiv \mathbf{x} \cap ([f](\mathbf{x}) + \mathbf{x} - \mathbf{b})$$

satisfies both (3.2) and (3.5). Indeed, the condition (3.2) is directly implied by the definition of $[g]$. If $x \in \mathbf{x}, f(x) \in \mathbf{b}$, then

$$x \in f(x) + x - \mathbf{b} \subseteq f(\mathbf{x}) + \mathbf{x} - \mathbf{b} \subseteq [f](\mathbf{x}) + \mathbf{x} - \mathbf{b} = [g](\mathbf{x}).$$

Therefore, the condition (3.5) follows this. ♣

Definition 3.3 (Strong Convergence). A sequence $(\mathbf{x}^{\langle k\rangle})_{k\geq 0}$ of boxes satisfying (3.2) and (3.3) is said to be *strongly convergent* if one of the followings holds:

- f has a *unique* zero $x^* \in \mathbf{x}$ and $x^* = \lim_{k\to\infty} \mathbf{x}^{\langle k\rangle}$;
- f has no zero in $\mathbf{x}$ and $\mathbf{x}^{\langle k\rangle} = \emptyset$ for some $k \in \mathbb{N}$.

(Recall that a *zero of a function* f is a point z such that $f(z) = 0$.)

The following property implies the condition (3.2) for the sequence defined by (3.6):

$$\forall \mathbf{y} \in \mathbb{I}^n : \mathbf{y} \subseteq \mathbf{x} \Rightarrow [g](\mathbf{y}) \subseteq \mathbf{y}. \tag{3.8}$$

This leads to the following concept.

[1] For simplicity, we restrict the presentation to boxes but other compact convex enclosures are possible.

Definition 3.4 (Interval Contraction ≡ Narrowing/Contracting Operator). A function $[g]$ satisfying (3.8) is called an *interval contraction* [Moore 1966, p. 58]. An interval contraction is also called a *contractor*, *narrowing operator* and *contracting operator* in literature. If there exists, in addition to (3.8), a positive number $\alpha < 1$ such that

$$\mathrm{w}([g](\mathbf{x})) \leq \alpha.\mathrm{w}(\mathbf{x}), \tag{3.9}$$

then $[g]$ is called a *strong interval contraction* with a *contractivity factor* α.

If $[g]$ is a *continuous interval contraction*, then the sequence (3.6) converges to a box $\mathbf{x}^{\langle\infty\rangle}$ which is a fixed point of $[g]$; that is, $\mathbf{x}^{\langle\infty\rangle} = [g](\mathbf{x}^{\langle\infty\rangle})$. The sequence defined by (3.6) is therefore called a *fixed point iteration.* If $[g]$ is a strong interval contraction, then the sequence converges to a real number x_0, which is a fixed point of the function $g(x) \equiv [g]([x, x])$. Therefore, the considered problem has at most one solution, x_0, provided that the condition (3.5) holds.

In Section 3.1.1.1 and Section 3.1.1.2, we recall two fundamental fixed-point methods for enclosing the solution set of a *linear interval equation* (see Definition A.12):

$$Ax = b, \tag{3.10}$$

where $A \in \mathbf{A} \in \mathbb{I}^{m\times n}$, $b \in \mathbf{b} \in \mathbb{I}^m$, and $x \in \mathbf{x} \in \mathbb{I}^n$. It was shown in [Heindl *et al.* 1998] that even the simpler problem of computing the hull of the solution set of (3.10) is NP-hard.

In Section 3.1.1.3, 3.1.1.4 and 3.1.1.5, we recall three families of fundamental fixed-point methods for enclosing the solution set of a nonlinear equation system of the form (2.20):

$$f(x) = 0, \quad x \in \mathbf{x}, \tag{3.11}$$

where $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ is a possibly *nonlinear function* and $\mathbf{x} \in \mathbb{I}^n$ is a box contained in D. We recall that the forms (2.10), (2.20) and (2.21) are mathematically equivalent. Therefore, the fixed point methods that apply to the form (2.20) also apply to the other forms. For these methods, note that if f is *continuously differentiable* on $\mathbf{x}$, then for any interval form $[f']$ of the *derivative* f' of f on $\mathbf{x}$, the *interval Jacobian matrix* $[f'](\mathbf{x})$ is a *Lipschitz matrix* for f on $\mathbf{x}$ (see Corollary 5.1.5 and the discussion in [Neumaier 1990, p. 174–175]).

Throughout this section, we use the following notations for matrices and vectors concerning their components.

Notation 3.5. Denote $A \equiv (A_{ij})_{m\times n}$, $\mathbf{A} \equiv (\mathbf{A}_{ij})_{m\times n}$, $b \equiv (b_1, \ldots, b_m)^{\mathrm{T}}$, $\mathbf{b} \equiv (\mathbf{b}_1, \ldots, \mathbf{b}_m)^{\mathrm{T}}$, $x \equiv (x_1, \ldots, x_n)^{\mathrm{T}}$, and $\mathbf{x} \equiv (\mathbf{x}_1, \ldots, \mathbf{x}_n)^{\mathrm{T}}$.

3.1.1.1. Krawczyk Iteration for Linear Equations

In this section we recall in brief the Krawczyk iteration for enclosing the solution set of the linear interval equation (3.10). It is a specialization of an iterative method for nonlinear systems originated by Krawczyk [1969] and described in detail in [Neumaier 1990, Section 4.2]. See Section 3.1.1.3 for a brief description of the Krawczyk iteration for nonlinear problems.

If $x \in \mathbf{x}$ is a solution of (3.10), then, for all preconditioning matrix $C \in \mathbb{R}^{n\times m}$, we have

$$x = Cb - (CA - I)x \in C\mathbf{b} - (C\mathbf{A} - I)\mathbf{x}.$$

Hence, for all $\mathbf{x} \in \mathbb{I}^n$, we have

$$\mathbf{x} \cap \Sigma(\mathbf{A}, \mathbf{b}) \subseteq \mathbf{x} \cap (C\mathbf{b} - (C\mathbf{A} - I)\mathbf{x}), \tag{3.12}$$

where $\Sigma(\mathbf{A}, \mathbf{b})$ denotes the solution set of (3.10), as defined in Definition A.14. This leads to the following concepts.

Definition 3.6 (Krawczyk Operator for Linear Equations). The *Krawczyk operator* for the linear interval equation (3.10) is defined as

$$\mathfrak{O}_{\mathrm{K}}(\mathbf{A}, \mathbf{b}, C, \mathbf{x}) \equiv C\mathbf{b} - (C\mathbf{A} - I)\mathbf{x}, \tag{3.13}$$

and the *Krawczyk iteration* for the linear interval equation (3.10) is defined as

$$\mathbf{x}^{\langle 0\rangle} \equiv \mathbf{x}, \;\; \mathbf{x}^{\langle k\rangle} \equiv \mathbf{x}^{\langle k-1\rangle} \cap \mathfrak{O}_{\mathrm{K}}(\mathbf{A}, \mathbf{b}, C, \mathbf{x}^{\langle k-1\rangle}) \quad (k \in \mathbb{Z}_+), \tag{3.14}$$

where C is a *preconditioning matrix*.

The Krawczyk iteration (3.14) enjoys the following property:

$$\forall k \in \mathbb{Z}_+ : \mathbf{x} \cap \Sigma(\mathbf{A}, \mathbf{b}) \subseteq \mathbf{x}^{\langle k\rangle} \subseteq \mathbf{x}^{\langle k-1\rangle}. \tag{3.15}$$

Other properties of the Krawczyk iteration can be found in Section 4.2 in [Neumaier 1990]. The width of the fixed point of the Krawczyk iteration is bounded as follows (originated by Neumaier [1987b], see Theorem 4.2.3 and Proposition 4.2.4 in [Neumaier 1990]).

Proposition 3.7. Let $\mathbf{A} \in \mathbb{I}^{n\times n}$, $\mathbf{b}, \mathbf{x} \in \mathbb{I}^n$, and $(\mathbf{x}^{\langle k\rangle})_{k\geq 0}$ the Krawczyk iteration in (3.14).

(*i*) If $\mathbf{A}$ is regular, then, for every enclosure $\mathbf{x} \in \mathbb{I}^n$ of $\Sigma(\mathbf{A}, \mathbf{b})$, we have

$$\square\Sigma(\mathbf{A}, \mathbf{b}) \subseteq \mathbf{x}^{\langle k\rangle} \subseteq \square\Sigma(\mathbf{A}, \mathbf{b}) + (C\mathbf{A} - I)(\mathbf{x}^{\langle k-1\rangle} - \mathbf{x}^{\langle k-1\rangle}), \tag{3.16}$$

$$\delta_{\mathrm{H}}(\mathbf{x}^{\langle k\rangle}, \square\Sigma(\mathbf{A}, \mathbf{b})) \leq 2\,|C\mathbf{A} - I|\,\mathrm{rad}(\mathbf{x}^{\langle k-1\rangle}). \tag{3.17}$$

(*ii*) If the conditioning matrix $C \in \mathbb{R}^{n\times n}$ is chosen such that $\|C\|_u \leq \beta < 1$ for some vector $u \in \mathbb{R}^n : u > 0$, then

$$\|\mathrm{rad}(\square\Sigma(\mathbf{A}, \mathbf{b}))\|_u \leq \|\mathrm{rad}(\lim_{k\to\infty} \mathbf{x}^{\langle k\rangle})\|_u \leq \frac{1+\beta}{1-\beta}\,\|\mathrm{rad}(\square\Sigma(\mathbf{A}, \mathbf{b}))\|_u. \tag{3.18}$$

If $\mathbf{A}$ and $\mathbf{b}$ are thin, then it is possible to obtain optimal enclosures of $\mathbf{A}^{-1}\mathbf{b}$ (see [Ris 1972], [Rump and Kaucher 1980], and [Rump 1984]).

3.1.1.2. Interval Gauss-Seidel Iteration

In this section, we recall in brief the *interval Gauss-Seidel elimination* for enclosing the solution set of the linear interval equation (3.10). It is believed that this method was first studied by Alefeld and Herzberger [1970], and then by Ris [1972]. The version presented in this section is a brief of the one in [Neumaier 1990, Section 4.3].

Let choose m^* arbitrary equations and m^* arbitrary variables of the system (3.10), where $m^* = \min\{m, n\}$. Without loss of generality, we assume that the chosen equations and variables are the first ones. The chosen equations can be explicitly rewritten as

$$A_{ii}x_i = b_i - \sum_{j=1;\ j\neq i}^{n} A_{ij}x_j \quad (i = 1, \ldots, m^*). \tag{3.19}$$

Let define a function $\Gamma : \mathbb{I}^3 \to \mathbb{I}$ such that, for any $\mathbf{c}, \mathbf{d}, \mathbf{y} \in \mathbb{I}$,

$$\Gamma(\mathbf{c}, \mathbf{d}, \mathbf{y}) \equiv \square\{y \in \mathbf{y} \mid cy = d, c \in \mathbf{c}, d \in \mathbf{d}\}. \tag{3.20}$$

In practice, we can use $\Gamma(\mathbf{c}, \mathbf{d}, \mathbf{y}) = \mathbf{y} \cap \mathbf{d}/\mathbf{c}$. It follows from (3.19) that

$$x_i \in \Gamma(\mathbf{A}_{ii},\ \mathbf{b}_i - \sum_{j=1;\ j\neq i}^{n} \mathbf{A}_{ij}\mathbf{x}_j,\ \mathbf{x}_i) \quad (i = 1, \ldots, m^*). \tag{3.21}$$

This leads to defining that $\mathbf{x}^{\langle 0\rangle} \equiv \mathbf{x}$ and that, for all $k > 0$,

$$\mathbf{x}_i^{\langle k\rangle} \equiv \Gamma(\mathbf{A}_{ii},\ \mathbf{b}_i - \sum_{j<i} \mathbf{A}_{ij}\mathbf{x}_j^{\langle k\rangle} - \sum_{j>i} \mathbf{A}_{ij}\mathbf{x}_j^{\langle k-1\rangle},\ \mathbf{x}_i^{\langle k-1\rangle}) \quad (\text{for } i = 1, \ldots, m^*), \tag{3.22a}$$

$$\mathbf{x}_i^{\langle k\rangle} \equiv \mathbf{x}_i^{\langle k-1\rangle} \quad (i = m^* + 1, \ldots, n) \tag{3.22b}$$

Definition 3.8 (Gauss-Seidel Operator). The rule (3.22) is called the *interval Gauss-Seidel iteration*. It defines a mapping $\mathfrak{O}_{\mathrm{GS}} : \mathbb{I}^{m\times n} \times \mathbb{I}^m \times \mathbb{I}^n \to \mathbb{I}^n$, called the *interval Gauss-Seidel operator*, which in turn defines a sequence of boxes:

$$\mathbf{x}^{\langle 0\rangle} = \mathbf{x},\ \mathbf{x}^{\langle k-1\rangle} \mapsto \mathbf{x}^{\langle k\rangle} \equiv \mathfrak{O}_{\mathrm{GS}}(\mathbf{A}, \mathbf{b}, \mathbf{x}^{\langle k-1\rangle}). \tag{3.23}$$

In practice, the iteration (3.22) is applied in conjunction with *preconditioning*, which results in the so-called *preconditioned interval Gauss-Seidel iteration/operator*:

$$\mathbf{x}^{\langle k\rangle} \equiv \mathfrak{O}_{\mathrm{GS}}(C\mathbf{A}, C\mathbf{b}, \mathbf{x}^{\langle k-1\rangle}) \quad (k \in \mathbb{Z}_+), \tag{3.24}$$

where $C \in \mathbb{R}^{n\times m}$ is a *preconditioning matrix*.

Remark 3.9. The rule (3.22) can be analogously extended to the case $m^* \leq \min\{m, n\}$.

The Gauss-Seidel iteration has the next properties ([Neumaier 1990, Proposition 4.3.4]).

Proposition 3.10. Let $\mathbf{A}, \mathbf{A}', \mathbf{A}'' \in \mathbb{I}^{m\times n}$; $C \in \mathbb{R}^{n\times m}$; $\mathbf{b}, \mathbf{b}' \in \mathbb{I}^m$; and $\mathbf{x}, \mathbf{x}' \in \mathbb{I}^n$. Then

$$\mathbf{x} \cap \Sigma(\mathbf{A}, \mathbf{b}) \subseteq \mathfrak{O}_{\mathrm{GS}}(\mathbf{A}, \mathbf{b}, \mathbf{x}) \subseteq \mathbf{x}; \tag{3.25}$$

$$\mathfrak{O}_{\mathrm{GS}}(I, C\mathbf{b}, \mathbf{x}) = C\mathbf{b} \cap \mathbf{x}; \tag{3.26}$$

$$\mathbf{A}\mathbf{x} \cap \mathbf{b} = \emptyset \Rightarrow \mathfrak{O}_{\mathrm{GS}}(\mathbf{A}, \mathbf{b}, \mathbf{x}) = \emptyset; \tag{3.27}$$

$$\mathbf{A}' \subseteq \mathbf{A}, \mathbf{x}' \subseteq \mathbf{x}, \mathbf{b}' \subseteq \mathbf{b} \Rightarrow \mathfrak{O}_{\mathrm{GS}}(\mathbf{A}', \mathbf{b}', \mathbf{x}') \subseteq \mathfrak{O}_{\mathrm{GS}}(\mathbf{A}, \mathbf{b}, \mathbf{x}); \tag{3.28}$$

$$\mathbf{A} = \mathbf{A}' + \mathbf{A}'' \Rightarrow \mathfrak{O}_{\mathrm{GS}}(\mathbf{A}, \mathbf{b}, \mathbf{x}) \subseteq \mathfrak{O}_{\mathrm{GS}}(\mathbf{A}', \mathbf{b} - \mathbf{A}''\mathbf{x}, \mathbf{x}). \tag{3.29}$$

Hereafter is another property on the width of the fixed point of the Gauss-Seidel iteration (see [Neumaier 1990, Proposition 4.3.11]).

Proposition 3.11. Let $\mathbf{A} \in \mathbb{I}^{n\times n}$; $\mathbf{b}, \mathbf{x} \in \mathbb{I}^n$; and $\langle \mathbf{A} \rangle$ the comparison matrix of $\mathbf{A}$. Then

$$\langle \mathbf{A} \rangle \, |\lim_{k\to\infty} \mathfrak{O}_{\mathrm{GS}}^{\langle k\rangle}(\mathbf{A}, \mathbf{b}, \mathbf{x})| \leq |\mathbf{b}|. \tag{3.30}$$

A comparison between the Krawczyk operator and the preconditioned Gauss-Seidel operator for linear interval equations is given as follows (originated by Neumaier [1984], see Theorem 4.3.5 in [Neumaier 1990, p. 138]).

Theorem 3.12. Let $\mathbf{A} \in \mathbb{I}^{m\times n}$, $\mathbf{b} \in \mathbb{I}^m$, $\mathbf{x} \subseteq \mathbf{y} \in \mathbb{I}^n$, and $C \in \mathbb{R}^{n\times m}$. We have

$$\mathfrak{O}_{\mathrm{GS}}(C\mathbf{A}, C\mathbf{b}, \mathbf{x}) \subseteq \mathfrak{O}_{\mathrm{GS}}(I, C\mathbf{b} - (C\mathbf{A} - I)\mathbf{y}, \mathbf{y}) = \mathbf{y} \cap \mathfrak{O}_{\mathrm{K}}(\mathbf{A}, \mathbf{b}, C, \mathbf{y}). \tag{3.31}$$

Theorem 3.12 shows that the Krawczyk iteration for linear systems is indeed a special case of the interval Gauss-Seidel iteration and that, for the same square system of linear equations, the closures produced by the preconditioned Gauss-Seidel iteration are tighter than those produced by the Krawczyk iteration. It implies that the interval Gauss-Seidel iteration cannot be improved by relaxation methods. Empirical comparisons were shown in [Hansen and Sengupta 1981; Ris 1972]. Note that both these iterations enjoy the *quadratic approximation property* (first proved by Gay [1982]).

3.1.1.3. Krawczyk Iteration for Nonlinear Equations

Now consider the problem (3.11). Let $c \in \mathbf{x}$ and $C \in \mathbb{R}^{n\times m}$. If x is a zero of f in $\mathbf{x}$ and $\mathbf{A}$ is a Lipschitz matrix for f on $\mathbf{x}$, then there exists an $m \times n$ real matrix $A \in \mathbf{A}$ such that $f(c) + A(x - c) = f(x) = 0$. Therefore, we have

$$x = x - C(f(c) + A(x - c)) = c - Cf(c) - (CA - I)(x - c) \in c - Cf(c) - (C\mathbf{A} - I)(\mathbf{x} - c). \tag{3.32}$$

Hence,

$$\mathbf{x} \cap f^{-1}(0) \subseteq \mathbf{x} \cap (c - Cf(c) - (C\mathbf{A} - I)(\mathbf{x} - c)). \tag{3.33}$$

This leads to the definitions of the *Krawczyk operator* and *iteration* as follows.

Definition 3.13 (Krawczyk Operator for Nonlinear Equations). The *Krawczyk operator* for the nonlinear system (3.11) is defined as

$$\mathfrak{O}_{\mathrm{K}}(f, \mathbf{A}, C, \mathbf{x}, c) \equiv c - Cf(c) - (C\mathbf{A} - I)(\mathbf{x} - c). \tag{3.34}$$

The *Krawczyk iteration* for the nonlinear system (3.11) is defined as

$$c^{\langle 0\rangle} \in \mathbf{x}^{\langle 0\rangle} \equiv \mathbf{x}, \tag{3.35a}$$

$$c^{\langle k\rangle} \in \mathbf{x}^{\langle k\rangle} \equiv \mathbf{x}^{\langle k-1\rangle} \cap \mathfrak{O}_{\mathrm{K}}(f, \mathbf{A}^{\langle k\rangle}, C^{\langle k\rangle}, \mathbf{x}^{\langle k-1\rangle}, c^{\langle k-1\rangle}) \quad (\text{for } k \in \mathbb{Z}_+). \tag{3.35b}$$

The following theorem gives the main properties of the Krawczyk operator (the parts (*i*) and (*ii*) were found by Krawczyk [1969]; the part (*iii*) by Kahan [1968] and Moore [1977]).

Theorem 3.14 (Krawczyk, Kahan). Let $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ be a *Lipschitz continuous function* on D, $\mathbf{x} \in \mathbb{I}^n$ a box contained in D, $c \in \mathbf{x}$, $C \in \mathbb{R}^{n\times m}$, and $\mathbf{A} \in \mathbb{I}^{m\times n}$ a Lipschitz matrix for f on $\mathbf{x}$. Then

(*i*) $\mathbf{x} \cap f^{-1}(0) \subseteq \mathfrak{O}_{\mathrm{K}}(f, \mathbf{A}, C, \mathbf{x}, c)$;

(*ii*) If $\mathbf{x} \cap \mathfrak{O}_{\mathrm{K}}(f, \mathbf{A}, C, \mathbf{x}, c) = \emptyset$, then f contains no zero in $\mathbf{x}$;

(*iii*) If $c \in \mathrm{int}(\mathbf{x})$ and $\emptyset \neq \mathfrak{O}_{\mathrm{K}}(f, \mathbf{A}, C, \mathbf{x}, c) \subseteq \mathbf{x}$, then $\mathbf{A}$ is strongly regular and f contains a *unique* zero in $\mathbf{x}$ (and hence in the box $\mathfrak{O}_{\mathrm{K}}(f, \mathbf{A}, C, \mathbf{x}, c)$).

Proof. The parts (*i*) and (*ii*) follow the above argument. To prove the part (*iii*), note that the proofs of Theorem 5.1.8 in [Neumaier 1990, p. 177] and related theorems for the case $m = n$ remain valid for the case $m \neq n$ if we interpret the notation A^{-1} as in Definition A.7. ■

Remark 3.15. If we replace the Lipschitz matrix $\mathbf{A}$ in the definition of the Krawczyk operator (3.34) with a *slope matrix* $\mathbf{A}$ for f at $[\mathbf{c}, \mathbf{x}]$, where $c \in \mathbf{c} \subseteq \mathbf{x}$, then, in Theorem 3.14, the parts (*i*) and (*ii*) still hold, and the uniqueness in the part (*iii*) reduces to the *existence* of a solution. Moreover, if the stronger assumption $\emptyset \neq \mathfrak{O}_{\mathrm{K}}(f, \mathbf{A}, C, \mathbf{x}, c) \subseteq \mathrm{int}(\mathbf{x})$ holds, then $\mathbf{x}$ contains a unique zero of f in $\mathbf{x}$. For more details, see [Neumaier 1990, Corollary 5.4.3] and [Schichl and Neumaier 2004a, Theorem 2.2].

Krawczyk [1983] found that if $\mathbf{A}$ is a *strongly regular Lipschitz matrix* on $\mathbf{x}$ and the preconditioning matrix C is chosen such that $\rho(|C\mathbf{A} - I|) = \beta < 1$, then the Krawczyk iteration (3.35) with $c^{\langle k\rangle} = \mathrm{mid}(\mathbf{x}^{\langle k\rangle})$ is strongly convergent and

$$\mathrm{rad}(\mathbf{x}^{\langle k\rangle}) \leq |C\mathbf{A} - I|\, \mathrm{rad}(\mathbf{x}^{\langle k-1\rangle}), \tag{3.36}$$

with noticing that $\lim_{k\to\infty} |C\mathbf{A} - I|^k = 0$ (see [Neumaier 1990, p. 181]). Moreover, the following theorem (found by Krawczyk [1986], see also Theorem 5.1.9 [Neumaier 1990, p. 178]) implies that, in the Krawczyk iteration (3.35), $c^{\langle k\rangle}$ should be chosen as the midpoint of $\mathbf{x}^{\langle k\rangle}$ and the conditioning matrix $C^{\langle k\rangle}$ should be chosen as $\mathrm{mid}(\mathbf{A}^{\langle k\rangle})^{-1}$, whenever it is possible.

Theorem 3.16 (Krawczyk). Let $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ be a *Lipschitz continuous function* on D, $\mathbf{x} \in \mathbb{I}^n$ a box contained in D, $c \in \mathbf{x}$, $C \in \mathbb{R}^{n\times m}$, and $\mathbf{A} \in \mathbb{I}^{m\times n}$ a Lipschitz matrix for f on $\mathbf{x}$. Then

(*i*) $\mathrm{rad}(\mathfrak{O}_{\mathrm{K}}(f, \mathbf{A}, C, \mathbf{x}, \mathrm{mid}(\mathbf{x}))) \subseteq \mathrm{rad}(\mathfrak{O}_{\mathrm{K}}(f, \mathbf{A}, C, \mathbf{x}, c))$;

(*ii*) If $\mathfrak{O}_{\mathrm{K}}(f, \mathbf{A}, C, \mathbf{x}, \mathrm{mid}(\mathbf{x})) \subseteq \mathbf{x}$ and the midpoint matrix $\mathrm{mid}(\mathbf{A})$ is regular, then $\mathfrak{O}_{\mathrm{K}}(f, \mathbf{A}, \mathrm{mid}(\mathbf{x})^{-1}, \mathbf{x}, \mathrm{mid}(\mathbf{A})) \subseteq \mathfrak{O}_{\mathrm{K}}(f, \mathbf{A}, C, \mathbf{x}, \mathrm{mid}(\mathbf{x}))$.

Proof. The proofs of Theorem 5.1.9 in [Neumaier 1990, p. 178] for the case $m = n$ remain valid for the case $m \neq n$ if we interpret the notation A^{-1} as in Definition A.7. ■

Let f be continuously differentiable in D and the specialized Krawczyk operator be

$$\mathfrak{O}'_{\mathrm{K}}(\mathbf{x}) \equiv c - \mathrm{mid}(A)^{-1} f(c) - |\mathrm{mid}(A)^{-1}|\, \mathrm{rad}(A)(\mathbf{x} - c), \tag{3.37}$$

where $c = \mathrm{mid}(\mathbf{x})$, $\mathbf{A} = [f'](\mathbf{x})$, and $[f']$ is an interval extension of the *derivative* of f in D. Krawczyk [1987] found that:

- If $\mathfrak{O}'_{\mathrm{K}}(\mathbf{x}) \subseteq \mathbf{x}$ and $[f'](\mathbf{x})$ is strongly regular, then $\mathfrak{O}'_{\mathrm{K}}(\mathfrak{O}'_{\mathrm{K}}(\mathbf{x})) \subseteq \mathfrak{O}'_{\mathrm{K}}(\mathbf{x})$;
- If $\mathfrak{O}'_{\mathrm{K}}(\mathbf{x}) \subseteq \mathrm{int}(\mathbf{x})$, then the iteration $\{\mathbf{x}^{\langle 0\rangle} \equiv \mathbf{x},\ \mathbf{x}^{\langle k\rangle} = \mathfrak{O}'_{\mathrm{K}}(\mathbf{x}^{\langle k-1\rangle}) \quad (k \in \mathbb{Z}_+)\}$ is a nested sequence of boxes, which converges to the unique zero of f in $\mathbf{x}$.

Moreover, if $[f']$ is the natural interval extension of arithmetic expressions that are Lipschitz on $\mathbf{x}$, then $\|\mathrm{rad}(\mathbf{x}^{\langle k\rangle})\| = \mathcal{O}(\|\mathrm{rad}(\mathbf{x}^{\langle k-1\rangle})\|^2)$ (see Theorem 5.1.15 in [Neumaier 1990, p. 189]).

3.1.1.4. Hansen-Sengupta Iteration

Now consider the problem (3.11). Let $c \in \mathbf{x}$ and $C \in \mathbb{R}^{n\times m}$. If x is a zero of f in $\mathbf{x}$ and $\mathbf{A}$ is a Lipschitz matrix for f on $\mathbf{x}$, then there exists an $m \times n$ real matrix $A \in \mathbf{A}$ such that $A(x-c) = f(x) - f(c) = -f(c)$. Hence, we have

$$\mathbf{x} \cap f^{-1}(0) \subseteq c + \mathfrak{O}_{\mathrm{GS}}(C\mathbf{A},\ -Cf(c),\ \mathbf{x}-c). \tag{3.38}$$

This leads to the definitions of the *Hansen-Sengupta operator* and *iteration* as follows.

Definition 3.17 (Hansen-Sengupta Operator). The *Hansen-Sengupta operator* for the nonlinear system (3.11) is defined as

$$\mathfrak{O}_{\mathrm{HS}}(f, \mathbf{A}, C, \mathbf{x}, c) \equiv c + \mathfrak{O}_{\mathrm{GS}}(C\mathbf{A},\ -Cf(c),\ \mathbf{x}-c). \tag{3.39}$$

The *Hansen-Sengupta iteration* for the nonlinear system (3.11) is defined as

$$c^{\langle 0\rangle} \in \mathbf{x}^{\langle 0\rangle} \equiv \mathbf{x}, \quad c^{\langle k\rangle} \in \mathbf{x}^{\langle k\rangle} \equiv \mathfrak{O}_{\mathrm{HS}}(f, \mathbf{A}^{\langle k\rangle}, C^{\langle k\rangle}, \mathbf{x}^{\langle k-1\rangle}, c^{\langle k-1\rangle}) \quad (\text{for } k \in \mathbb{Z}_+). \tag{3.40}$$

The following theorem gives the main properties of the Hansen-Sengupta operator (the parts (i) and (ii) by Hansen and Sengupta [1981]; the part (iii) by Moore and Qi [1982]).

Theorem 3.18 (Hansen & Sengupta). Let $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ be a *Lipschitz continuous function* on D, $\mathbf{x} \in \mathbb{I}^n$ a box contained in D, $c \in \mathbf{x}$, $C \in \mathbb{R}^{n\times m}$, and $\mathbf{A} \in \mathbb{I}^{m\times n}$ a Lipschitz matrix for f on $\mathbf{x}$. Then

(i) $\mathbf{x} \cap f^{-1}(0) \subseteq \mathfrak{O}_{\mathrm{HS}}(f, \mathbf{A}, C, \mathbf{x}, c)$;

(ii) If $\mathbf{x} \cap \mathfrak{O}_{\mathrm{HS}}(f, \mathbf{A}, C, \mathbf{x}, c) = \emptyset$, then f contains no zero in $\mathbf{x}$;

(iii) If $c \in \mathrm{int}(\mathbf{x})$ and $\emptyset \neq \mathfrak{O}_{\mathrm{HS}}(f, \mathbf{A}, C, \mathbf{x}, c) \subseteq \mathbf{x}$, then $\mathbf{A}$ is strongly regular and f contains a *unique* zero in $\mathbf{x}$ (and hence in the box $\mathfrak{O}_{\mathrm{HS}}(f, \mathbf{A}, C, \mathbf{x}, c)$).

Proof. The parts (i) and (ii) follow the above argument. To prove the part (iii), note that the proofs of Theorem 5.1.8 in [Neumaier 1990, p. 177] and related theorems for the case $m = n$ remain valid for the case $m \neq n$ if we interpret the notation A^{-1} as in Definition A.7. ■

Remark 3.19. If we replace the Lipschitz matrix $\mathbf{A}$ in the definition of the Hansen-Sengupta operator (3.39) with a *slope matrix* $\mathbf{A}$ for f at $[c, \mathbf{x}]$, then the parts (i) and (ii) in Theorem 3.18 still hold, and the uniqueness in the part (iii) reduces to the *existence* of a solution. See [Neumaier 1990, Corollary 5.4.3] for more details.

The Hansen-Sengupta iteration is indeed a nonlinear version of the interval Gauss-Seidel iteration. Similar to the linear case, the Hansen-Sengupt operator provides better enclosures than the Krawczyk operator does, in the following sense [Neumaier 1990, p. 177]:

$$\mathfrak{O}_{\mathrm{HS}}(f, \mathbf{A}, C, \mathbf{x}, c) \subseteq \mathfrak{O}_{\mathrm{K}}(f, \mathbf{A}, C, \mathbf{x}, c). \tag{3.41}$$

In 1979, G. Alefeld found that if $\mathbf{A}$ is a *strongly regular Lipschitz matrix* on $\mathbf{x}$ and $C\mathbf{A}$ is an H-matrix, then the Hansen-Sengupta iteration (3.40) with $c^{\langle k\rangle} = \mathrm{mid}(\mathbf{x}^{\langle k\rangle})$ is *strongly convergent* (see Theorem 5.2.5 in [Neumaier 1990, p. 182]). Thiel [1989] found that if, in addition to the above assumption, the *fixed point inverse* of $C\mathbf{A}$ is regular, then the Hansen-Sengupta iteration (3.40) is *strongly convergent* for all choice of $c^{\langle k\rangle} \in \mathbf{x}^{\langle k\rangle}$ (see Theorem 5.2.6 in [Neumaier 1990, p. 182] and Definition A.15).

3.1.1.5. Interval Newton Iteration

Now consider the problem (3.11). Let $c \in \mathbf{x}$ and $C \in \mathbb{R}^{n\times m}$. If x is a zero of f in $\mathbf{x}$ and $\mathbf{A}$ is a *Lipschitz set* for f on $\mathbf{x}$, then there exists an $m \times n$ real matrix $A \in \mathbf{A}$ such that $CA(x - c) = C(f(x) - f(c)) = -Cf(c)$. Hence, by Definition A.12, we have

$$x \in c + \Sigma(C\mathbf{A}, -Cf(c)) = c - \Sigma(C\mathbf{A}, Cf(c)) \subseteq c - \square\Sigma(C\mathbf{A}, Cf(c)). \tag{3.42}$$

This leads to the definitions of the *interval Newton operator* and *iteration* as follows.

Definition 3.20 (Interval Newton Operator). The *interval Newton operator* for the nonlinear system (3.11) is defined as

$$\mathfrak{O}_{\mathrm{N}}(f, \mathbf{A}, C, \mathbf{x}, c) \equiv c - \square\Sigma(C\mathbf{A}, Cf(c)). \tag{3.43}$$

The *interval Newton iteration* for the system (3.11) is defined as

$$c^{\langle 0\rangle} \in \mathbf{x}^{\langle 0\rangle} \equiv \mathbf{x}, \tag{3.44a}$$
$$c^{\langle k\rangle} \in \mathbf{x}^{\langle k\rangle} \equiv \mathbf{x}^{\langle k-1\rangle} \cap \mathfrak{O}_{\mathrm{N}}(f, \mathbf{A}^{\langle k\rangle}, C^{\langle k\rangle}, \mathbf{x}^{\langle k-1\rangle}, c^{\langle k-1\rangle}) \quad (\text{for } k \in \mathbb{Z}_+). \tag{3.44b}$$

The following theorem summarizes the main properties of the interval Newton operator (the parts (*i*) and (*ii*) by Moore [1966]; the part (*iii*) for differentiable functions by Nickel [1971], and latter extended in [Neumaier 1990, Theorem 5.1.7]; see also [Nickel 1981]).

Theorem 3.21 (Moore, Nickel). Let $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ be a *Lipschitz continuous function* on D, $\mathbf{x} \in \mathbb{I}^n$ a box contained in D, $c \in \mathbf{x}$, $C \in \mathbb{R}^{n\times m}$, and $\mathbf{A} \in \mathbb{I}^{m\times n}$ a *Lipschitz set* for f on $\mathbf{x}$. Then

(*i*) $\mathbf{x} \cap f^{-1}(0) \subseteq \mathfrak{O}_{\mathrm{N}}(f, \mathbf{A}, C, \mathbf{x}, c)$;

(*ii*) If $\mathbf{x} \cap \mathfrak{O}_{\mathrm{N}}(f, \mathbf{A}, C, \mathbf{x}, c) = \emptyset$, then f contains no zero in $\mathbf{x}$;

(*iii*) If $\mathbf{A}$ is regular, $c \in \mathrm{int}(\mathbf{x})$ and $\mathfrak{O}_{\mathrm{N}}(f, \mathbf{A}, C, \mathbf{x}, c) \subseteq \mathbf{x}$, then f contains a *unique* zero in $\mathbf{x}$ (and hence in the box $\mathfrak{O}_{\mathrm{N}}(f, \mathbf{A}, \mathbf{x}, c)$).

Proof. The parts (*i*) and (*ii*) follow the above argument. To prove the part (*iii*), note that the proofs of Theorem 5.1.7 in [Neumaier 1990, p. 176] and related theorems for the case $m = n$ remain valid for the case $m \neq n$ if we interpret the notation A^{-1} as in Definition A.7. ■

Remark 3.22. Neumaier [1986] found that if we replace the *Lipschitz set* $\mathbf{A}$ in the definition of the interval Newton operator (3.43) with a *slope matrix* $\mathbf{A}$ for f at $[c, \mathbf{x}]$, then the parts (*i*) and (*ii*) in Theorem 3.21 still hold, and the uniqueness in the part (*iii*) reduces to the *existence* of a solution. See [Neumaier 1990, Theorem 5.4.2] for more details.

For any $\mathbf{A} \in \mathbb{I}^{m\times n}$, we denote by $\mathcal{M}_{\mathbf{A}} : \mathbb{I}^m \to \mathbb{I}^n$ some *sublinear mapping* such that

$$\forall \mathbf{b} \in \mathbb{I}^m : \square\Sigma(\mathbf{A}, \mathbf{b}) \subseteq \mathcal{M}_{\mathbf{A}}(\mathbf{b})$$

and redefine the Newton operator (3.43) as

$$\mathfrak{O}'_{\mathrm{N}}(f, \mathbf{A}, C, \mathbf{x}, c) \equiv c - \mathcal{M}_{C\mathbf{A}}(Cf(c)). \tag{3.45}$$

The new operator is called the *general interval Newton operator* and the corresponding iteration is called the *general interval Newton iteration*.

Neumaier [1985] found that if $\mathbf{A}$ is a *strongly regular Lipschitz matrix* on $\mathbf{x}$ and $\mathcal{M}_{C\mathbf{A}}$ is regular, then the general interval Newton iteration is *strongly convergent* for all choices of $c^{\langle k\rangle} \in \mathbf{x}^{\langle k\rangle}$. Moreover, for all $k \in \mathbb{N}$, we have

$$c^{\langle k\rangle} \in \mathbf{x}^{\langle k+1\rangle} \Rightarrow \mathbf{x}^{\langle k+1\rangle} = c^{\langle k\rangle} \wedge f(c^{\langle k\rangle}) = 0 \tag{3.46}$$

holds (see Theorem 5.2.8 in [Neumaier 1990, p. 183]). Consequently, the volume relation $\mathrm{vol}(\mathbf{x}^{\langle k+1\rangle}) \le \frac{1}{2}\mathrm{vol}(\mathbf{x}^{\langle k\rangle})$ holds if we choose $c^{\langle k\rangle} = \mathrm{mid}(\mathbf{x}^{\langle k\rangle})$. Neumaier [1990, Theorem 5.2.12] also showed that the following combination of the general interval Newton operator and the Hansen-Sengupta operator:

$$c^{\langle 0\rangle} \in \mathbf{x}^{\langle 0\rangle} \equiv \mathbf{x}, \tag{3.47a}$$

$$c^{\langle k-\frac{1}{2}\rangle} \in \mathbf{x}^{\langle k-\frac{1}{2}\rangle} \equiv \mathbf{x}^{\langle k-1\rangle} \cap \mathfrak{O}'_{\mathrm{N}}(f, \mathbf{A}^{\langle k\rangle}, C^{\langle k\rangle}, \mathbf{x}^{\langle k-1\rangle}, c^{\langle k-1\rangle}) \quad (\text{for } k \in \mathbb{Z}_+), \tag{3.47b}$$

$$c^{\langle k\rangle} \in \mathbf{x}^{\langle k\rangle} \equiv \mathfrak{O}_{\mathrm{HS}}(f, \mathbf{A}^{\langle k\rangle}, C^{\langle k\rangle}, \mathbf{x}^{\langle k-\frac{1}{2}\rangle}, c^{\langle k-\frac{1}{2}\rangle}) \quad (\text{for } k \in \mathbb{Z}_+) \tag{3.47c}$$

also leads to strong convergence if $C\mathbf{A}$ is an H-matrix.

3.1.2. Other Interval Methods for Linear Systems

In Section 3.1.2.1, we recall a fundamental method for enclosing the solution set of a *linear interval equation* of the form (3.10), that is, of the form:

$$Ax = b, \tag{3.48}$$

where $A \in \mathbf{A} \in \mathbb{I}^{n\times n}$, $b \in \mathbf{b} \in \mathbb{I}^n$, and $x \in \mathbf{x} \in \mathbb{I}^n$. Note that any linear system can be made square by adding zero coefficients and/or redundant variables. In Section 3.1.2.2, we brief an interesting method for computing the hull of the solution of (3.48), under nice assumptions.

In Section 3.1.2.3, we discuss enclosing the solution set of the *linear interval inequality*

$$Ax \le b, \tag{3.49}$$

where $A \in \mathbf{A} \in \mathbb{I}^{m\times n}$, $b \in \mathbf{b} \in \mathbb{I}^m$, and $x \in \mathbf{x} \in \mathbb{I}^n$. Such a system may be directly modeled in real world applications or generated by linear relaxation methods (see Section 3.3.1). Throughout this section, we use the notations as depicted in Notation 3.5.

3.1.2.1. Interval Gauss Elimination

We consider the linear interval equation (3.48), for simplicity. There are several variants of the *Gauss elimination* method [Wilkinson 1965]. Any of them can be adapted to interval arithmetic to compute tight enclosures of the solution set by simply replacing each real arithmetic step by the corresponding interval arithmetic step. A standard version was described in [Neumaier 1990, Section 4.5]. Similarly to the standard Gauss elimination, the interval version factorizes the coefficient matrix into the product of a *lower triangular matrix* and an *upper triangular matrix*, $\mathbf{L} = (\mathbf{L}_{ij})_{n\times n}$ and $\mathbf{U} = (\mathbf{U}_{ij})_{n\times n}$ respectively, with the following elimination:

$$\mathbf{L}_{ij} \equiv (\mathbf{A}_{ij} - \sum_{k>j} \mathbf{L}_{ik}\mathbf{U}_{kj})/\mathbf{U}_{jj} \quad \text{for } i > j, \tag{3.50a}$$

$$\mathbf{U}_{ij} \equiv \mathbf{A}_{ij} - \sum_{k<i} \mathbf{L}_{ik}\mathbf{U}_{kj} \quad \text{for } i \le j, \tag{3.50b}$$

$$\mathbf{y}_i \equiv \mathbf{b}_i - \sum_{k<i} \mathbf{L}_{ik}\mathbf{y}_k \quad \text{for } i = 1, \ldots, n, \tag{3.50c}$$

$$\mathbf{x}_i \equiv (\mathbf{y}_i - \sum_{j>i} \mathbf{U}_{ij}\mathbf{x}_j)/\mathbf{U}_{ii} \quad \text{for } i = n, \ldots, 1. \tag{3.50d}$$

Thus, we have

$$\mathbf{A} \subseteq \mathbf{LU}, \quad \mathbf{Ly} \supseteq \mathbf{b}, \quad \mathbf{Ux} \supseteq \mathbf{y}. \tag{3.51}$$

The elimination procedure (3.50) leads to the definition of *Gauss inverse* as follows.

Definition 3.23 (Gauss Inverse). Let $\mathbf{A} = (\mathbf{A}_{ij})_{n\times n} \in \mathbb{I}^{n\times n}$. The *Gauss inverse* of $\mathbf{A}$ is the unique mapping, denoted by $\mathbf{A}^{\mathrm{G}} : \mathbb{I}^n \to \mathbb{I}^n$, that maps each $\mathbf{b} \in \mathbb{I}^n$ to the unique solution $\mathbf{x}$, denoted by $\mathbf{A}^{\mathrm{G}}\mathbf{b}$, computed by (3.50).

Suppose the elimination procedure (3.50) does not fail because of a division by an interval containing zero (otherwise, it will produce $[-\infty, \infty]$). If no diagonal element of $\mathbf{U}$ contains zero, then $\mathbf{A}$ is regular. For example, when $\mathbf{A}$ is an H-matrix, then it is possible to factorize $\mathbf{A}$ as shown above (see Theorem 4.5.7 in [Neumaier 1990, p. 158]). In 1974, W. Barth, E. Nuding and H. Beeck showed that if $\mathbf{A}$ is an M-matrix, then

$$\square\Sigma(\mathbf{A}, \mathbf{b}) \subseteq \mathbf{A}^{\mathrm{G}}\mathbf{b} \subseteq \mathbf{A}^{-1}\mathbf{b},$$

with equality if $\mathbf{b} < 0$, $\mathbf{b} > 0$, or $\mathbf{b} \ni 0$ (see Theorem 4.5.8 in [Neumaier 1990, p. 159]). Krawczyk and Neumaier [1987] also proved that $\mathbf{A}^{\mathrm{G}}\mathbf{b} \subseteq \mathbf{A}^{\mathrm{F}}\mathbf{b}$.

Unfortunately, simply replacing a *real* version of Gauss elimination by an *interval* version generally does not result in a good algorithm. Bounds of intermediate quantities tend to grow rapidly because of accumulated rounding errors and especially because of the dependence among computed intervals (see [Hansen and Walster 2004, Section 5.4]). This overestimation has been empirically shown by several researchers in 1970s. This leads to the need for preconditioning. Let $C \in \mathbb{R}^{n\times n}$ be a matrix, we consider the preconditioned problem

$$CAx = Cb \tag{3.52}$$

under the same assumption as in the original problem.

Neumaier [1990, Theorem 4.5.12] proved that if $C\mathbf{A}$ is strictly diagonally dominant; that is,

$$\sum_{j\neq i} |C\mathbf{A}|_{ij}/\langle C\mathbf{A}\rangle_{ii} \leq \beta < 1 \quad \forall i \in \{1,\ldots,n\}, \tag{3.53}$$

then

$$\|\mathrm{rad}(\square\Sigma(\mathbf{A},\mathbf{b}))\|_\infty \leq \|\mathrm{rad}((C\mathbf{A})^{\mathrm{G}}(C\mathbf{b})\|_\infty \leq \frac{1+\beta}{1-\beta}\|\mathrm{rad}(\square\Sigma(\mathbf{A},\mathbf{b}))\|_\infty. \tag{3.54}$$

The preconditioning matrix C can be chosen as $\mathrm{mid}(\mathbf{A})$. While preconditioned Gauss elimination gives good results when $\mathrm{rad}(\mathbf{A})$ is small, preconditioning matrices with wide intervals may lead to a singular $\mathrm{mid}(\mathbf{A})^{-1}\mathbf{A}$, even when $\mathbf{A}^{\mathrm{G}}$ exists. The issue of efficiently characterizing a class of matrices $\mathbf{A} \in \mathbb{I}^{n\times n}$ when $\mathbf{A}^{\mathrm{G}}$ exists seems to be difficult (see [Neumaier 1990, p. 163]). Miller [1972, 1973] proved that the preconditioned Gauss elimination enjoy the *quadratic approximation property*.

The interval Gauss elimination can be combined with fixed point methods by splitting $\mathbf{A} = \mathbf{B} + \mathbf{E}$ and defining the iteration

$$\mathbf{x}^{\langle k\rangle} \equiv \mathbf{B}^{\mathrm{G}}(\mathbf{b} - \mathbf{E}\mathbf{x}^{\langle k-1\rangle}). \tag{3.55}$$

For more details, see [Neumaier 1987a], [Schwandt 1987], and [Mayer 1985, 1987, 1988]; see also [Neumaier 1990, Section 4.4].

3.1.2.2. Hull Method

We use a matrix $C \in \mathbb{R}^{n\times n}$ to precondition the linear interval equation (3.48). A procedure for computing the hull $\square\Sigma(\mathbf{A},\mathbf{b})$ of (3.48) was proposed independently by Hansen [1992] and Bliek [1992, Section 4.4]. Therein, an explicit formula for computing the hull has been devised. However, the proofs were not completely rigorous. One year later, Rohn [1993] provided a correct proof of the explicit formula. He also simplified the formula to improve the amount of computations. Later on, Ning and Kearfott [1997] generalized Rohn's result and proposed an improved method for bounding the hull when $\mathbf{A}$ is an H-matrix. They showed that the result is optimal when the midpoint matrix is diagonal. Recently, Neumaier [1999, 2000] gave a simple proof of the method and showed how the rigor can be obtained in finite precision arithmetic. The version described briefly in this section is a further improved version from [Hansen 2000] and [Hansen and Walster 2004, Section 5.8].

Let $C = \mathrm{mid}(\mathbf{A})^{-1}$. We can write $\mathbf{A} = \mathrm{mid}(\mathbf{A}) + Q[-1,1]$, where $Q \in \mathbb{R}^{n\times n}$. Therefore, the preconditioned matrix can be written as

$$C\mathbf{A} = I + CQ[-1,1]. \tag{3.56}$$

This means that the center of $C\mathbf{A}$ is the identity matrix. Denote $C\mathbf{A} = \mathbf{M} = [\underline{M}, \overline{M}]$ and $C\mathbf{b} = \mathbf{d} = [\underline{d}, \overline{d}]$. Then, for all $i, j = 1,\ldots,n$, we have

$$\underline{M}_{ij} = -\overline{M}_{ij} \quad (\text{for } i \neq j); \tag{3.57}$$

$$\underline{M}_{ii} + \overline{M}_{ii} = 2. \tag{3.58}$$

We now consider the preconditioned system. Suppose we multiply the i-th equation of the system by -1 and change the sign of x_i simultaneously. It follows from (3.57) that the off-diagonal components are unchanged. The diagonal components change sign twice so they

have no net change. Thus, the coefficient matrix is unchanged while x_i and $\mathbf{d}_i$ change sign. Therefore, we can assume that the lower bound $\underline{d}_i \geq 0$ by changing the sign of $\mathbf{d}_i$ (and x_i). By a similar argument, we can assume that

$$0 \leq |\underline{d}_i| \leq \overline{d}_i \quad (\text{for } i = 1, \ldots, n). \tag{3.59}$$

Now, assume that $\mathbf{M}$ is regular. Let $\mathbf{P} = \mathbf{M}^{-1} \in \mathbb{I}^{n\times n}$. We define

$$\mathbf{u}_i = 1/(2\mathbf{P}_{ii} - 1), \quad \mathbf{z}_i = (\underline{d}_i + \overline{d}_i)\mathbf{P}_{ii} - e_i^{\mathrm{T}}\mathbf{P}\overline{d}, \tag{3.60}$$

where e_i is the i-th column of the identity matrix. The hull of the solution set $\mathbf{h} = (\mathbf{h}_1, \ldots, \mathbf{h}_n) \in \mathbb{I}^n$ is then computed as follows:

$$\mathbf{h}_i \equiv \begin{cases} \mathbf{u}_i\mathbf{z}_i & \text{if } \mathbf{z}_i > 0, \\ \mathbf{z}_i & \text{if } \mathbf{z}_i \leq 0. \end{cases} \quad (\text{for } i = 1, \ldots, n) \tag{3.61}$$

The proof of this method can be found in [Hansen 2000]. To obtain this result, we assume the center of computed $\mathbf{A}$ is exactly the identity matrix. In practice, it is not because of rounding errors, for example, caused by the computations of C and CQ. The reader can find in [Hansen 2000] and [Hansen and Walster 2004, Section 5.8] a rigorous computation procedure that takes into account the rounding errors.

A combination of the interval Gauss-Seidel method (see Section 3.1.1.2) and the hull method was also proposed in [Hansen and Walster 2004, Section 5.8]. Basically, the combination method uses the interval Gauss-Seidel method with a limited number of iteration steps for the first p equations ($1 \leq p < n$). Where the other variables $x_{p+1}, \ldots, x_n$ is replaced by its interval bounds (domains). The resulting system is now a system of p unknowns in which the midpoint matrix C is the identity. We then apply the hull method to the resulting system. This may fail if the new coefficient matrix is not regular. In this case, we can use the Gauss-Seidel method. If the new coefficient matrix is regular, the hull method obtains a sharp enclosure for first p components of x. Once the new bounds on $x_1, \ldots, x_p$ have been obtained, we can obtain the new bounds on the remaining component of $\mathbf{x}$ by using the Gauss-Seidel method.

3.1.2.3. Linear Interval Inequalities

We now consider the system of inequalities (3.49). We introduce a vector of m nonnegative slack variables $y = (y_1, \ldots, y_m)^{\mathrm{T}} \geq 0$ to convert the linear interval inequality (3.49) into a linear interval equation

$$Ax + y = b, \tag{3.62}$$

or into the form

$$A'z = b, \tag{3.63}$$

where

$$A' = (A \;\; I) \in \mathbb{R}^{m\times(n+m)}, \quad z = (x_1, \ldots, x_n, y_1, \ldots, y_m)^{\mathrm{T}} \in \mathbb{R}^{n+m}. \tag{3.64}$$

Now we can use methods for linear interval equations to enclose the solution set of (3.63). For example, we can use the Krawczyk iteration for linear interval equations (see Section 3.1.1.1) or the interval Gauss-Seidel iteration with Remark 3.9 (see Section 3.1.1.2). In the application of interval Gauss-Seidel iteration in Section 3.1.1.2, we set $m^* = m$. The m^* variables for each iteration step are chosen from $\{x_1, \ldots, x_n\}$ if $m \leq n$; or consists of $x_1, \ldots, x_n$ and arbitrary $(m - n)$ variables from $\{y_1, \ldots, y_m\}$ otherwise.

The reader can find a rule for ordering the component equations and choosing the variables in Chapter 6 of [Hansen and Walster 2004].

3.1.3. Exclusion Tests

The basic *branch-and-prune* methods that use interval forms to find all solutions of a system of nonlinear equations frequently have the difficulty that sub-boxes containing no solution cannot be easily eliminated if there is a nearby solution outside the box. This has the *cluster effect* that many small boxes near each solution are created by splitting/branching. More broadly, the cluster effect may occur when solving optimization problems or constraint satisfaction problems in the *branch-and-bound* framework or the *branch-and-prune* framework, respectively (see [Kearfott and Du 1992, 1994] and [Van Hentenryck *et al.* 1997b]). In particular, the *branch-and-prune* methods that use interval forms with the *linear approximation property* to solve constraint satisfaction problems often produce an exponentially growing number of tiny boxes near solutions when the resolution/precision ε approaches to 0. The interval forms with the *quadratic approximation property* are often sufficient to bound the number of boxes for sufficiently small ε. See [Schichl and Neumaier 2004a, Section 3] for more details on the cluster effect in constraint satisfaction problems; see also [Neumaier 2004, Section 15] for details on the cluster effect in optimization problems.

In order to reduce the cluster effect, one might uses a test to exclude as many and as large regions that contain no solution as possible. One of the most common tools used in finding all *zero points* (also called solutions and roots in literature) of a system of nonlinear equations of the form (2.20) is the *exclusion test*. The concept of an exclusion test can be traced back at least to [Moore 1979, p. 76], though it may be used earlier in other forms. For simplicity, we only give the definition of the tests that take a box as input. The concept of an exclusion test (on boxes) can be defined formally as follows.

Definition 3.24 (Exclusion Test). Given a function $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$. An *exclusion test* for f is a Boolean function $t_f : \mathbb{I}^n \to \{0, 1\}$ such that, for any $\mathbf{x} \in \mathbb{I}^n$,

$$t_f(\mathbf{x}) = 0 \Rightarrow \forall x \in \mathbf{x} \cap D : f(x) \neq 0. \tag{3.65}$$

In the output of an exclusion test, the value 0 stands for `false` and the value 1 for `true`. By definition, $t_f(\mathbf{x}) = 0$ implies that f has no zero point in $\mathbf{x}$. Therefore, $t_f(\mathbf{x}) = 1$ is a *necessary condition* for f to have a zero point in $\mathbf{x}$.

Given a system of the form (2.20), a recursive version of the **Exclusion** algorithm is presented in Algorithm 3.1 (see also [Georg 2001, 2003] for non-recursive versions). This algorithm is a generic bisection search based only on the concept of an exclusion test (see Section 3.1.6 for a combination of an exclusion test and other tests). It gets as input a box playing the role of variable domains and provides as output a collection, S^+, of *ε-bounded boxes*[2] that cannot be discarded by using the chosen exclusion test. Each box of the output is a tiny box that potentially contains one or more solutions. If the output is empty, it implies that the problem has no solution. The accuracy of the output depends on the quality of the underlying exclusion test. The sooner the exclusion test can discard (boxes), the faster and more accurate the algorithm can be. Note that the exclusion test is not necessary fixed for every iteration. That is, the exclusion test used as input of each recursive call in Algorithm 3.1 may be different. A combination of an exclusion test and other tests is discussed in Section 3.1.6.

[2] An *ε-bounded box* is a box of which the sides are less than or equal to ε.

Algorithm 3.1: The **Exclusion** algorithm

```
Input: x, t_f, ε, S+.
Output: S+.
if t_f(x) = false then return ;
if w(x) < ε then
|  S+ := S+ ∪ {x};
|  return ;
end
(x1, x2) := Bisect(x);
Exclusion(x1, t_f, ε, S+);
Exclusion(x2, t_f, ε, S+);
```

During last forty years, numerous exclusion tests have been developed by using interval arithmetic. Simple exclusion-based solution methods in interval analysis often aim at constructing an interval form $[f]$ of f and defining the exclusion test as

$$t_f(\mathbf{x}) = 1 \Leftrightarrow 0 \in [f](\mathbf{x}). \tag{3.66}$$

So far, different interval methods are mainly distinguished in the way of constructing the function $[f]$, although some other improvements/features have also been introduced.

The Krawczyk operator (Section 3.1.1.1), interval Gauss-Seidel method (Section 3.1.1.2), and the interval Gauss elimination (Section 3.1.2.1) can be used to construct exclusion tests for linear systems.

3.1.3.1. Lipschitz Functions

One of the simplest exclusion tests for *Lipschitz* functions was devised from the well-known *Lipschitz condition*. Let L be a *Lipschitz constant* for a *Lipschitz continuous function* f on $\mathbf{x} \in \mathbb{I}^n$; that is,

$$\forall y, z \in \mathbf{x} : \|f(y) - f(z)\|_\infty \leq L\|y - z\|_\infty. \tag{3.67}$$

Thus, if $\mathbf{x}$ contains a zero of f, then (see [Xu *et al.* 1996, p. 185])

$$\|f(\text{mid}(\mathbf{x}))\|_\infty \leq L\|\text{rad}(\mathbf{x})\|_\infty. \tag{3.68}$$

The condition (3.68) is hence a necessary condition for the system (2.20) to have a solution in $\mathbf{x}$. This condition is equivalent to an exclusion test, called the *Lipschitz exclusion test*. Indeed, we can define it as

$$t_f(\mathbf{x}) = \texttt{true} \Leftrightarrow \|f(\text{mid}(\mathbf{x}))\|_\infty \leq L\|\text{rad}(\mathbf{x})\|_\infty. \tag{3.69}$$

If $f = g - h$ is the difference of two increasing functions on $\mathbf{x} = [\underline{x}, \overline{x}]$, a simple test was also proposed in [Xu *et al.* 1996] as a necessary condition for having a solution in $\mathbf{x}$:

$$g(\underline{x}) \leq h(\overline{x}) \tag{3.70a}$$
$$h(\underline{x}) \leq g(\overline{x}) \tag{3.70b}$$

Moreover, Xu *et al.* [1996] proved that the splitting, called the *uniform dichotomization*, that consists of n dichotomizations, each of which is to dichotomize a domain, is *asymptotically*

optimal among the *uniform cell subdivisions* (i.e., each side of a box is split into $k_i \geq 2$ equal parts at subdivision level i). They also proved that if f is *twice continuously differentiable* or is the difference of two *monotone and continuously differentiable functions*, then the *uniform dichotomous search* using the Lipschitz exclusion test can discard almost boxes except a constantly bounded number of boxes when $\varepsilon \to 0$. That leads to the complexity $\mathcal{O}(\log \frac{1}{\varepsilon})$ for the first case. This is far better than the complexity $\mathcal{O}(\varepsilon^{-N})$ obtained in the general case, where N is the maximum level of subdivision.

More sophisticated methods employ advanced computations for exclusion tests. For instance, the parts (*ii*) of Theorem 3.14, Theorem 3.18, Theorem 3.21, Remark 3.15, Remark 3.19, and Remark 3.22 show that it is possible to construct exclusion tests for Lipschitz functions by using the Krawczyk, Hansen-Sengupta, and interval Newton operators with either Lipschitz matrices or *slope matrices*.

3.1.3.2. Taylor Expansion

We now consider the functions $f, g : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$. For each *multi-index* $\mathbf{i} = (i_1, \dots, i_n)^{\mathrm{T}} \in \mathbb{N}^n$ and a real vector $x = (x_1, \dots, x_n)^{\mathrm{T}} \in \mathbb{R}^n$, the power term of order $\mathbf{i}$ is defined as

$$x^{\mathbf{i}} \equiv \prod_{j=1}^{n} x_j^{i_j}. \tag{3.71}$$

Xu *et al.* [1997] proposed an exclusion test based on *power series expansion*. In particular, for two power series' $f(x) = \sum_{\mathbf{i}} f_{\mathbf{i}} x^{\mathbf{i}}$ and $g(x) = \sum_{\mathbf{i}} g_{\mathbf{i}} x^{\mathbf{i}}$, they define $f \prec\prec g$ if $|f_{\mathbf{i}}| \leq g_{\mathbf{i}}$ for every $\mathbf{i} \in \mathbb{N}^n$. If $f \prec\prec g$ and the series g converges on a box $\mathbf{x} \in \mathbb{I}^n$ contained in D, then

$$|f(c)| \leq g(|c| + r) - g(|r|), \tag{3.72}$$

is equivalent to an exclusion test, called the *power series exclusion test*, for f on $\mathbf{x}$, where $c = \mathrm{mid}(\mathbf{x})$ and $r = \mathrm{rad}(\mathbf{x})$. Xu *et al.* [1997] also proved that the number of boxes that cannot be discarded by this exclusion test is bounded by a constant, provided that f is sufficiently smooth together with other weak assumptions (see [Xu *et al.* 1997]).

Very recently, K. Georg and his collaborators [Allgower *et al.* 2002; Georg 2001, 2003] have extended the above method of Xu *et al.* [1997] by using the *Taylor expansion* of order $k \in \mathbb{N}$ with integral remainder:

$$f(c + x) = f(c) + \sum_{\mathbf{i} \in \mathbb{N}^n,\ 0 < \|\mathbf{i}\|_1 < k} \partial^{\mathbf{i}} f(c) x^{\mathbf{i}} + \sum_{\mathbf{i} \in \mathbb{N}^n,\ \|\mathbf{i}\|_1 = k} x^{\mathbf{i}} \int_0^1 \partial^{\mathbf{i}} f(c + xt)\ k(1 - t)^{k-1} dt, \tag{3.73}$$

where $\partial^{\mathbf{i}}$ is the *partial derivative* of order $\mathbf{i} = (i_1, \dots, i_n) \in \mathbb{N}^n$, which is defined as

$$\partial^{\mathbf{i}} \equiv \Big(\prod_{j=1}^{n} i_j!\Big)^{-1} \prod_{j=1}^{n} \partial_j^{i_j}. \tag{3.74}$$

Instead of using the relation $\prec\prec$ defined by Xu *et al.* [1997], K. Georg and his collaborators used a *domination relation* $\prec_k$ of order $k \in \mathbb{Z}_+$. The relation $f \prec_k g$ holds on D if and only if $\partial^{\mathbf{i}} f$ and $\partial^{\mathbf{i}} g$ are integrable on D and, for every $x, y \in D : |x| \leq |y|$ and every $\mathbf{i} \in \mathbb{N}^n : \|\mathbf{i}\|_1 \leq k$, we have

$$|\partial^{\mathbf{i}} f(x)| \leq \partial^{\mathbf{i}} g(|x|) \leq \partial^{\mathbf{i}} g(|y|). \tag{3.75}$$

The relation $\prec_k$ implies the relation $\prec_l$ if $k > l$. The relation $\prec\prec$ is equivalent to the relation $\prec_\infty$, where the $\prec_\infty$ holds if and only if the relation $\prec_k$ holds for every $k \in \mathbb{Z}_+$.

Let $f, g : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$ be two functions such that $f \prec_k g$ holds on D for some $k \in \mathbb{Z}_+$ and $\mathbf{x} \in \mathbb{I}^n$ a box in D. In [Georg 2003], the *Taylor exclusion test* of order k for f on $\mathbf{x}$ is defined as

$$|f(c)| \leq g(|c + r|) - g(|c|) - \sum_{\mathbf{i} \in \mathbb{N}^n,\ 0 < \|\mathbf{i}\|_1 < k} (\partial^{\mathbf{i}} g(|c|) - \partial^{\mathbf{i}} f(c)) r^{\mathbf{i}}, \tag{3.76}$$

where $c = \operatorname{mid}(\mathbf{x})$ and $r = \operatorname{rad}(\mathbf{x})$.

Note that, for $k = 1$, the Taylor exclusion test (3.76) reduces to the power series exclusion test in (3.72), under the assumption $f \prec_1 g$ instead of $f \prec\prec g$. Note also that, for $k = 1$, if there are *Lipschitz constants* $C_{\mathbf{i}}$ such that

$$C_{\mathbf{i}} \geq \sup_{x \in \mathbf{x}} |\partial^{\mathbf{i}} f(x)|, \tag{3.77}$$

then the condition

$$|f(c)| \leq \sum_{\|\mathbf{i}\|_1 = 1} C_{\mathbf{i}} r^{\mathbf{i}} \tag{3.78}$$

is an exclusion test for f on $\mathbf{x}$, which is very similar to the Lipschitz exclusion test defined in (3.68). Similarly, the condition

$$|f(c)| \leq \sum_{\|\mathbf{i}\|_1 = 1} |\partial^{\mathbf{i}} f(c)| r^{\mathbf{i}} + \sum_{\|\mathbf{i}\|_1 = 2} C_{\mathbf{i}} r^{\mathbf{i}} \tag{3.79}$$

is also an exclusion test for f on $\mathbf{x}$. The reader can find more details in [Georg 2001].

The complexity of the Taylor exclusion test is summarized in Theorem 3.25. The reader can find in Section 3.3.2.2 another method that uses linear programming on the *domination* of nonlinear terms in the Taylor expansion to devise exclusion tests.

Theorem 3.25 (Georg). Let $f, g : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$ be two functions such that the relation $f \prec_k g$ holds on D for some $k \in \mathbb{Z}_+$. If, for each zero z of f, there exists $l \leq k$ in $\mathbb{Z}_+$ such that $\partial^{\mathbf{i}} f(z) = 0$ holds for all $\|\mathbf{i}\|_1 < l$ and that

$$\exists \alpha > 0 : \alpha \|x - z\|_\infty^l \leq \|f(x)\|_\infty \text{ whenever } \|x - z\|_\infty \leq \alpha; \tag{3.80}$$

then the number of boxes that cannot be discarded in the **Exclusion** algorithm using the Taylor exclusion test is bounded by a constant as $\varepsilon \to 0$.

3.1.4. Exclusion Regions

Quite early, Kantorovich's theorem (Theorem 3.26) has been used to devise exclusion tests (and existence/uniqueness tests as well). However, the computed regions that do not contain any zero are not in the form of boxes or *convex sets*, but in the form of the difference of two boxes. For completeness, we recall Kantorovich's theorem here.

Theorem 3.26 (Kantorovich, 1948). Let $f : D \subseteq \mathbb{R}^n \to R^n$ be a twice continuously differentiable and $c \in \mathbb{R}^n$ a vector in a box $\mathbf{x} \subseteq D$ such that $f'(x)$ is invertible. Suppose there exist three positive constants α, β, γ such that $\Delta = 1 - 2\alpha\beta\gamma > 0$ and

$$\|f'(c)^{-1}\|_\infty \leq \alpha, \quad \|f'(c)^{-1}f(c)\|_\infty \leq \beta, \tag{3.81a}$$

$$\forall x \in \mathbf{x},\ 1 \leq i \leq n : \sum_{1\leq j,k\leq n} \left|\partial^2_{jk} f_i(x)\right| \leq \gamma. \tag{3.81b}$$

Let denote $\underline{r} \equiv 2\beta/(1+\sqrt{\Delta})$ and $\overline{r} \equiv (1+\sqrt{\Delta})/(\alpha\gamma)$. Then

(*i*) There is *no* zero x^* of f in $\mathbf{x}$ satisfying $\underline{r} \leq \|x^* - c\|_\infty \leq \overline{r}$;

(*ii*) There is *at most* one zero x^* of f in $\mathbf{x}$ satisfying $\|x^* - c\|_\infty \leq 2/(\alpha\gamma)$;

(*iii*) If $\|x - c\|_\infty < \overline{r}$ holds for all $x \in \mathbf{x}$, then there is a *unique* zero x^* of f in $\mathbf{x}$ and this zero satisfies $\|x^* - c\|_\infty \leq \underline{r}$.

Inspired by Kantorovich's theorem, Schichl and Neumaier [2004a] devised several ways to reduce the cluster effect by identifying an inclusion region R^{I} in which there exists a solution, and an *exclusion region* $R^{\mathrm{E}} \supset R^{\mathrm{I}}$ of which the interior contains only the solutions in R^{I}. Hence, the region $\mathrm{int}(R^{\mathrm{E}}) \setminus R^{\mathrm{I}}$ can be safely discarded. The new methods use *slope matrices* instead of the second order derivative. They significantly enlarge the size of exclusion regions.

The first method is based on the following theorem (cf. [Schichl and Neumaier 2004a, Theorem 4.3]). In this theorem, the inclusion and exclusion regions are boxes if S is a box.

Theorem 3.27 (Schichl & Neumaier). Let $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$ be a twice continuously differentiable function on a *convex set* D, X a *convex* subset of D, $c \in X$, $C \in \mathbb{R}^{n\times n}$, and $S \subseteq X$ any set containing c. Suppose we have the following componentwise bounds:

$$|Cf(c)| \leq \bar{b} \in \mathbb{R}^n, \tag{3.82a}$$

$$|Cf'(c) - I| \leq B' \in \mathbb{R}^{n\times n}, \tag{3.82b}$$

$$|Cf[c,c,x]| \leq B \in \mathbb{R}^{n\times n\times n} \quad \forall x \in S. \tag{3.82c}$$

For $0 < v \in \mathbb{R}^n$, we define $w \equiv (I - B')v \in \mathbb{R}^n$ and $a \equiv Bvv \in \mathbb{R}^n$. Suppose $D_i \equiv w_i^2 - 4a_ib_i > 0$ for all $i = 1, \ldots, n$. We also define

$$\lambda^{\mathrm{E}} \equiv \min_{1\leq i\leq n} \left\{\frac{w_i + \sqrt{D_i}}{2a_i}\right\}, \quad \lambda^{\mathrm{I}} \equiv \max_{1\leq i\leq n} \left\{\frac{2\bar{b}_i}{w_i + \sqrt{D_i}}\right\}. \tag{3.83}$$

If $\lambda^{\mathrm{I}} < \lambda^{\mathrm{E}}$, then there is at least one zero of f in the inclusion region $R^{\mathrm{I}} \equiv S \cap [c - \lambda^{\mathrm{I}}v,\ c + \lambda^{\mathrm{I}}v]$. The zeros in R^{I} are the only zeros of f in the interior of the exclusion region $R^{\mathrm{E}} \equiv S \cap [c - \lambda^{\mathrm{E}}v,\ c + \lambda^{\mathrm{E}}v]$.

In practice, the preconditioning matrix C in Theorem 3.27 can be chosen as an approximation of $f'(z)^{-1}$, where z is an approximate zero of f. Such an approximate zero can be computed by using any local iterative method, such as Newton-like methods. A simplified procedure was devised for quadratic functions in [Schichl and Neumaier 2004a, Corollary 4.4].

The second method in [Schichl and Neumaier 2004a] is to compute inclusion and exclusion

regions in the form of *polytopes* (cf. [Schichl and Neumaier 2004a, Theorem 5.1])

$$P^{\mathrm{I}} = \{x \in \mathbb{R}^n \mid (w-v)^{\mathrm{T}}|x-c| \le \overline{b}^{\mathrm{T}} w\},$$
$$P^{\mathrm{E}} = \{x \in \mathbb{R}^n \mid P(w)|x-c| + B'^{\mathrm{T}} w \le w\},$$

where $0 \le v \le w \in \mathbb{R}^n$, $B = (B_1, \dots, B_n) \in \mathbb{R}^{n\times n\times n}$, $P(w) \equiv (B_1^{\mathrm{T}} w, \dots, B_n^{\mathrm{T}} w) \in \mathbb{R}^{n\times n}$, and the bounds are defined as in Theorem 3.27. A simplified formula was also given therein.

The above two methods well address regions near zeros of f. Constructing exclusion regions around arbitrary points was also proposed in [Schichl and Neumaier 2004a, Theorem 7.1].

Theorem 3.28 (Schichl & Neumaier). Let $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$ be a twice continuously differentiable function on a *convex set* D, X a *convex* subset of D, $c \in X$, and $C \in \mathbb{R}^{n\times n}$. Suppose we have the following componentwise bounds:

$$|Cf(c)| \ge \underline{b} \in \mathbb{R}^n, \tag{3.84a}$$
$$|Cf'(c)| \le B' \in \mathbb{R}^{n\times n}, \tag{3.84b}$$
$$|Cf[c,c,x]| \le B \in \mathbb{R}^{n\times n\times n} \quad \forall x \in M_u, \tag{3.84c}$$

where $0 < u \in \mathbb{R}^n$ such that $\underline{b} - B'u - Buu > 0$ and $M_u \equiv \{x \mid |x-c| \le u\}$. Then there is no zero of f in M_u.

3.1.5. Existence and Uniqueness Tests

Depending on goals, it is possible to derive various *solution algorithms* based on the branch-and-prune framework. For example, to obtain merely a list of boxes, of which the union contains all solutions and the total measure is less than the measure of the initial region, a computational uniqueness test is not necessary. However, to isolate all solutions or determine the precise number of solutions, such a uniqueness test is crucial. Furthermore, accepting as large a box as possible in which uniqueness can be proven often leads to a faster, more practical algorithm. The concepts of an *existence test* and a *uniqueness test* can be defined formally as follows.

Definition 3.29 (Existence Test). Given a function $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$. An *existence test* for f is a function $t_f : \mathbb{I}^n \to \{0,1\}$ such that, for any $\mathbf{x} \in \mathbb{I}^n$, f has a zero in $\mathbf{x}$ if $t_f(\mathbf{x}) = 1$.

Definition 3.30 (Uniqueness Test). Given a function $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$. A *uniqueness test* for f is a function $t_f : \mathbb{I}^n \to \{0,1\}$ such that, for any $\mathbf{x} \in \mathbb{I}^n$, f has a unique zero in $\mathbf{x}$ if $t_f(\mathbf{x}) = 1$.

In the output of existence and uniqueness tests, the value 0 stands for `false` and the value 1 for `true`. Computational existence and uniqueness tests can be devised based on Lipschitz matrices. One can use the Krawczyk, Hansen-Sengupta, and interval Newton iterations for this purpose. For instance, the parts (*iii*) of Theorem 3.14, Theorem 3.18, and Theorem 3.21 show that it is possible to use, respectively, the Krawczyk, Hansen-Sengupta, and interval

Newton operators with Lipschitz matrices as uniqueness tests. Moreover, Remark 3.15 and Remark 3.19, and Remark 3.22 also show that, respectively, the Krawczyk, Hansen-Sengupta, and interval Newton operators with *slope matrices* can be used as existence tests. Moreover, the Krawczyk operator with slope matrices under the stronger assumption in Remark 3.15 can be used for uniqueness tests.

The Kantorovich theorem (Theorem 3.26) has been used to devise existence/uniqueness tests (see [Ortega and Rheinboldt 1970, 2000, Theorem 12.6.1]). However, Shen and Neumaier [1990] proved that the Krawczyk operator with slope matrices always provide an existence region that is at least as large as that computed by using the Kantorovich theorem. Hansen [1997] improved the Krawczyk's method by improving the preconditioning, but he gave only heuristics for the process. Schichl and Neumaier [2004a] found the following theorem that can serve as an existence test (cf. [Schichl and Neumaier 2004a, Theorem 4.2]).

Theorem 3.31 (Schichl & Neumaier). Let $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$ be a twice continuously differentiable function on a convex set D, X a convex subset of D, $c \in X$, and $C \in \mathbb{R}^{n\times n}$. Suppose we have the following componentwise bounds:

$$|Cf(c)| \leq \bar{b} \in \mathbb{R}^n, \tag{3.85a}$$
$$|Cf'(c) - I| \leq B' \in \mathbb{R}^{n\times n}, \tag{3.85b}$$
$$|Cf[c, c, x]| \leq B \in \mathbb{R}^{n\times n\times n} \quad \forall x \in M_u, \tag{3.85c}$$

where $0 < u \in \mathbb{R}^n$ such that $(B' + Bu)u + \bar{b} \leq u$ and $M_u \equiv \{x \mid |x - c| \leq u\}$. Then there exists a zero of f in M_u.

Schichl and Neumaier [2004a] also proposed the following theorem to verify the uniqueness of a solution found in a box.

Theorem 3.32 (Schichl & Neumaier). Let $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$ be a twice continuously differentiable function on a convex set D, X a convex subset of D, and $\mathbf{x} \in \mathbb{I}^n$ a subset of X. Suppose c is an approximate zero of f in X and $C, B \in \mathbb{R}^{n\times n}$ such that

$$|CF[c, \mathbf{x}] - I| + |CF[\mathbf{x}, c, \mathbf{x}]|\ |\mathbf{x} - c| \leq B. \tag{3.86}$$

If $\|B\| < 1$ for some *monotone norm* $\|.\|$, then $\mathbf{x}$ contains at most one solution of f.

By combining Theorem 3.31 and Theorem 3.32, it is possible to better verify the uniqueness of a solution in a box. A simplified formula for quadratic functions was also given in [Schichl and Neumaier 2004a, Section 6].

3.1.5.1. Epsilon-Inflation

As mentioned in Section 3.1.1, the interval Jacobian matrix $[f'](\mathbf{x})$ is a Lipschitz matrix for f on $\mathbf{x}$ [Neumaier 1990, p. 174–175]; hence, it can be used in the Krawczyk, Hansen-Sengupta, and interval Newton operators. More than ten years ago, one has often preferred interval Jacobian matrices to slope matrices because it is simple to incorporate them in a computational uniqueness test. In general, interval slopes provide tighter bounds and faster convergence, but

only allow existence (and not uniqueness) to be verified when used in a straightforward manner (see Remark 3.15, Remark 3.19, and Remark 3.22). However, ten years ago, Rump [1994] proposed an effective two-stage processing method to first verify the existence in as small a region as possible, then verify the uniqueness in as large a region as possible. In particular, when given an initial guess x_a for an approximate zero of f, Rump's method computes a box $\mathbf{x}_\exists \ni x_a$ in which there must exist a zero of f and then a lager box $\mathbf{x}_{\exists!}$ containing $\mathbf{x}_\exists$ such that f contains a unique zero in $\mathbf{x}_{\exists!}$ (see Algorithm 2.1 in [Rump 1994]). Rump's method uses *slopes* instead of interval Jacobian matrices. Two boxes, $\mathbf{x}_\exists$ and $\mathbf{x}_{\exists!}$, obtained by Rump's method involves the so-called *epsilon-inflation*. The procedure starts by constructing a small box centered at x_a (e.g., by resorting to a bisection method), and then expanding it by using the epsilon-inflation until the existence or uniqueness can be verified. This verification procedure shows much promise in practice, especially for finding just one solution.

The epsilon-inflation was originated in Rump's PhD thesis [Rump 1980] and described in detailed in [Mayer 1995, 1998; Rump 1998]. In particular, for a box $\mathbf{x}$, the *epsilon-inflation* (or *ε-inflation*) is defined as

$$\mathbf{x} \circ \varepsilon \equiv \begin{cases} \mathbf{x} + \mathrm{w}(\mathbf{x})[-\varepsilon, \varepsilon] & \text{if } \mathrm{w}(\mathbf{x}) \neq 0, \\ \mathbf{x} + [-\eta, \eta] & \text{otherwise,} \end{cases} \tag{3.87}$$

where η denotes a tiny positive number, often chosen as the smallest computer-representable positive number. The corresponding iteration is then defined as:

$$\mathbf{y}^{\langle k \rangle} \equiv \mathbf{x}^{\langle k \rangle} \circ \varepsilon, \tag{3.88a}$$

$$\mathbf{x}^{\langle k+1 \rangle} \equiv \mathbf{z} + \mathbf{C}\mathbf{y}^{\langle k \rangle}, \tag{3.88b}$$

where $\mathbf{x}^{\langle k \rangle}$, $\mathbf{y}^{\langle k \rangle}$ and $\mathbf{z}$ are boxes of size n, $\mathbf{C}$ is an interval matrix of size $n \times n$. The convergence of this method is obtained by applying the fixed point theorem of Banach or Brouwer (see Section B.2), under some assumptions [Rump 1992, 1998]:

- If the operations in (3.88) are performed in interval arithmetic (respectively, power set operations) and $\rho(|\mathbf{C}|) < 1/(1 + 2\varepsilon)$ (respectively, $\rho(\mathbf{C}) < 1/(1 + 2\varepsilon)$), then $\mathbf{x}^{\langle k+1 \rangle} \subseteq \mathrm{int}(\mathbf{y}^{\langle k \rangle})$ holds for some k;

- If the operations in (3.88) are performed in interval arithmetic (respectively, power set operations) and $\mathbf{x}^{\langle k+1 \rangle} \subseteq \mathrm{int}(\mathbf{y}^{\langle k \rangle})$ for some k, then $\rho(|\mathbf{C}|) < 1$ (respectively, $\rho(\mathbf{C}) < 1$)).

It is worth mentioning that, among the variants, the epsilon-inflation method has been extended for *P-contractions* by Mayer [1995, 1998] and improved for systems of linear equations by Rohn and Georg [1998].

3.1.6. Inclusion Tests

Kearfott [1987a,b] proposed a test, originally called the *root inclusion test*, for use in generalized bisection algorithms for finding all solutions of a system of nonlinear equations. This test is indeed a combination of an exclusion test (see Section 3.1.3) and a uniqueness test (see Section 3.1.5). Consider a system of equations of the form (2.20), the *root inclusion test* is a function t_f mapping a box $\mathbf{x}$ to one of the three values $\{\mathtt{true}, \mathtt{false}, \mathtt{unknown}\}$ such that:

1. $t_f(\mathbf{x}) = \mathtt{true}$ implies that there is a unique solution of the system (2.20) within $\mathbf{x}$;

2. $t_f(\mathbf{x}) = \texttt{false}$ implies that there is no solution of the system (2.20) within $\mathbf{x}$.

Basically, the root inclusion test is partly based on interval arithmetic to get lower and upper bounds on $f(\mathbf{x})$:

$$\forall x \in \mathbf{x} : l_j \leq f_j(x) \leq u_j \quad (\text{for } i = 1, \ldots, m), \tag{3.89}$$

Thus, if $l_j > 0$ or $u_j < 0$ for some j, we can set $t_f(\mathbf{x}) = \texttt{false}$. To know if $t_f(\mathbf{x}) = \texttt{true}$, similar bounds on the partial derivatives of f_j are computed. Using these bounds, we obtain a new box $\mathbf{x}'$ such that all images of points in $\mathbf{x}$ under a single application of the chord method are contained in $\mathbf{x}'$. The *chord method* is an iteration of the form $x' := x - yf(x)$, where y is held fixed relative to x and is only an approximate inverse to $f'(x)$ [Ortega and Rheinboldt 1970, p. 181]. Thus, if $\mathbf{x}' \subseteq \mathbf{x}$, the Schauder fixed point theorem and the fact that stationary points of the chord method correspond to solutions of (2.20) show that there is a solution of (2.20) in $\mathbf{x}$.

In [Jaulin *et al.* 2001; Jaulin and Walter 1993], the concept of a root inclusion test was extended for solving systems of the form (2.10), where the solution set may contain a continuum. The test obtained is hence called the *inclusion test*. It is an interval form $[t] : \mathbb{I}^n \to \{0, 1, [0, 1]\}$ for the Boolean function $t : \mathbb{R}^n \to \{0, 1\}$ that characterizes the solution set, S, of the considered problem:

$$t(x) = 1 \Leftrightarrow x \in S; \tag{3.90}$$

where the value 0 stands for `false`, the value 1 for `true`, and the interval value $[0, 1]$ for `unknown`. However, the concept of an inclusion test in [Jaulin *et al.* 2001] is no longer compatible with the concept of a root inclusion test in [Kearfott 1987a,b]. Alternatively, we can easily generalize the above two concepts to make them compatible as follows.

Definition 3.33 (Generalized Inclusion Test). Given k properties $p_1, p_2, \ldots, p_k$, each is a function from $\mathbb{I}^n$ to $\{\texttt{false}, \texttt{true}\}$ such that they are mutually exclusive; that is, if $p_i(\mathbf{x}) = \texttt{true}$ for some $i \in \{1, \ldots, k\}$ then $\forall j \neq i : p_j(\mathbf{x}) \neq \texttt{true}$. A *generalized inclusion test* (w.r.t. these k properties) is a function $t_k : \mathbb{I}^n \to \{0, 1, \ldots, k\}$ such that, for any $\mathbf{x} \in \mathbb{I}^n$,

$$\forall i \in \{1, \ldots, k\} \ : \ t_k(\mathbf{x}) = i \Rightarrow p_i(\mathbf{x}) = \texttt{true}. \tag{3.91}$$

The concept of a generalized inclusion test allows computing approximations of the solution set of not only the system (2.20), but also the system (2.10). Indeed, let S be the solution set of the problem of the form (2.10). A generic bisection search technique, called the **Inclusion** algorithm, for finding all solutions of (2.10) based on generalized inclusion tests is presented in Algorithm 3.2. The **Inclusion** algorithm takes as input a box $\mathbf{x} \in \mathbb{I}^n$, among the input parameters, and aims at producing $(k + 1)$ collections of boxes $S_0, S_1, \ldots, S_k$; where each box in S_i satisfies the property p_i. That is, for every $i \in \{1, \ldots, k\}$, we have:

$$\forall \mathbf{x} \in S_i : p_i(\mathbf{x}) = \texttt{true}. \tag{3.92}$$

If there is a property p_j which states that there is no solution in a given box $\mathbf{x}$, then the **Inclusion** algorithm computes an outer approximation of the solution set by returning the union $\bigcup_{i=0,\ i \neq j}^{k} S_i$, with respect to the set of given properties $\{p_1, p_2, \ldots, p_k\}$. The collection S_j should be considered as a trashcan. Adding an object to a trashcan should be interpreted as

Algorithm 3.2: The **Inclusion** algorithm

```
Input: x, t_k, ε; S_0, S_1, ..., S_k.
Output: S_0, S_1, ..., S_k.
if t_k(x) > 0 ∨ (t_k(x) = 0 ∧ w(x) < ε) then
    S_{t_k(x)} := S_{t_k(x)} ∪ {x};
    return ;
end
(x_1, x_2) := Bisect(x);
Inclusion(x_1, t_k, ε, S_0, S_1, ..., S_k);
Inclusion(x_2, t_k, ε, S_0, S_1, ..., S_k);
```

discarding it. Note that the collection S_0 consists of ε-bounded boxes (i.e., the sides of which are bounded by a precision ε) and undetermined by any given properties. In progressively solving, this collection will, therefore, be taken into further processing when reducing ε.

The concept of a generalized inclusion test allows interpreting in a common view the above mentioned tests, including the exclusion tests, uniqueness tests, root inclusion tests, and inclusion tests. Hereafter are the interpretations:

- **Exclusion Test:** $k = 1$, the property $p_1(\mathbf{x})$ states that there is no zero of the function f in $\mathbf{x}$. The collection S_1 is a trashcan, is then ignored. The collection S_0 consists of ε-bounded boxes, each may contain a solution. This type of tests should be applied to systems of equations.

- **Existence Test:** $k = 1$, the property $p_1(\mathbf{x})$ states that there is a unique zero of the function f in $\mathbf{x}$. The collection S_1 consists of boxes, each contains at least one solution. The collection S_0 consists of ε-bounded boxes, each may contain a solution. Note that no boxes are discarded in this case. In practice, one rarely uses existence tests alone. This type of tests should be applied to systems of equations.

- **Uniqueness Test:** $k = 1$, the property $p_1(\mathbf{x})$ states that there is a unique zero of the function f in $\mathbf{x}$. The collection S_1 consists of boxes, each contains a unique solution. The collection S_0 consists of ε-bounded boxes, each may contain a solution. Note that, in this case, no boxes are discarded. In practice, one rarely uses the uniqueness test alone. This type of tests should be applied to systems of equations.

- **Root Inclusion Test:** $k = 2$, the property $p_1(\mathbf{x})$ states that there is a unique zero of the function f in $\mathbf{x}$, the property $p_2(\mathbf{x})$ states that there is no zero of the function f in $\mathbf{x}$. The collection S_1 consists of boxes, each contains a uniques solution. The collection S_2 is a trashcan, is then ignored. The collection S_0 consists of ε-bounded boxes, each may contain a solution. Note that, owning to the property p_2, some boxes may be discarded. This type of tests should be applied to systems of equations.

- **Inclusion Test:** $k = 2$, the property $p_1(\mathbf{x})$ states that all points in $\mathbf{x}$ are solutions of the considered problem, the property $p_2(\mathbf{x})$ states that there is no solution of the considered problem in $\mathbf{x}$. The collection S_1 consists of boxes all the points of which are solutions. The collection S_2 is a trashcan, is then ignored. The collection S_0 consists of ε-bounded boxes, each may contain a solution. Note that, owning to the property p_2, some boxes

may be discarded. This type of tests should be applied to systems of inequalities. For example, we can define an inclusion test for the system (2.10) as follows:

$$t_2(\mathbf{x}) = 1 \quad \Leftrightarrow \quad [f](\mathbf{x}) \subseteq \mathbf{b}, \tag{3.93a}$$

$$t_2(\mathbf{x}) = 2 \quad \Leftrightarrow \quad [f](\mathbf{x}) \cap \mathbf{b} = \emptyset, \tag{3.93b}$$

where $[f]$ is an interval form of f. The collections $S^- = S_1$ and $S^+ = S_1 \cup S_2$ are inner and outer approximations of the solution set S, respectively; that is, $\mathrm{pts}(S^-) \subseteq S \subseteq \mathrm{pts}(S^+)$ (w.r.t. the meaning of point sets).

An algorithm is similar to the **Inclusion** algorithm, which is called **SIVIA** (Set Inverter Via Interval Analysis), was described in [Jaulin *et al.* 2001; Jaulin and Walter 1993] to solve the *set inversion problem*: find the *inverse images* of a function $f : X \subseteq \mathbb{R}^n \to \mathbb{R}^m$ when given $Y \subseteq \mathbb{R}^m$; that is, compute the set $S = X \cap f^{-1}(Y) \equiv X \cap \{x \in \mathbb{R}^n \mid f(x) \in Y\}$.

Lottaz [2000, Section 3.3.3] proposed to use the inclusion test (3.93) in place of the bisection search proposed in [Sam-Haroud 1995, Section 3.4] to determine the feasibility of boxes in the 2^k-*tree representation* [Sam-Haroud 1995]. Namely, in this method, the boxes in a 2^k-*tree* are classified into three categories based on the output of the inclusion test: black (= `false`), white (= `true`), and grey (= `unknown`). The improved classification method, which relies on the inclusion test (3.93), is more rigorous and faster than the original one. After constructing the *octree representation* (i.e., $k = 3$) for *ternary constraints*, which are obtained by ternarizing a factorable NCSP (see Section 2.1.3.5), a constraint propagation algorithm is performed on obtained octrees to achieve *relational* $(3, 2)$-*consistency* with the time complexity $\mathcal{O}(n^5)$ as described in Section 2.3.4.2, where n is the number of variables in tenarized CSPs.

3.2. Constraint Programming Methods

3.2.1. Classical Complete Search Methods

The most simple *backtracking* search is called the *simple backtracking* or *chronological backtracking* (BT) [Golomb and Baumert 1965]. It picks one variable at a time and consider one value for the variable at a time, making sure that the newly picked value is compatible with all the values picked so far (i.e., no inconsistency is detected). If at any stage no value can be assigned to a variable without leading to an inconsistency of the problem, the variable corresponding to the value that was last picked is revised and another value which is not yet tried, if any, is assigned to it. If, from the current node, the algorithm is allowed to go further back to ancestor nodes than just the parent node, it is called *intelligent backtracking* or *dependency directed backtracking* (see [Baar *et al.* 1981]). The backtracking-like methods were designed to address constraint satisfaction problems (CSPs) with discrete domains.

The backtracking-like methods can also change the order of exploring the search space, for example, using the well-known search orders, such as *breadth-first search* (BFS) and *depth-first search* (DFS). Many other search orders for tree search can also be used: *iterative broadening* search (IB) [Ginsberg and Harvey 1992], *limited discrepancy search* (LDS) [Harvey and Ginsberg 1995], *improved limited discrepancy search* (ILDS) [Korf 1996], *depth-bounded discrepancy search* (DDS) [Walsh 1997], and *interleaved depth-first search* (IDFS) [Meseguer 1997].

The reader can find detailed overviews of fundamental complete search methods in constraint satisfaction for CSPs with discrete domains in [Frei 2000, Appendix B] and [Torrens Arnal 2003, Appendix B]. These overviews cover the following methods: *backmarking* (BM)

[Gaschnig 1977], *backjumping* (BJ) [Gaschnig 1978], *forward checking* (FC) [Haralick and Elliot 1980; McGregor 1979], *full* and *partial look-ahead* [Haralick and Elliot 1980], *graph-based backjumping* (GBJ) [Dechter 1990], *partial chronological backtracking* (PBT) and *partial forward checking* (PFC) [Freuder and Wallace 1992], *conflict-directed backjumping* (CBJ) [Prosser 1993], *minimal forward checking* (MFC) [Dent and Mercer 1994], *maintaining arc consistency* (MAC) [Sabin and Freuder 1994].

The overviews in [Frei 2000, Appendix B] and [Torrens Arnal 2003, Appendix B] also cover hybrid algorithms such as *backmarking with backjumping* (BM-BJ), *backmarking with conflict-directed backjumping* (BM-CBJ), *forward checking with backjumping* (FC-BJ), *forward checking with conflict-directed backjumping* (FC-CBJ). These combinations are all due to Prosser [1993]. The overviews also cover some heuristics for accelerating the search, such as *dynamic variable ordering*, and cover *value ordering* heuristics for complete search techniques, such as *dynamic backtracking* (DB) [Ginsberg 1993], *partial order backtracking* (POB) [McAllester 1993], *partial dynamic backtracking* (PDB) [Ginsberg and McAllester 1994], *general partial order backtracking* (GBP), and *flexible partial order backtracking* (FPB) [Bliek 1998].

The most popular complete search for CSPs with continuous domains is the *bisection search*. In the bisection search, each variable domain is bisected into two parts. Each part is considered for further branching after trying with other methods, such as problem reduction methods. The search tree is therefore a binary tree. However, the most simple search for CSPs with continuous domain is the *griding* search or *cell subdivision* search. It splits each variable domain into cells of size ε (the precision), and considers each cell generated by the split, one by one. It can also be viewed as a problem reduction technique.

For more details on complete search methods for CSPs with continuous domains, see Section 3.1.3, Section 3.1.6, and also [Tsang 1993, Chapter 3] and [Apt 2003, Chapter 8]. The reader can also find in [Neumaier 2004] a very recent and comprehensive survey on complete search methods for optimization problems and CSPs with continuous domains.

3.2.2. Branch-and-Prune Methods

In constraint programming, a typical framework for solving constraint satisfaction problems (CSPs) is the *branch-and-prune* framework. It is a variant of the classic *branch-and-bound* framework for solving optimization problems, invented independently by Land and Doig [1960] and Little *et al.* [1963]. Basically, a branch-and-prune algorithm alternates pruning and branching steps. At each pruning step, the algorithm attempts to reduce the considered problem. In other words, it uses a problem reduction technique. At each branching step, the considered problem is split into subproblems. To be complete, a branch-and-prune algorithm has to guarantee that the pruning and branching steps are equivalence-preserving transformations. The branch-and-prune methods are essentially backtracking.

A branch-and-prune algorithm is similar to the **Solution** algorithm in Algorithm 2.1. It might, in practice, have the recursive form as in Algorithm 3.3 or the loop form as in Algorithm 3.4, but is not limited to these forms. In general, it provides as output two sets: the set SOLUTIONS of found solutions and the set ATOMS of atomic subproblems. A problem is considered as atomic if it is not amenable to be split, for instance, when its domains are smaller than an allowed tolerance. In constraint programming, the pruning steps of a branch-and-prune algorithm (i.e., the Prune procedure) usually use constraint propagation techniques. For example, when solving numerical CSPs (NCSPs), they can use interval constraint propagation techniques. The domain reduction techniques are the most typical among the problem

reduction techniques used at the pruning steps. One often aims at constructing domain reduction operators to reduce NCSPs with continuous domains. The contractiveness, correctness, monotonicity, and idempotence are often desired (but not required) for domain reduction techniques (see Property 2.44). Since the previously available branch-and-prune algorithms are different mainly in the domain reduction techniques they use, we only discuss about fundamental domain reduction techniques in the next subsections.

Algorithm 3.3: The **Branch&Prune** algorithm – the recursive version

```
Input: a CSP P.
Output: SOLUTIONS, ATOMS.
if Prune&Check(P, SOLUTIONS, ATOMS) then return ;          ◄ On page 80.
(P_1, ..., P_k) := Split(P);
for i := 1, ..., k do
 | Branch&Prune(P_i, SOLUTIONS, ATOMS);
end
```

Algorithm 3.4: The **Branch&Prune** algorithm – the loop version

```
Input: a CSP P.
Output: SOLUTIONS, ATOMS.
if Prune&Check(P, SOLUTIONS, ATOMS) then return ;          ◄ On page 80.
WAITINGLIST := {P};
while WAITINGLIST ≠ ∅ do
 | Take a CSP P_0 from WAITINGLIST;
 | (P_1, ..., P_k) := Split(P_0);
 | for i := 1, ..., k do
 | | if Prune&Check(P_i, SOLUTIONS, ATOMS) then continue for;   ◄ On page 80.
 | | WAITINGLIST := WAITINGLIST ∪ {P_i};
 | end
end
```

Function Prune&Check(P, SOLUTIONS, ATOMS)

```
Prune(P);
if P is inconsistent then return ;
if P is globally solved then
 | SOLUTIONS := SOLUTIONS ∪ {P}; return true;
end
if P is atomic then
 | ATOMS := ATOMS ∪ {P}; return true;
end
return false;
```

3.2.2.1. Hull Consistency by Search

The concept of *hull consistency* is described in Section 2.3.3.1. In this section we present a simple search algorithm for achieving $\mathbb{F}$-hull consistency for NCSPs. It is presented in Algo-

rithm 3.6. In this algorithm, the search procedure at Line 1 is performed either by a simple increment search or by a bisection search. Other search procedures can also used in place of this search procedure. Since the set of canonical intervals is totally ordered, there always exist the smallest and greatest canonical intervals as required at Line 1. We can use a search procedure similar to the functions **SearchLowerBound** and **SearchUpperBound** in Section 3.2.2.3.

To achieve $\mathbb{F}$-*hull consistency* for an NCSP, a propagation algorithm similar to **AC3** (an algorithm for maintaining arc consistency, see [Mackworth 1977; Tsang 1993]) is performed. The difficult is how to check if a canonical box contains a solution. This can be done by using mathematical techniques in Section 3.1. It does, however, not always give the exact result.

Algorithm 3.6: The $\mathbb{F}$-**Hull Consistency** algorithm – $\mathbb{F}$-hull consistency by search

```
Input: an NCSP P ≡ (V ≡ (x_1, ..., x_n), D, C), a box x ≡ [x_, x‾] ⊆ D.
Output: new domains x' ∈ I^n of V.
x' := x;
WaitingList := C;
while WaitingList ≠ ∅ and x' ≠ ∅ do
    Take a constraint C from WaitingList;
    for each variable x_i constrained by C do
1       search for the outermost canonical interval bounds l ≡ [l_, l‾] and u ≡ [u_, u‾] such
        that l ∩ C[x_i] ≠ ∅ and u ∩ C[x_i] ≠ ∅, respectively;
2       x'_i := [l_, u‾];
        if x'_i has been modified at Line 2 then
            WaitingList := WaitingList ∪ {C' ∈ C | C' ≠ C, x_i is contrained by C'}
        end
    end
end
```

3.2.2.2. Hull Consistency by Propagation

The concept of *hull consistency* is described in Section 2.3.3.1. In this section, we present the **HC4** algorithm, which was proposed by Benhamou *et al.* [1999], with very slight modifications, for simplicity. We consider an NCSP of the form (2.10): $f(x) \in \mathbf{b}$, where $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ is a factorable function and x is a vector of real variables associated with domains $\mathbf{x} \in \mathbb{I}^n$. The other notations of vectors follow the form (2.10) in Section 2.1.3.2.

Tree Representation. The explicit *tree representation* of factorable NCSPs was first introduced by Benhamou *et al.* [1999] for constraint propagation, although the implicit representation equivalent to it has been used in earlier works (e.g., see [McCormick 1976, 1983], [Neumaier 1990, p. 13], and [Benhamou and Older 1992, 1997]). Each factorable constraint $f_j(x) \in \mathbf{b}_j$ is of the form $\varphi(E_1, \ldots, E_k) \in \mathbf{b}$; where φ is a k-ary operation and $E_1, \ldots, E_k$ are factorable subexpressions. It can be recursively represented by an *attribute tree* the root node of which represents the operation φ, and each subexpression E_i is recursively represented in the same way. In the original tree representation of the inequality constraints of the system (2.10), auxiliary variables are introduced to represent the *constraint ranges* (Definition 2.59), as shown in Figure 3.1. For compactness, we combine each node representing an auxiliary variable (and the corresponding root node) with its sibling to get the *compact tree representation* (see Figure 3.2). Therefore, we no longer need the auxiliary variables.

Each node of the tree representation of a constraint is associated with two intervals, called the *forward* and *backward node ranges*. The former is used in forward evaluation and the latter is used in backward propagation. The exact value, hence the exact range, of the subexpression represented by an arbitrary node must be in the intervals associated with that node, that is, in its forward and backward node ranges.

Remark 3.34. Granvilliers and Benhamou [2001] considered the tree representation as a DAG by combining all nodes representing the same subexpressions of individual constraints. However, this does not so much differ from the tree representation because multiple occurrences of the same subexpressions are represented individually for multiple constraints. It is hence less compact than the corresponding *DAG representation* described in Section 6.2.

Example 3.35. Let consider a CSP of the form (2.10):

$$\mathcal{P} \equiv \langle \sqrt{x} + 2\sqrt{xy} + 2\sqrt{y} \leq 7,\ 0 \leq x^2\sqrt{y} - 2xy + 3\sqrt{y} \leq 2;\ x \in [1,16],\ y \in [1,16]\rangle.$$

The first constraint is an inequality with the constraint range $[-\infty, 7]$. The second constraint is a two-sided inequality with the constraint range $[0, 2]$. $\mathcal{P}$ can be written in the form:

$$\begin{cases} \sqrt{x} + 2\sqrt{xy} + 2\sqrt{y} & = & u, \\ x^2\sqrt{y} - 2xy + 3\sqrt{y} & = & v, \end{cases} \tag{3.94}$$

where $x \in [1,16]$ and $y \in [1,16]$ are the initial variables; and $u \in [-\infty, 7]$, $v \in [0,2]$ are the auxiliary variables. The constraint system (3.94) is equivalent to $\mathcal{P}$ w.r.t. the variables x and y. The tree representation of the system (3.94) is depicted in Figure 3.1. The compact tree representation corresponding to the tree in Figure 3.1 is depicted in Figure 3.2. ♣

Forward Evaluation and Backward Propagation on Trees. An algorithm, called **HC4revise**,[3] was proposed by Benhamou *et al.* [1999] to achieve ($\mathbb{F}$-)hull consistency on all nodes, when considered as primitive constraints, of the tree representation of a constraint, not on the constraint itself. It consists of the following two main processes:

1. ***Recursive Forward Evaluation*** – **RFE:** It recursively traverses the compact tree representation of a constraint in the *post-order*, from leaves to the root, and evaluates the forward node range $\mathcal{R}_f[\mathbf{N}]$ of each visited node $\mathbf{N}$ by using the *natural interval form* (see Section A.2.1) of the elementary operation represented by $\mathbf{N}$.

2. ***Recursive Backward Propagation*** – **RBP:** It recursively traverses the compact tree representation of a constraint in the *pre-order*, from the root to leaves, and prunes the backward node range $\mathcal{R}_b[\mathbf{N}]$ of each visited node $\mathbf{N}$ by using the *projection narrowing operator* associated with the father of $\mathbf{N}$.

The **HC4revise** algorithm is presented concisely in Algorithm 3.7. At Line 1 of the **RBP** procedure, the constraint range $\mathcal{R}_b[E]$ is intersected with the node range computed by the **RFE** procedure. At Line 2 of the **RBP** procedure, the elementary operation ψ represented by E defines a relation ψ^* on the sequence $(E, E_1, \ldots, E_k)$; where $E, E_1, \ldots, E_k$ are considered as variables taking values in $\mathcal{R}_b[E], \mathcal{R}_f[E_i], \ldots, \mathcal{R}_f[E_i]$, respectively. At Line 3 of the

[3] It is also referred to as the *forward-backward contractor* in [Jaulin *et al.* 2001, Section 4.2.4].

RBP procedure, the projection of ψ^* on the variable E_i is computed. Since the relation ψ is elementary, the relation ψ^* is very simple. There are some simple formulas to compute the projections (see [Benhamou *et al.* 1999] for more details). The **HC4** algorithm presented in Algorithm 3.10 uses a constraint propagation mechanism similar to the **AC3** algorithm (an algorithm for maintaining arc consistency, see [Mackworth 1977; Tsang 1993]).

The **HC4revise** algorithm is a domain reduction operator applying to a constraint. It enjoys the following properties: contractiveness, correctness, and monotonicity (see Property 2.44).

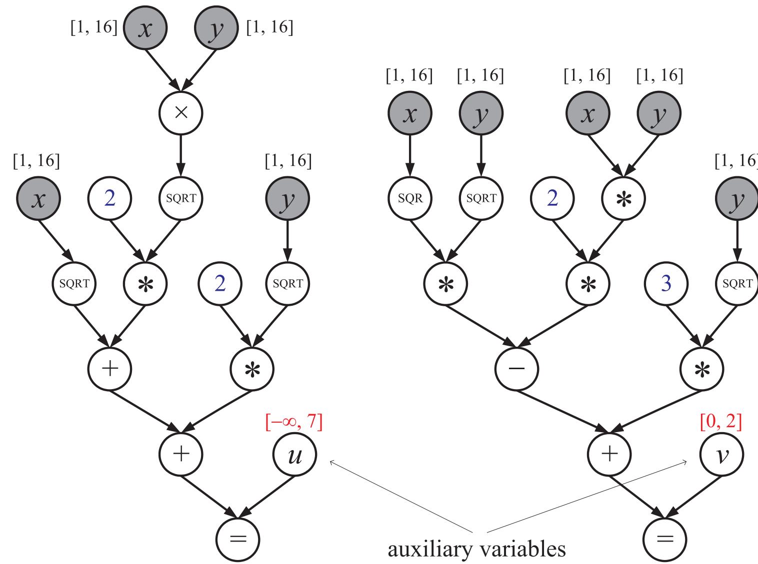

Figure 3.1. The tree representation of the NCSP in Example 3.35. The initial variables, x and y, are represented by the grey nodes that are pointers to the two domains of them. Each tree starting at the root represents a constraint. Each node representing a real constant is associated with the smallest interval that contains the constant. If a node is not specified with a domain in Figure 3.1, its domain is the universal interval $[-\infty, +\infty]$.

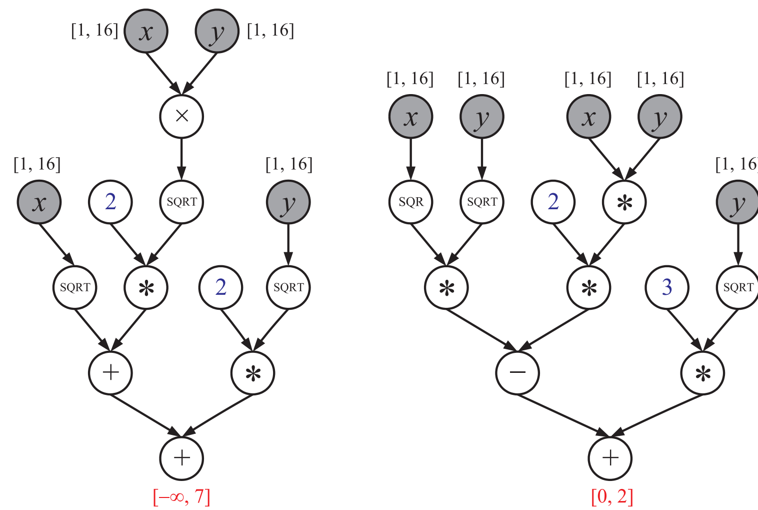

Figure 3.2. The compact tree representation of the NCSP in Example 3.35. No auxiliary variable is needed. The whole tree is presented in a consistent way.

Algorithm 3.7: The **HC4revise** algorithm – a forward-backward propagation on a tree

Input: a compact tree $\mathcal{T}_C$; domains $\mathbf{x} \in \mathbb{I}^n$ of variables $(x_1, \ldots, x_n)$.
Output: new domains $\mathbf{x}' \in \mathbb{I}^n$ of $(x_1, \ldots, x_n)$.
$\mathbf{x}' := \mathbf{x}$;
RFE($\mathcal{T}_C$, $\mathbf{x}'$); ◂ On page 84.
RBP($\mathcal{T}_C$, $\mathbf{x}'$); ◂ On page 84.

Procedure RFE(in/out: a compact tree $\mathcal{T}_E$; **in:** $\mathbf{x} \in \mathbb{I}^n$)

if E is a variable x_i **then** $\mathcal{R}_f[E] := \mathbf{x}_i$;
else if E is the expression $\psi(E_1, \ldots, E_q)$ **then**
  $\mathcal{R}_f[E] := \psi(\mathbf{RFE}(\mathcal{T}_{E_1}, \mathbf{x}), \ldots, \mathbf{RFE}(\mathcal{T}_{E_q}, \mathbf{x}))$;
end

Procedure RBP(in/out: a compact tree $\mathcal{T}_E$, $\mathbf{x} \in \mathbb{I}^n$)

if E is a variable x_i **then** $\mathbf{x}_i := \mathbf{x}_i \cap \mathcal{R}_b[E]$;
else if E is the expression $\psi(E_1, \ldots, E_q)$ **then**
1   $\mathcal{R}_b[E] := \mathcal{R}_b[E] \cap \mathcal{R}_f[E]$;
2   Let ψ^* be the relation $E = \psi(E_1, \ldots, E_q)$ on $\mathbf{E} := (\mathcal{R}_b[E], \mathcal{R}_f[E_i], \ldots, \mathcal{R}_f[E_i])^{\mathrm{T}}$;
   for $i := 1, \ldots, q$ **do**
3     $\mathcal{R}_b[E_i] := \mathbf{E} \cap \psi^*[E_i]$;
     RBP($\mathcal{T}_{E_i}$, $\mathbf{x}$);
   end
end

Algorithm 3.10: The **HC4** algorithm – hull consistency on primitive constraints

Input: an NCSP $\mathcal{P} \equiv (\mathcal{V} \equiv (x_1, \ldots, x_n), \mathcal{D}, \mathcal{C})$, a box $\mathbf{x} \subseteq \mathcal{D}$.
Output: new domains $\mathbf{x}' \in \mathbb{I}^n$ of $\mathcal{V}$.
$\mathbf{x}' := \mathbf{x}$;
WaitingList $:= \mathcal{C}$;
while WaitingList $\neq \emptyset$ **and** $\mathbf{x}' \neq \emptyset$ **do**
   Take a constraint C from WaitingList;
1   $\mathbf{y} :=$ **HC4revise**($\mathcal{T}_C, \mathbf{x}'$); ◂ On page 84.
   if $\mathbf{y} \neq \mathbf{x}'$ **then**
     Put into WaitingList the constraint C and every constraint C' sharing with C at least a variable of which the domain has been reduced at Line 1;
     $\mathbf{x}' := \mathbf{y}$;
   end
end

Let **HC4revise*** be the algorithm obtained from **HC4revise** by removing the intersection at Line 1 of the **RBP** procedure if the considered node is the root. The following theorem [Benhamou *et al.* 1999, Proposition 4] characterizes the correctness of the **HC4revise** algorithm.

Theorem 3.36. Let C be a factorable constraint on n real variables associated with n domains of which the Cartesian product contains $\mathbf{x} \in \mathbb{I}^n$. Then

$$\square(C \cap \mathbf{x}) \subseteq \textsf{HC4revise}(\mathcal{T}_C, \mathbf{x}) \subseteq \textsf{HC4revise*}(\mathcal{T}_C, \mathbf{x}).$$

Proof. See the proof of Proposition 4 of [Benhamou *et al.* 1999]. ■

3.2.2.3. Box Consistency by Search

The concept of *box consistency* is defined in Section 2.3.3.3. In this section, we present a simple search algorithm to achieve box consistency for NCSPs, which was proposed by Benhamou *et al.* [1999]. In Algorithm 3.11, we present an algorithm, called **BC3Revise**, to achieve box consistency for a constraint w.r.t. one of its variables.

The **BC3revise** algorithm is a domain reduction operator for one constraint. It enjoys the following properties: contractiveness, correctness, and monotonicity (see Property 2.44). To achieve box consistency and box(Γ) consistency for an NCSP, a constraint propagation

Algorithm 3.11: The **BC3Revise** algorithm – box consistency by search

Input: an interval form Γ of an NCSP $\mathcal{P}$, domains $\mathbf{x} \in \mathbb{I}^n$, $i \in \{1, \ldots, n\}$.
Output: new domains $\mathbf{x}' \in \mathbb{I}^n$ of $\mathcal{V}$.
$\mathbf{l} \equiv [\underline{l}, \overline{l}] :=$ **SearchLowerBound**$(\Gamma, \mathbf{x}, i)$; ◂ On page 85.
if $\mathbf{l} = \emptyset$ **then return** $\emptyset$;
$\mathbf{x}' := \mathbf{x}$; $\mathbf{x}'_i := [\underline{l}, \overline{\mathbf{x}}_i]$;
$\mathbf{u} \equiv [\underline{u}, \overline{u}] :=$ **SearchUpperBound**$(\Gamma, \mathbf{x}', i)$; ◂ On page 85.
$\mathbf{x}'_i := [\underline{l}, \overline{u}]$;

Function SearchLowerBound(in: an interval form Γ, $\mathbf{x} \in \mathbb{I}^n$, $i \in \{1, \ldots, n\}$)

if $\mathbf{x} \notin \Gamma$ **then return** $\emptyset$;
if $\mathbf{x}_i$ is canonical **then return** $\mathbf{x}_i$;
$\mathbf{c} := \mathbf{x}$; $\mathbf{c}_i := [\underline{x}_i, \underline{x}_i^+]$;
if $\mathbf{c} \in \Gamma$ **then return** $\mathbf{c}_i$;
$t := \text{mid}([\underline{x}_i^+, \overline{x}_i])$;
if $\mathbf{d} \equiv$ **SearchLowerBound**$(\Gamma, [\underline{x}_i^+, t], i) \neq \emptyset$ **then return d**;
return SearchLowerBound$(\Gamma, [t, \overline{x}_i], i)$;

Function SearchUpperBound(in: an interval form Γ, $\mathbf{x} \in \mathbb{I}^n$, $i \in \{1, \ldots, n\}$)

if $\mathbf{x} \notin \Gamma$ **then return** $\emptyset$;
if $\mathbf{x}_i$ is canonical **then return** $\mathbf{x}_i$;
$\mathbf{c} := \mathbf{x}$; $\mathbf{c}_i := [\overline{x}_i^-, \overline{x}_i]$;
if $\mathbf{c} \in \Gamma$ **then return** $\mathbf{c}_i$;
$t := \text{mid}([\underline{x}_i, \overline{x}_i^-])$;
if $\mathbf{d} \equiv$ **SearchUpperBound**$(\Gamma, [t, \overline{x}_i^-], i) \neq \emptyset$ **then return d**;
return SearchUpperBound$(\Gamma, [\underline{x}_i, t], i)$;

Algorithm 3.14: The **BC3** algorithm – box(Γ) consistency by bounds search + **AC3**

Input: interval forms $\Gamma = \{\Gamma_1, \ldots, \Gamma_n\}$ of an NCSP $\mathcal{P} \equiv (\mathcal{V}, \mathcal{D}, \mathcal{C})$, a box $\mathbf{x} \subseteq \mathcal{D} \subseteq \mathbb{R}^n$.
Output: new domains $\mathbf{x}' \in \mathbb{I}^n$ of $\mathcal{V}$.

```
x' := x;
WaitingList := C;
while WaitingList ≠ ∅ and x' ≠ ∅ do
    Take a constraint C from WaitingList;
    foreach variable x_i constrained by C do
        y := BC3Revise(Γ_i, x', i);
        if x'_i ≠ y then
            WaitingList := WaitingList ∪ {C' ∈ C | C' ≠ C, x_i is contrained by C'};
            x'_i := y;
        end
    end
end
```

algorithm similar to the **AC3** algorithm (which maintains arc consistency, see [Mackworth 1977; Tsang 1993]) is performed, where **BC3revise** is in place of arc consistency for each constraint of the NCSP. This propagation algorithm, called **BC3**, is presented in Algorithm 3.14.

3.2.2.4. Box Consistency by Propagation

In Algorithm 3.15, we present a version of *box consistency*, called **BC4**, which was proposed by Benhamou *et al.* [1999]. The **BC4** algorithm is based on the **HC4revise** algorithm (Section 3.2.2.2) and the **BC3** algorithm (Section 3.2.2.3).

In the rest of this section, we use the notations defined as follows.

Notation 3.37. Consider a factorable NCSP $\mathcal{P} \equiv (\mathcal{V}, \mathcal{D}, \mathcal{C})$, where $\mathcal{V} \equiv (x_1, \ldots, x_n)$ and $\mathcal{D}$ contains a box $\mathbf{x} \equiv (\mathbf{x}_1, \ldots, \mathbf{x}_n)^{\mathrm{T}} \in \mathbb{I}^n$.

- Let denote by $\mathbf{C}$ the natural interval form of each $C \in \mathcal{C}$, by $\mathbf{C}_{x_k}$ the interval constraint obtained from $\mathbf{C}$ by replacing all variables x_i with its domain, $\mathbf{x}_i$, where $i \neq k$.
- Let $\mathcal{C}^1_x$ the set of constraints of $\mathcal{C}$ in which x occurs exactly one time and $\mathcal{C}^{2+}_x$ the set of constraints of $\mathcal{C}$ in which x occurs more than one time.
- Let $\mathcal{S}^1_{x_i} \equiv \{\mathbf{C}_{x_i} \mid C \in \mathcal{C}^1_{x_i}\}$ and $\mathcal{S}^{2+}_{x_i} \equiv \{\mathbf{C}_{x_i} \mid C \in \mathcal{C}^{2+}_{x_i}\}$.

The following theorem characterizes the relation between the **HC4revise*** algorithm (Section 3.2.2.2) and *box*(Γ) *consistency*. It follows that the **HC4revise*** algorithm is an algorithm for achieving box(Γ) consistency for a constraint, under the assumption in Theorem 3.38.

Theorem 3.38. Let C be a constraint of $\mathcal{C}$ as described in Notation 3.37. Suppose x_k is a variable that occurs in C only one time. Denote $\Gamma \equiv \{\mathbf{C}, \ldots, \mathbf{C}_{x_k}, \ldots, \mathbf{C}\}$, where $\mathbf{C}_{x_k}$ is located at the k-th position. Then C is box(Γ) consistency in $\mathbf{x}'$ on x_k, where $\mathbf{x}' :=$ **HC4revise***$(\mathcal{T}_C, \mathbf{x})$.

Proof. See the proof of Proposition 3 of [Benhamou *et al.* 1999]. ∎

Algorithm 3.15: The **BC4** algorithm – box(Γ) consistency by **BC3** + **HC4revise**

Input: an NCSP $\mathcal{P} \equiv (\mathcal{V} \equiv (x_1, \ldots, x_n), \mathcal{D}, \mathcal{C})$, a box $\mathbf{x} \subseteq \mathcal{D}$.
Output: new domains $\mathbf{x}' \in \mathbb{I}^n$ of $\mathcal{V}$.

```
repeat
    x' = x;
    repeat
        CONTINUE := false;
        foreach C ∈ 𝒞 do
            x'' := x';
            x' := HC4revise(𝒯_C, x');
            𝒱¹_C = {x ∈ vars(C) | x occurs in C once};
            CONTINUE := (x' ≠ ∅ ∧ (CONTINUE ∨ (∃k : x'_k ≠ x''_k, x_k ∈ 𝒱¹_C)))
        end
    until CONTINUE = false;
    if x' ≠ ∅ then
        Γ²⁺ := (𝒮²⁺_{x_1}, ..., 𝒮²⁺_{x_n})  (see Notation 3.37);
        x' := BC3(Γ²⁺, x');
    end
until x' = x or x' = ∅;
```

Now let **BC4*** be the algorithm obtained from the **BC4** algorithm by replacing **HC4revise** (in Algorithm 3.15) with **HC4revise*** (see Section 3.2.2.2). The **BC4*** algorithm is an algorithm for achieving box(Γ) consistency for NCSPs, under the assumption in Theorem 3.39.

Theorem 3.39. Consider an NCSP $\mathcal{P}$ and the related notations in Notation 3.37. Let $\Gamma = (\Gamma_1, \ldots, \Gamma_n)$, where $\Gamma_i \equiv \mathcal{S}^1_{x_i} \cup \mathcal{S}^{2+}_{x_i}$ for all $i = 1, \ldots, n$. Then $\mathcal{P}$ is box(Γ) consistent in $\mathbf{x}' := \mathsf{BC4^*}(\mathcal{P}, \mathbf{x})$.

Proof. See the proof of Proposition 5 of [Benhamou *et al.* 1999]. ∎

3.2.2.5. *k*B-Consistency by Search

The concepts of kB-*consistency* and kB($\mathbb{F}$)-*consistency* are described in Section 2.3.3.2. On the same floating-point number system, $\mathbb{F}$-hull consistency and kB($\mathbb{F}$)-consistency are equivalent. Therefore, we refer to Algorithm 3.6 as a procedure to compute 2B($\mathbb{F}$)-consistency, where the procedure at Line 1 in Algorithm 3.6 is incrementally performed by search.

In Algorithm 3.16, we present an algorithm, which was proposed by Bordeaux *et al.* [2001], to achieve kB($\mathbb{F}$)-consistency for NCSPs (with $k \geq 3$). This algorithm is equivalent to the function $\Phi^{\mathbb{F}}_{k\mathrm{B}}$ in Definition 2.123. In this algorithm, each time a canonical interval bound σ at Line 1 in Algorithm 3.16 is considered, if a nonempty fixed point of the procedure $\Phi^{\mathbb{F}}_{(k-1)\mathrm{B}}$ has been computed for larger domains than the slice of the considered NCSP in σ (and has been cached at $\mathtt{CACHE}(\sigma)$), then there is no need to repeat the procedure. That is why there is a check at Line 2. When a canonical interval bound is reconsidered, at Line 3, it possible to restart the procedure from the domains $\mathbf{x} \cap \mathtt{CACHE}(t)$ because we have

$$\Phi^{\mathbb{F}}_{(k-1)\mathrm{B}}(\mathcal{P}|_{x_i \in \sigma}) = \Phi^{\mathbb{F}}_{(k-1)\mathrm{B}}(\mathcal{P}|_{x \in \mathbf{x} \cap \mathtt{CACHE}(\sigma)}).$$

The reader can find the proof in [Bordeaux *et al.* 2001]. The main loop starting at Line 4 is to find the outermost (thick) canonical interval bound such that the procedure $\Phi^{\mathbb{F}}_{(k-1)\mathrm{B}}$ cannot prove the inconsistency. It is an incremental search. After this search, the reduced domains are cached (at Line 5).

Algorithm 3.16: The kB-**Consistency** algorithm – the function $\Phi^{\mathbb{F}}_{k\mathrm{B}}$

```
Input: an NCSP P with domains x ≡ [x̲, x̄] ∈ 𝕀^n of (x_1, ..., x_n).
Output: new domains x' ∈ 𝕀^n of (x_1, ..., x_n).
for i := 1, ..., n do
  | CACHE([x̲_i, x̲_i^+]) := CACHE([x̄_i^-, x̄_i]) := x;
end
repeat
1   foreach i := 1, ..., n; σ in {[x̲_i, x̲_i^+], [x̄_i^-, x̄_i]} do
2     if CACHE(σ) ⊈ x then
3       CACHE(σ) := Φ^F_(k-1)B(P|_{x∈x∩CACHE(σ)});
        if CACHE(σ) = ∅ then
4         while σ ∩ x_i ≠ ∅ ∧ Φ^F_(k-1)B(P|_{x_i∈σ}) = ∅ do
            if σ = [x̲_i, x̲_i^+] then
              | x̲_i := x̲_i^+; σ := [x̲_i, x̲_i^+];
            end
            if σ = [x̄_i^-, x̄_i] then
              | x̄_i := x̄_i^-; σ := [x̄_i^-, x̄_i];
            end
          end
          if σ ∩ x_i = ∅ then return x' := ∅;
5         CACHE(σ) := Φ^F_(k-1)B(P|_{x_i∈σ});
        end
      end
    end
until x is unchanged;
x' := x;
```

Bordeaux *et al.* [2001] proved that the algorithm in Algorithm 3.16 computes a $k\mathrm{B}(\mathbb{F})$-consistent box in the optimal time complexity $\mathcal{O}(mN^{k-1}n^{k-2})$, where N is the maximum number of (floating-point) elements in a domain, n is the number of variables, m is the number of constraints. Note that the time complexity of the old algorithm proposed by Lhomme [1993] to achieve kB-consistency is $\mathcal{O}(mN^{k-1}n^{2k-4})$.

3.2.3. Cooperation of Solution Techniques

3.2.3.1. Cooperation of Domain Reduction Techniques

The cooperation of interval constraint propagation techniques has been proposed in [Van Hentenryck *et al.* 1997a,b]. This has led to the implementation in the systems Newton and Numerica, which efficiently combine the box consistency notions with different interval forms of constraints. These ideas have been extended in [Benhamou *et al.* 1999] and [Granvilliers *et al.* 1999] in order to avoid useless redundant computations and accelerate local computations.

These techniques essentially combine the box consistency notions over natural and centered interval forms with the hull consistency notion, for example, as described in Section 3.2.2.4.

There have been some strategies to schedule (or combine) variable reductions [Lhomme *et al.* 1998], different local consistency notions [Benhamou *et al.* 1999], or preconditioning matrices [Kearfott and Shi 1996], parallel computations [Granvilliers and Hains 2000], and distributed computations [Monfroy and Réty 1999]. Generic algorithms [Granvilliers and Monfroy 2000] were also proposed to combine several domain reduction techniques in one solver.

Algorithm 3.17: The **Fixed Point Combination** algorithm

Input: $\mathbf{x} \in \mathbb{I}^n$
Output: $\mathbf{x}' \in \mathbb{I}^n$
$k := 0$; $\mathbf{x}^{\langle 0 \rangle} := \mathbf{x}$;
repeat
  $k := k + 1$;
  choose an operator R in $\mathcal{L}$ according to strategy $\mathcal{S}$;
  $\mathbf{x}^{\langle k \rangle} := R(\mathbf{x}^{\langle k-1 \rangle})$;
until $\mathbf{x}^{\langle k \rangle}$ is a fixed point of all operators in $\mathcal{L}$;
$\mathbf{x}' := \mathbf{x}^{\langle k \rangle}$;

In Algorithm 3.17, we present a generic algorithm, called **Fixed Point Combination**, to combine domain reduction operators from a finite list $\mathcal{L}$. A strategy to choose an operator from $\mathcal{L}$ is called a *fair strategy* if, for any $k > 0$ and any operator R in $\mathcal{L}$, there exists a finite step $k_1 > k$ at which R is chosen. Several basic strategies for collaborations between interval domain reduction operators have been discussed in [Jaulin *et al.* 2001, Section 4.4]. Jaulin *et al.* [2001, Theorem 4.3] pointed out that, given a set $\mathcal{L}$ of monotonic domain reduction operators, the **Fixed Point Combination** algorithm converges, for any fair strategy $\mathcal{S}$, to the same fixed point that is the largest box included in the initial domain such that it is the fixed point of every operator in $\mathcal{L}$. In order to reduce time for convergence, the strategy $\mathcal{S}$ must be dynamically designed so that the slow convergence of operators can be identified (for instance, as in the *cooperative strategy* [Lhomme *et al.* 1998]).

Moreover, Lhomme *et al.* [1998] showed that it is not efficient to combine a set of domain reduction operators which compute a fixed point of an iteration because this may result in *slow convergence*. Therefore, one should stop at a desired precision (e.g., see the definition of box$\langle \pm\varphi \rangle$ consistency in Definition 2.127 and the method in [Granvilliers *et al.* 1999]).

Recently, several combinations of different local consistency notions were described in [Granvilliers 2001]. This paper tackles the issue of combining the hull consistency and box consistency notions, and the interval Newton method into a new domain reduction technique. This combination, however, tends to be symbolic-based and inflexible. Namely, the choice of consistency techniques is based on the number of occurrences of variables in constraints.

3.2.3.2. Cooperation of Symbolic-Interval Techniques

A good survey on symbolic-interval cooperation was given in [Granvilliers *et al.* 2001], also in [Bliek *et al.* 2001, Chapter 6]. Therein, three types of symbolic-interval techniques for improving the solution of a nonlinear constraint system were discussed:

(i) The simplification of constraint expressions for tackling the *dependency* problem of interval arithmetic;

(*ii*) The combination of constraint expressions for handling the independence of computations;

(*iii*) The cooperation of solvers for collectively solving a constraint system.

However, the control of the percentages of symbolic and numerical computations in order to accelerate the solving process remains a difficult issue for the above techniques.

Several cooperation strategies were proposed in [Benhamou and Granvilliers 1997], They are however not possible to decide *a priori* which one will be the most efficient for solving a given constraint system. To date, various symbolic techniques have been tried in symbolic-interval constraint solving, such as *Gröbner computation*, *factorization*, *simplex method*, *Gaussian elimination*, *substitution*, and *abstraction*. The theoretical cooperation schemes based on shared variables in [Benhamou 1996] and [Apt 1999] seem to be suitable in interval constraint propagation. When dealing with the symbolic-numerical cooperation, however, these schemes require an ordering on the manipulated elements. Unfortunately, defining this ordering is a difficult issue, if not impossible in numerous cases.

3.3. Relaxation Based Methods

3.3.1. Linear Relaxation with Linear Programming

In order to reduce the domains of a numerical constraint satisfaction problem (NCSP), $\mathcal{P} \equiv (\mathcal{V}, \mathcal{D}, \mathcal{C})$, an efficient approach is to generate a linear relaxation system $\mathcal{P}'$ of $\mathcal{P}$ (Definition 2.38) by using linear relaxation techniques. The obtained relaxation system can be explored further (e.g., by resorting to *linear programming* techniques) to evaluate lower and upper bounds of variables. Namely, the lower and upper bounds of each variable x in $\mathcal{V}$ satisfying the constraints of $\mathcal{P}'$ can be computed by solving the following two *linear programs*, respectively:

$$\begin{aligned} \text{minimize} \quad & x \\ \text{s.t.} \quad & \mathcal{P}', \end{aligned}$$

and

$$\begin{aligned} \text{maximize} \quad & x \\ \text{s.t.} \quad & \mathcal{P}'. \end{aligned}$$

Because the latter can be easily converted into the former by replacing x with $-x$, we will limit the presentation to minimization problems in the rest of this section.

In general, computing bounds on n variables amounts to solving $2n$ minimization problems. Any linear programming technique can be employed to compute approximate bounds on variables. However, to get rigorous bounds, one often has to use rigorously bounding techniques with directed rounding controls. To name a few among them, we refer to the interval-based simplex method by Kashiwagi [1996, 1997], the safe procedure for embedding directed rounding controls into the framework of constraint programming by Michel *et al.* [2003], and the generic safe bounding technique by Neumaier and Shcherbina [2004]. It is interesting that the post-processing technique in [Neumaier and Shcherbina 2004] can provide rigorous bounds of reasonable size on the objective functions of linear programs without interfering in the source codes of simplex-based linear programming solvers.

3.3.1.1. Linear Relaxation Based on Quadratic Terms

In [McCormick 1976, 1983] and [Al-Khayyal 1990; Al-Khayyal and Falk 1983], *quadratic terms*, such as x^2 and xy, is relaxed to get linear inequalities. Namely, a term x^2, where x is a real variable taking its value in $\mathbf{x} \equiv [\underline{x}, \overline{x}]$, can be relaxed by introducing a new variable z in place of x^2:

$$-(\underline{x} + \overline{x})x + z + \underline{x}\overline{x} \leq 0. \tag{3.95}$$

The domain for the new variable z is given by $\mathbf{z} := \mathbf{x}^2$. The term xy, where x and y are two real variables taking their values in $\mathbf{x} \equiv [\underline{x}, \overline{x}]$ and $\mathbf{y} \equiv [\underline{y}, \overline{y}]$ respectively, can also be relaxed by introducing a new variable z in place of xy: the inequalities of the form

$$bx + ay - ab \leq z \leq dx + cy - cd \tag{3.96}$$

hold for all $(a, b) \in \{(\underline{x}, \underline{y}),\ (\overline{x}, \overline{y})\}$ and $(c, d) \in \{(\underline{x}, \overline{y}),\ (\overline{x}, \underline{y})\}$. The domain of the new variable z is set to the interval product $\mathbf{z} := \mathbf{xy}$.

In [Lebbah *et al.* 2003a,b], any *power term* of the form $x_1^{n_1} \dots x_k^{n_k}$ is recursively composed of terms of the form x^2 and xy. Therefore, linear relaxations of a system of factorable constraints can be generated by recursively resorting to the rules (3.95) and (3.96). Many new auxiliary variables are introduced in this procedure. After obtaining a system $\mathcal{L}$ of linear relaxations for the considered constraint system, a constraint propagation procedure, called **Quad**, is enforced in order to reduce domains of initial variables. The **Quad** algorithm is presented in Algorithm 3.18. Note that the structure of $\mathcal{L}$ is unchanged after each loop step (Line 2 in Algorithm 3.18), only coefficients in $\mathcal{L}$ will be changed and these changes can be easily updated by fixed simple formulas.

Algorithm 3.18: The **Quad** algorithm – a propagation based on linear relaxations

Input: an NCSP $\mathcal{P} \equiv (\mathcal{V}, \mathcal{D}, \mathcal{C})$, a box $\mathbf{x} \subseteq \mathcal{D} \subseteq \mathbb{R}^n$.
Output: new domains $\mathbf{x}' \in \mathbb{I}^n$ of $\mathcal{V}$.
1 Generate a system $\mathcal{L}$ of linear relaxations based on the rules (3.95) and (3.96);
$\mathbf{x}' := \mathbf{x}$;
2 **while** $\mathbf{x}' \neq \emptyset$ **and** the reduction of $\mathbf{x}'$ is greater than ε **do**
  Update the coefficients of $\mathcal{L}$ according the new domains $\mathbf{x}'$;
3   Reduce the domains $\mathbf{x}'$ of $\mathcal{V}$ by resorting to a linear programming technique on $\mathcal{L}$;
end

3.3.1.2. More General Linear Relaxation Techniques

Many techniques have been proposed to generate linear relaxations of a nonlinear system of forms that are more general than quadratic. For example, there are linear relaxation techniques based on *power terms* [Borradaile and Van Hentenryck 2004] and on rational power/quotient terms [Hongthong and Kearfott 2004] of factorable constraints. A more general linear relaxation method was also proposed in [Georg 2003], where the *high order terms* in the Taylor expansion of differentiable functions/constraints is dominated by simpler form (see Section 3.1.3.2). There are also general linear relaxation techniques which can be made rigorous, as described in Section 3.3.3.

It is clear that any linear relaxation technique that keeps initial variables, such as the above mentioned ones, can be used in place of Line 1 in Algorithm 3.18. Domain reduction techniques

described in the sections 3.1.1.1, 3.1.1.2, 3.1.2, and 3.3.3 can also be used in place of Line 3 in Algorithm 3.18 at some stages. The exclusion techniques in Section 3.3.2 can be used at that place. However, the structure of $\mathcal{L}$ may vary; hence, the cost for updating the coefficients in $\mathcal{L}$ may be prohibitively high.

3.3.2. Exclusion Test Using Linear Programming

3.3.2.1. Exclusion Test Using Dual Simplex Method

In general, the computation of tight bounds of variables of a linear relaxation system (i.e., a system of linear equations/inequalities) is still very cost for the purpose of domain reduction. One may reduce this purpose to an exclusion test (see Section3.1.3). That is, the inconsistency of linear relaxation system implies the inconsistency of the initial CSP. Moreover, when using the branch-and-prune methods to solve the initial CSP, search nodes (branches) often share some common properties with their parent. Therefore, some information computed at a level may be reused for the next level, in the search tree. Yamamura and Fujioka [2003]; Yamamura and Tanaka [2002] followed these common ideas to devise efficient exclusion tests from linear relaxations. In particular, they use the first phase of the simplex method for this purpose. Their ideas are summarized hereafter.

We recall that the *simplex method* consists of two phases. In the first phase of the simplex method, the constraint system in a linear program is converted into a system of equations by introducing slack variables (see Section 2.1.3.2) of the standard form for the simplex method

$$\begin{aligned} \text{minimize} \quad & z \equiv cx \\ \text{s.t.} \quad & Ax = b, \\ & x \geq 0, \end{aligned}$$

where x is a vector of n real variables, A is a $m \times n$ real matrix, b is a vector of m real numbers, c is a vector of n real numbers. If A has rank m, the linear constraints in the program can be transformed into *Gauss-Jordan form* and written in a so-called *basic tableau*. The general form in Table 3.1, where x_B are called *basic variables* and x_N are called *nonbasic variables*. The reduced form of basic tableau in given in Table 3.2. A *basic feasible solution* to the basic tableau given in Table 3.1 is a basic solution (i.e., $x_N = 0$) to the corresponding system of equations such that $x_B \geq 0$. A basic feasible solution can be found by using *artificial variables*. In case no basic feasible solution is found, then the first phase of the simplex method terminates with the inconsistency confirmed [Press *et al.* 1992]. The second phase of the simplex method starts with a basic feasible solution obtained by the first phase to optimize the objective function. It is an prime *iterative descent method*. The simplex method applying to the dual linear program of the considered linear program is called the *dual simplex method*.

Table 3.1. The general form of a basic tableau in the simplex method

basis	$-z$	x_B	x_N	rhs
x_B	0	I	N	b'
$-z$	1	0	c'_N	$-z_0$

Table 3.2. The reduced form of a basic tableau in the simplex method

basis	x_N	rhs
x_B	N	b'
$-z$	c'_N	$-z_0$

When using the simplex method to devise an exclusion test, we do not need the second of the simplex method since the objective function is an arbitrary constant. A square system of nonlinear equations can be relaxed (e.g., by using interval arithmetic [Yamamura 2000, 2003; Yamamura and Fujioka 2003; Yamamura and Tanaka 2002]) to a linear program of the form

$$\begin{aligned} \text{minimize} \quad & \text{constant} & (3.97) \\ \text{s.t.} \quad & Px^0 + Qy^0 = r, \\ & a \le x^0 \le b, & (3.98) \\ & c \le y^0 \le d, \end{aligned}$$

where x^0 is a vector of n initial real variables, y^0 is a vector of n auxiliary real variables in place of the nonlinear terms in n nonlinear equations, and $a, b, c, d \in \mathbb{R}^n$. In (3.97), by substituting $x \equiv x^0 - a$, $y \equiv y^0 - c$, and introducing two suitable vectors u and v of slack variables of size n; we obtain a linear program of the standard form

$$\begin{aligned} \text{minimize} \quad & \text{constant} & (3.99) \\ \text{s.t.} \quad & Px + Qy = r, \\ & x + u = b - a, \\ & y + v = d - c, \\ & x \ge 0,\ y \ge 0,\ u \ge 0,\ v \ge 0. \end{aligned}$$

This system has $3n$ equations. Because both $b - a \ge 0$ and $d - c \ge 0$ hold, we only need to introduce n artificial variables for the first n equations: $Px + Qy = r$. Then u, v and these artificial variables are used as the initial basic variables, while x, y are used as the initial nonbasic variables. Hence, the size of the resulting basic tableau is $(3n + 2) \times (2n + 3)$.

Let the *optimal feasible tableau* be, for example,[4] of the pattern in Table 3.3, where $\alpha + \beta = b_k - c_k$. The term *optimal* implies that the optimality condition is satisfied in the *auxiliary objective function low* [Press *et al.* 1992]. Depending on which variables are chosen to be basic variables, there are nine possible patterns for optimal feasible tableaus. Note that the sum of the x_k-row and the u_k-row in Table 3.3 is $(0, \dots, 0, b_k - a_k)$.

A branch-and-prune method resorting to the above exclusion test usually restarts the first phase of the simplex method from the scratch. Hence, the total number of pivots to be used by the simplex method is often very huge for a large scale system. This leads to an unacceptable running time for large scale systems.

We now consider to bisect the domain $[a_k, b_k]$ of a variable x_k in the original nonlinear problem, where $1 \le k \le n$. Without loss of generality, we consider the first half $[a'_k, b'_k] \equiv [a_k, b'_k]$, where b'_k is the bisection point. This bisection leads to the fact that the corresponding

[4] The results are similar for the other patterns of the optimal feasible tableau.

Table 3.3. The optimal feasible tableau for the linear program (3.99)

basis	$\dots$	v_k	$\dots$	rhs
$\vdots$		$\vdots$		$\vdots$
x_k	$\dots$	γ	$\dots$	α
$\vdots$		$\vdots$		$\vdots$
u_k	$\dots$	$-\gamma$	$\dots$	β
$\vdots$		$\vdots$		$\vdots$
y_k	0	-1	0	$d_k - c_k$
$\vdots$		$\vdots$		$\vdots$
$-z$	0	0	0	$-z_0$

linear program to be considered has the form

$$\begin{array}{rl} \text{minimize} & \text{constant} \\ \text{s.t.} & Px' + Qy' = r, \\ & x_i + u_i = b_i - a_i, \quad (\text{for } i = 1, \dots, n;\ i \neq k) \\ & x'_k + u'_k = b'_k - a'_k, \\ & y_i + v_i = d_i - c_i, \quad (\text{for } i = 1, \dots, n;\ i \neq k) \\ & y'_k + v'_k = d'_k - c'_k, \\ & x' \geq 0,\ y' \geq 0,\ u' \geq 0,\ v' \geq 0, \end{array}$$

where $[c'_k, d'_k]$ is the new domain of y_k,[5] $x'_k \equiv x^0_k - a'_k$, $y'_k \equiv y^0_k - c'_k$, and

$$\begin{array}{rcl} x' & \equiv & (x_1, \dots, x_{k-1}, x'_k, x_{k+1}, \dots, x_n)^{\mathrm{T}}, \\ y' & \equiv & (y_1, \dots, y_{k-1}, y'_k, y_{k+1}, \dots, y_n)^{\mathrm{T}}, \\ u' & \equiv & (u_1, \dots, u_{k-1}, u'_k, u_{k+1}, \dots, u_n)^{\mathrm{T}}, \\ v' & \equiv & (v_1, \dots, v_{k-1}, v'_k, v_{k+1}, \dots, v_n)^{\mathrm{T}}. \end{array}$$

Notice that (3.99) and (3.100) are different only in the constant terms, ignoring the meaning of variables. The numerical relations between old and new variables are

$$\begin{array}{rcl} x'_k & = & x_k, \\ u'_k & = & u_k - (b_k - b'_k), \\ y'_k & = & y_k - (c'_k - c_k), \\ v'_k & = & v_k - (d_k - d'_k). \end{array}$$

Substituting these relations into the optimal feasible tableau computed for (3.99), we get the *optimal tableau* in Table 3.4 for the linear program (3.100) easily with no cost. The new optimal tableau in Table 3.4 differs the one in Table 3.3 only in the right hand side (rhs) column. Note that, all other elements in the rhs column change values, but they are not written for simplicity.

[5] We recall that y_k represents the bounds on the nonlinear term of the k-th equation in the original square nonlinear system.

Table 3.4. The optimal tableau for the linear program (3.100)

basis	$\ldots$	v'_k	$\ldots$	rhs
$\vdots$		$\vdots$		$\vdots$
x'_k	$\ldots$	γ	$\ldots$	$\alpha + \gamma(d_k - d'_k)$
$\vdots$		$\vdots$		$\vdots$
u'_k	$\ldots$	$-\gamma$	$\ldots$	$\beta - (b_k - b'_k) - \gamma(d_k - d'_k)$
$\vdots$		$\vdots$		$\vdots$
y'_k	0	-1	0	$d'_k - c'_k$
$\vdots$		$\vdots$		$\vdots$
$-z$	0	0	0	$-z_0$

The optimal tableau in Table 3.4 may be infeasible (i.e., constants on the rhs column may be negative), but always dual feasible [Yamamura and Tanaka 2002]. Therefore, starting from this tableau, we can perform the dual simplex method and check the consistency, instead of the inconsistency. In most cases, this dual simplex method requires only very few pivots.

Yamamura and Tanaka [2002] showed that this method is very efficient for system of equations the form $f(x) \equiv Px + Qg(x) - r = 0$, where $g : \mathbb{R}^n \to \mathbb{R}^n$ is a function with component $g_i(x_i)$ for all $i = 1, \ldots, n$. Note that, general nonlinear factorable functions can be converted to this form (see Section 2.1.3.4).

The reader can find some related works in [Kashiwagi 1996, 1997], [Nakaya and Oishi 1998], [Yamamura *et al.* 1998], [Yamamura and Nishizawa 1999], [Yamamura and Hata 2000], and [Yamamura and Kumakura 2001], .

3.3.2.2. Exclusion Test Using Taylor Expansion and Linear Programming

In Section 3.1.3.2, we have studied that Georg [2003] used the Taylor expansion to devise exclusion test. He also devise linear programs from the *domination* of nonlinear terms in the Taylor expansion of functions. The resulting system can obviously benefit from any linear programming techniques. In this section, we use the same notations as in Section 3.1.3.2.

From the Taylor expansion with integral remainder (3.73), Georg [2003, Theorem 14] proved the following result, which can be used as an exclusion test.

Suppose $f \prec_k g : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$, where $k > 1$ (see also Section 3.1.3.2). Let $\mathbf{x} \in \mathbb{I}^n$ be a box in D. Then a necessary condition for f to have a zero in $\mathbf{x}$ is that the following linear program has optimal value zero:

$$\begin{aligned}
\text{minimize}\quad & e^{\mathrm{T}}t \qquad (3.100)\\
\text{s.t.}\quad & Ef'(c)x + Ey + t = p,\\
& x + u = 2r,\\
& y + v = 2d,\\
& x \geq 0,\ y \geq 0,\ u \geq 0,\ v \geq 0,\ t \geq 0,
\end{aligned}$$

where $E \in \mathbb{R}^{n \times n}$ is a diagonal matrix with ± 1 on its diagonal, $c \equiv \mathrm{mid}(\mathbf{x})$, $r \equiv \mathrm{rad}(\mathbf{x})$, and

$$p \;\equiv\; E\left(-f(c) + f'(c)r + d\right) \geq 0,$$

$$d \equiv g(|c+r|) - g(|c|) - g'(|c|)r - \sum_{\mathbf{i}\in\mathbb{N}^n,\ 0<\|\mathbf{i}\|_1<k} (\partial^{\mathbf{i}} g(|c|) - \partial^{\mathbf{i}} f(c)) r^{\mathbf{i}} \geq 0.$$

Note that the linear program (3.100) has the feasible basis $t = p$, $u = 2r$, and $v = 2d$, which can be used to start the simplex methods. Similarly to the argument in Section 3.3.2.1, the first phase of the simplex method is sufficient for the purpose of devising an exclusion test. The proof can be found in [Georg 2003]. Georg [2003] also found the following relation in comparison to the Taylor exclusion test in Section 3.1.3.2.

Theorem 3.40. The exclusion test resulting from (3.100) implies the Taylor exclusion test resulting from (3.76).

3.3.3. Linear Relaxation with Fixed Point Methods

Since Yamamura [1993, 1996] had provided a practical method for converting a factorable constraint system into a separable constraint system (Section 2.1.3.4), one started studying interval methods on separable systems. Among the works in this direction, we name a series of successful techniques by Kolev [1998, 1999, 2001, 2002]. Hereafter, we present the main ideas of this series.

3.3.3.1. Linear Relaxation of Separable Functions

We consider a square system of nonlinear equations of the form $f(x) = 0$, where f is a separable function from $D \subseteq \mathbb{R}^n$ to $\mathbb{R}^m$,[6] and $x \equiv (x_1, \ldots, x_n)^{\mathrm{T}}$ is a vector of n real variables. The domains to be considered is a box $\mathbf{x} \equiv (\mathbf{x}_1, \ldots, \mathbf{x}_n)^{\mathrm{T}}$. Since the function f is separable, it can be written as

$$f(x) = \sum_{j=1}^{n} f_j(x_j), \tag{3.101}$$

where f_j is a factorable univariate function from $\mathbb{R}$ to $\mathbb{R}^m$. In previous methods, the range of each function $f_j(x_j)$ over $\mathbf{x}$ is often included in interval enclosures by using interval derivatives or interval slopes. Quite differently, Kolev [1998] proposed to enclose it by a hybrid form $\mathcal{L}(\mathbf{x}_j, x_j) = a_j x_j + \mathbf{d}_j$ $(1 \leq j \leq n)$, where $x_j \in \mathbf{x}_j$, $\mathbf{d}_j \equiv [\underline{d}_j, \overline{d}_j] \in \mathbb{I}^m$, and $a_j \in \mathbb{R}^m$. Obviously, the inclusion property should hold; that is,

$$\forall x_j \in \mathbf{x}_j : f_j(x_j) \in a_j x_j + \mathbf{d}_j \quad (\text{for } 1 \leq j \leq n). \tag{3.102}$$

Let $\mathbf{x}_j \equiv [\underline{x}_j, \overline{x}_j]$, where $\underline{x}_j \neq \overline{x}_j$, then the vector a_j can be easily computed by the *real slope*

$$a_j = (f_j(\overline{x}_j) - f_j(\underline{x}_j))/(\overline{x}_j - \underline{x}_j) \quad (\text{for } 1 \leq j \leq n), \tag{3.103}$$

where operations are performed componentwise. The remaining task is to enclose the range $f_j(\mathbf{x}_j)$ between two vectors of lines $L_j(x_j) \equiv a_j x_j + \underline{d}_j$ and $U_j(x_j) \equiv a_j x_j + \overline{d}_j$; that is,

$$\forall x_j \in \mathbf{x}_j : L_j(x_j) \leq f_j(x_j) \leq U_j(x_j) \quad (\text{for } 1 \leq j \leq n). \tag{3.104}$$

[6] Kolev [1998] assumed that $m = n$, the same argument (presented here) also holds for $m \neq n$.

Suppose f is continuously differentiable,[7] we consider, for each $j \in \{1, \ldots, n\}$, the following m equations:[8]

$$f'_j(x_j) - a_j = 0, \; x_j \in \mathbf{x}_j.$$

Additionally, we ***suppose that this system has only a finite number of solutions***: $x_j^{\langle 1 \rangle}, \ldots, x_j^{\langle N \rangle}$; then

$$\underline{d}_j := \min_{1 \le j \le N} \left\{ f'_j(x_j^{\langle j \rangle}) - a_j x_j^{\langle j \rangle} \right\} \quad (\text{for } 1 \le j \le n), \tag{3.105a}$$

$$\overline{d}_j := \max_{1 \le j \le N} \left\{ f'_j(x_j^{\langle j \rangle}) - a_j x_j^{\langle j \rangle} \right\} \quad (\text{for } 1 \le j \le n), \tag{3.105b}$$

are the best among ones satisfying (3.104), where the operations are interpreted in the componentwise manner. The reader can find the proof and a detailed algorithm in [Kolev 1998].

Once the hybrid form $\mathcal{L}(\mathbf{x}_j, x_j)$ satisfying (3.102) is obtained for all $j \in \{1, \ldots, n\}$, all zeros of $f(x)$ in $\mathbf{x}$ are contained in the solution set of the system of linear equations

$$Ax = b, \; b \in \mathbf{b} \tag{3.106}$$

where $A = -(a_1, \ldots, a_n) \in \mathbb{R}^{m \times n}$, $\mathbf{b} = \sum_{j=1}^{n} \mathbf{d}_j \in \mathbb{I}^m$, and $b \in \mathbb{R}^m$.

Remark 3.41 ([Kolev 1998]). In the above computations, real arithmetic can be replaced by interval arithmetic for the rigor. Therefore, the obtained values for $a_j, \underline{b}_j, \overline{b}_j$ are no longer real numbers. Rigorous bounds on solutions of (3.104) can also be computed by using interval arithmetic (see Section 3.1). However, the matrix A in (3.106) computed by using interval arithmetic is no longer a real matrix, but an interval matrix.

Now assume that $A \in \mathbb{R}^{m \times n}$. Equivalently, the form (3.106) can also be written as

$$Ax \in \mathbf{b}. \tag{3.107}$$

Tight bounds on the variables x of the system (3.107) can be obtained by resorting to linear programming techniques, as discussed in Section 3.3.1. One can also use the linear programming techniques in Section 3.3.2 for checking if the system (3.107) contains no solution. Kolev [1998] proposed a method to get bounds on the variables, which is not based on linear programming technique. Hereafter, we recall the main steps of this method.

Assumption 3.42. The matrix $A \in \mathbb{R}^{m \times n}$ in the system (3.107) is assumed to be invertible. Although Kolev [1998] also assumed $m = n$, the argument given here does not require $m = n$ since we can use the extended inverse of matrix in Definition A.7.

Under Assumption 3.42, it is simple to enclose the hull of the solution set of (3.106) by $\mathbf{x} \cap A^{-1}\mathbf{b}$ because $\square\Sigma(A, \mathbf{b}) \subseteq A^{-1}\mathbf{b}$ (see Definition A.7 and Theorem A.19). If $m = n$, then $\square\Sigma(A, \mathbf{b}) = A^{-1}\mathbf{b}$ (see [Neumaier 1990, p 93]). For convenience, we define the Kolev operator and iteration as follows.

[7] In case the function f is piecewise continuously differentiable, a similar argument could be applied.
[8] We recall that f_j is a vector of m functions mapping from $\mathbb{R}$ to $\mathbb{R}$.

Definition 3.43 (Kolev Operator). For a separable function f defined in (3.101), the following is called the *Kolev operator* for f in $\mathbf{x}$:

$$\mathfrak{O}_{\mathrm{KL}}(f, \mathbf{x}) \equiv A^{-1}\mathbf{b}, \tag{3.108}$$

where $A = -(a_1, \ldots, a_n) \in \mathbb{R}^{m\times n}$, $\mathbf{b} = \sum_{j=1}^{n} \mathbf{d}_j \in \mathbb{I}^m$, a_j is defined by (3.103), and $\mathbf{d}_j \equiv [\underline{d}_j, \overline{d}_j]$ is defined by (3.105). The *Kolev iteration* is defined as

$$\mathbf{x}^{\langle 0\rangle} \equiv \mathbf{x},\ \mathbf{x}^{\langle k+1\rangle} \equiv \mathbf{x}^{\langle k\rangle} \cap \mathfrak{O}_{\mathrm{KL}}(f, \mathbf{x}^{\langle k\rangle}) \quad (\text{for } k \in \mathbb{Z}_+). \tag{3.109}$$

We recall the fixed point property of the Kolev operator [Kolev 1998, Theorem 2.3] for the case $m = n$ as follows.

Theorem 3.44. Let $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$ be a separable *continuously differentiable function*. Given a compact box $\mathbf{x} \subseteq D$. If $\mathfrak{O}_{\mathrm{KL}}(f, \mathbf{x})$ is defined and contained in $\mathbf{x}$, then f has at least a zero in $\mathbf{x}$.

Proof. Let $b(x) = Ax + f(x)$. It follows from (3.102) or (3.104) that $b(x) \in \mathbf{b}$ for all $x \in \mathbf{x}$. Hence, for all $x \in \mathbf{x}$, we have $g(x) \equiv x + A^{-1}f(x) = A^{-1}b(x) \in A^{-1}\mathbf{b} = \mathfrak{O}_{\mathrm{KL}}(f, \mathbf{x}) \subseteq \mathbf{x}$. Since A^{-1} is invertible, $g(x) = x \Rightarrow f(x) = 0$ holds. Hence, it follows from Theorem B.15 that f has a zero in $\mathbf{x}$. ∎

A variant of the Kolev iteration (3.109) were also proposed in [Kolev 1998]. For instance, if $\mathbf{x}_i^{\langle k+1\rangle} \subset \mathbf{x}_i^{\langle k\rangle}$ (or stronger condition: $\mathrm{w}(\mathbf{x}_i^{\langle k+1\rangle}) < \varepsilon\, \mathrm{w}(\mathbf{x}_i^{\langle k\rangle})$) holds for some k and some $i < n$, then we repeat the procedures (3.103) and (3.105) for domains $\mathbf{x}_i^{\langle k+1\rangle}$ to compute $a_i^{\langle k+1\rangle}$ and $\mathbf{d}_i^{\langle k+1\rangle} \equiv [\underline{d}_i^{\langle k+1\rangle}, \overline{d}_i^{\langle k+1\rangle}]$, respectively. The interval vector $\mathbf{b}^{\langle k+1\rangle} \equiv [\underline{b}^{\langle k+1\rangle}, \overline{b}^{\langle k+1\rangle}]$ at the iteration step $(k+1)$ can then be updated as follows:

$$\begin{aligned}
\underline{b}^{\langle k+1\rangle} &:= \underline{b}^{\langle k\rangle} - \underline{d}_i^{\langle k\rangle} + \underline{d}_i^{\langle k+1\rangle},\\
\overline{b}^{\langle k+1\rangle} &:= \overline{b}^{\langle k\rangle} - \overline{d}_i^{\langle k\rangle} + \overline{d}_i^{\langle k+1\rangle},\\
A^{\langle k+1\rangle} &:= -(a_1^{\langle k\rangle}, \ldots, a_{i-1}^{\langle k\rangle}, a_i^{\langle k+1\rangle}, a_{i+1}^{\langle k\rangle}, a_n^{\langle k\rangle}).
\end{aligned}$$

For the case $m = n$, we only need to update the i-th row, r_i, of $(A^{\langle k\rangle})^{-1}$ in order to compute the i-th row of $\mathfrak{O}_{\mathrm{KL}}(f, \mathbf{x}^{\langle k\rangle}) \equiv (A^{\langle k\rangle})^{-1}\mathbf{b}^{\langle k\rangle}$, hence the i-th row of $\mathbf{x}^{\langle k+1\rangle}$. It can be done by solving the system of n linear equations.

$$(A^{\langle k\rangle})^{\mathrm{T}} r_i = e_i,$$

where e_i is the i-th column of the identity matrix. This procedure can be proceeded for $i = 1, \ldots, n$ then change to the usual Kolev iteration. For sufficiently narrow boxes, Kolev's method is convergent of order two. For more details, see [Kolev 1998].

3.3.3.2. Linear Relaxation of Factorable Systems

From Factorable Systems to Separable Systems. Now consider a factorable constraint system. It is converted to a separable constraint system, for example, by using the technique

in Section 2.1.3.4. The obtained system has n original real variables x and k auxiliary real variables y. Using the technique in Section 3.3.3.1, we can get a square system of linear equations

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}, \tag{3.110}$$

where all $A_{11} \in \mathbb{R}^{n\times n}$, $A_{12} \in \mathbb{R}^{n\times k}$, $A_{21} \in \mathbb{R}^{k\times n}$, $A_{22} \in \mathbb{R}^{k\times k}$, $b \in \mathbf{b}_1 \in \mathbb{I}^n, b \in \mathbf{b}_2 \in \mathbb{I}^k$, $x \in \mathbf{x}$, and $y \in \mathbf{y}$.

For simplicity, Kolev [1999] assumed that A_{22} is unit matrix. By eliminating y, we have

$$(A_{11} - A_{12}A_{21})x = b_1 - A_{12}b_2.$$

Let $C = A_{11} - A_{12}A_{21}$ and suppose C is invertible. We have

$$x = C^{-1}b_1 - C^{-1}A_{12}b_2 \in C^{-1}\mathbf{b}_1 - C^{-1}A_{12}\mathbf{b}_2. \tag{3.111}$$

New bounds $\mathbf{x}'$ on x is computed by using (3.111) and then intersected with $\mathbf{x}$; that is, $\mathbf{x}' := \mathbf{x}' \cap \mathbf{x}$. New bounds on y can be computed by using

$$y = b_2 - A_{21}x \in \mathbf{b} - A_{21}\mathbf{x}'. \tag{3.112}$$

This method can be combined with the Kolev iteration methods, as described in [Kolev 1999].

Linear Relaxation Using Affine Arithmetic. In [Kolev 2001, 2002], two linear relaxation methods were also proposed for directly computing linear relaxations of factorable systems. In these methods, a factorable function is composed of elementary operations. Each operation is then computed by using Kolev generalized affine arithmetic or Kolev affine arithmetic (see Section 2.2.3.4).

When using Kolev affine arithmetic (or affine arithmetic) to compute an affine form of a separable factorable function $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ in a compact box $\mathbf{x} \in \mathbb{I}^n$, we get a linear expression of the form

$$f(\hat{x}) = \mathcal{L}(x, y) = -Ax + By + b, \quad \hat{x} \in \mathbf{x}, \tag{3.113}$$

where $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{m\times k}$, $b \in \mathbb{R}^m$, x is vector of n initial variables taking values in $\mathbf{x}$, and y is a vector of k auxiliary variables taking values in a box $\mathbf{y} \in \mathbb{I}^k$. The variables in y are noise variables arising in the computations using (Kolev) affine arithmetic. Therefore, $\mathbf{y} = ([-1, 1], \ldots, [-1, 1])^{\mathrm{T}}$. If x is a zero of f and A^{-1} is defined,[9] then

$$x = A^{-1}By + A^{-1}b \in A^{-1}B\mathbf{y} + A^{-1}b. \tag{3.114}$$

This leads to a new rule for domain reduction for fixed point methods:

$$\mathbf{x} := \mathbf{x} \cap (A^{-1}B\mathbf{y} + A^{-1}b). \tag{3.115}$$

3.3.4. Convexification Based Methods

High order relaxation can also be used for solving NCSPs. For completeness, we only give pointers to very recent papers on this advanced topic: a convexification technique for *nonlinear functions* can be found in [Jansson 2000], a book specializing on convexification [Tawarmalani and Sahinidis 2002], an extended technique for computing rigorous bounds on polynomial relaxations can be found in [Garloff *et al.* 2003]. The reader can find the references therein for more details on this topic.

[9] The inverse A^{-1} can be defined as usual for the case $m = n$, or as in Definition A.7 for the case $m \neq n$.

3.4. Incomplete Search Methods

One of the most well-known stochastic search methods is the *hill climbing* method, also called the *multiple random start* method. It is a general search and has been being used in various areas. Basically, the hill climbing algorithm starts with a random focal point in the search space. Given the current focal point P, all the points near to P according a *neighborhood function* are evaluated by using the *evaluation function*. If there exist some points with higher values than P, one of these points will be picked nondeterministically to become the new focal point. This process continues until the value of the current focal point is higher than the values of the nodes in its neighborhood.

A drawback of the hill-climbing method is that it may get stuck in a local hill. Alternatively, many techniques have been devised to overcome this drawback. Among them are *simulated annealing search* [Kirkpatrick *et al.* 1983], *tabu search* [Glover 1986, 1989, 1990], *min-conflicts heuristic* [Minton *et al.* 1992], *random-walk search* [Selman and Kautz 1993], *connectionist method* (or *connectionist approach*) [Davenport *et al.* 1994], and *genetic search* for CSPs [Eiben *et al.* 1994]. Some of these methods were described in detail in [Tsang 1993, Chapter 8] and [Apt 2003, Chapter 6].

The reader can find a brief description of the above methods in [Torrens Arnal 2003, Appendix B]. Neumaier [2004, Section 7] gave pointers to other incomplete search methods for solving constrained optimization problems, such as *homotopy/continuation* methods, *clustering* methods, *local descent* methods, *response surface* methods. Many of these methods can be adapted to solving CSPs. A special survey on homotopy/continuation methods can be found in [Allgower and Georg 1980]. Very recent *homotopy/continuation* methods for solving systems of equations can also be found in [Inoue *et al.* 2002, 2004, 2001].

Notably, a very recent PhD thesis of van Hermet [2002] provides detailed descriptions and empirical comparisons of several evolutionary computation methods for solving CSPs, including the *initiative evolutionary* method, the *microgenetic iterative descent* method, the *co-evolutionary constraint satisfaction* method, the *stepwise adaptation of weights* method, the *grouping genetic* method, and the *zooming adaptation of weights* method.

Chapter 4

Improvements to Search Strategies for Numerical CSPs

Note: This chapter includes the research conducted jointly with Djamila Sam-Haroud and Marius-Calin Silaghi in [Vu *et al.* 2002, 2003]. This is a major contribution of the thesis.

4.1. Introduction

In design applications (e.g., estimation, robust control [Asarin *et al.* 2000; Jaulin *et al.* 2001]; automation and robotics [Jaulin *et al.* 2001; Neumaier and Merlet 2002]; shape design [Snyder 1992]; civil engineering problems, see Appendix C), the solution set of a numerical constraint satisfaction problem (NCSP) often expresses relevant alternatives that need to be identified as precisely and as completely as possible. In many of such applications the complete solution set of an NCSP is required, while in others it is only highly desired, but not mandatory. For example, in practical contexts such as early stages of a design process, a formulated NCSP may have many solutions that express equally relevant choices. In this case, one often desires to find as many solutions as possible because investigating many solutions at earlier stages potentially increases the chance of success at later stages. It also makes it possible to identify good design alternatives. Therefore, the complete solution set is often sought for, provided that the response time is reasonable. Since a general NCSP is NP-hard, the time for finding all its solutions at a high precision (or within a very small tolerance) are often prohibitively long. In most cases, low or medium precisions are sufficient for real world applications. Hence, there is a trade-off between timely but poor information and slow but more precise information.

Recent interval-based constraint solvers, such as Numerica [Van Hentenryck 1998] and ILOG Solver [ILOG 2003], have shown their ability to find all solutions of certain instances of NCSPs efficiently within an arbitrary positive tolerance. Most of them are essentially *branch-and-prune methods*, which interleave *branching steps* with *pruning steps* (see Section 3.2.2). They mainly focus on improving the pruning steps, while leaving the branching steps unimproved. Namely, they use the domain *bisection* or *dichotomization* for the branching steps and only aim at generating a collection of boxes, each encloses one solution. This approach is referred to as the *point-wise approach*. Such branching strategies have been mainly designed to address NCSPs with *isolated solutions* (see Figure 4.1a). Therefore, they are often inefficient when applied, in a straightforward manner, to NCSPs with *non-isolated solutions* (see Figure 4.1b). In this case, the set of generated enclosures is prohibitively verbose or poorly informative. Neither the

computational time nor the compactness of the solution representation are satisfactory.

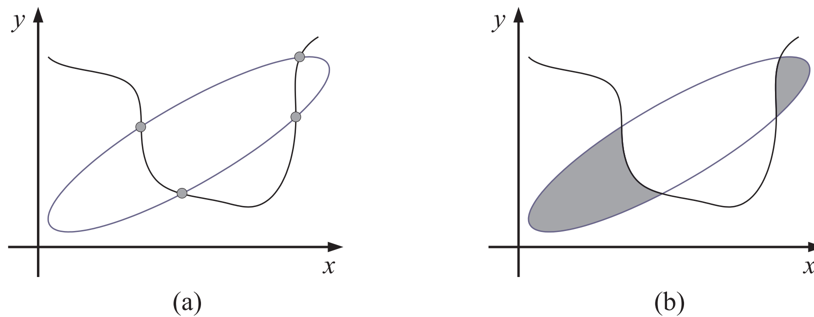

Figure 4.1. (a) An NCSP with four isolated solutions (grey dots); (b) An NCSP with two continuums of solutions (grey regions).

In contrast to the above approach, a new approach has been developed in order to provide promising alternatives to the point-wise techniques. They consist in covering a spectrum of non-isolated solutions with a number of subsets of $\mathbb{R}^n$ (see Definition B.4). These subsets are usually chosen to be simple ones such as interval boxes. Due to theirs simplicity, interval boxes have been used in many *set covering* techniques, which forms *box covering* techniques (see Section 3). These (previously available) algorithms are based on simple domain splitting and have one of the following limitations: (i) they are designed for special constraints only (e.g., universally quantified constraints [Benhamou and Goualard 2000]); (ii) they uniformly enforce dichotomous splitting strategies on all variables (e.g., [Jaulin 1994], [Sam-Haroud 1995; Sam-Haroud and Faltings 1996]). In general, most known algorithms follow a branching strategy that insists on splitting interval domains until canonical intervals are reached or their widths are not greater than a predefined precision. The most successful techniques enhance the solving process by applying inclusion tests or domain reduction techniques to the overall constraint system, after each split (see Section 3.1.6 and Section 3.2.2). This policy is referred to as *dichotomous maintaining bounds by consistency* (**DMBC**). When applied to NCSPs with non-isolated solutions, most **DMBC** techniques still produce verbose representations for the boundaries of non-isolated solution sets. Hence, either their applicability is restricted or the tractability limits are rapidly reached.

Recently, a new search strategy, called **UCA6**, was proposed by Silaghi *et al.* [2001]. This search strategy allows enforcing domain reduction techniques on the negation of individual constraints. The difference between the input domains $\mathbf{x}$ and the obtained domains $\mathbf{x}'$ is feasible w.r.t. a considered constraint, C. Hence, splitting $\mathbf{x}$ around $\mathbf{x}'$ allows identifying a *feasible region* $\mathbf{y} := \mathbf{x} \setminus \mathbf{x}'$. The constraint C is redundant in the region $\mathbf{y}$, hence can be eliminated from consideration. The constraints that are still in consideration are called the *running constraints*. In the subproblems generated during the solving process, there may be some variables that do not occur in any running constraint. These variables no longer need to be considered further. In general, the **UCA6** algorithm allows better splitting decisions, hence faster branch-and-prune solution methods. This algorithm is however still slow and provides verbose output in some cases. The first reason is that the orthogonal splitting policy generates a significant number of nearly aligned boxes near the boundaries of constraints. The second reason is that the **UCA6** algorithm often spends too much time in producing too small boxes (near the boundaries of constraints and the boundary of the solution set) w.r.t. a predefined precision ε. This is often unnecessary in many real world applications.

In this chapter, we propose a new search strategy, called **UCA6**$^+$, in order to overcome the above limitations of the **UCA6** algorithm by revising the branching steps of the **UCA6** algorithm. The new contribution in the **UCA6**$^+$ algorithm is twofold. First, the **UCA6**$^+$ algorithm allows better controlling domain reduction operators when the precision is practically detected as sufficient with respect to the requirements of applications. Namely, it prevents unnecessarily spending efforts on reducing the domains of which the sizes are smaller than a predefined precision ε. Moreover, when a certain number of domains are smaller than ε, it resorts to a simple solver that is efficient for small and very low dimensional problems. The gain is then in both the running time and the alignment of boxes. Second, the **UCA6**$^+$ algorithm allows resorting to geometric techniques, such as the *extreme vertex representation* [Aguilera 1998; Bournez *et al.* 1999], to combine aligned boxes resulting in the previous stage into larger equivalent boxes. Therefore, the representation of the solution set is more concise. This hence potentially accelerates the query on the explicit representation of the solution set.

In case the solution set is isolated, the **UCA6**$^+$ algorithm leaves the branching steps unchanged. That is, it uses the bisection as usual. There is a simple heuristic to predict whether solutions are isolated or not. It is simply based on checking if the constraints are equalities. In case the considered problem has inequality constraints, it allows performing domain reduction techniques on the negation of these constraints. In general, the **UCA6**$^+$ algorithm improves search techniques in case the solution set contains continuums of solutions and keeps the same procedure as **DMBC** techniques in case solutions are isolated. In the former case, the **UCA6**$^+$ algorithm provides inner and outer approximations w.r.t. a predefined precision ε. Moreover, a large percentage of provided approximations are often proved to be sound solutions (see Definition 2.47). Our experiments show that the new search technique improves the efficiency as well as the conciseness of the solution representation.

4.2. Motivation

We start by giving two small introductory examples to illustrate the inadequacy of the point-wise approach when blindly applied to NCSPs with non-isolated solutions.

Example 4.1. The first example illustrates how the point-wise approach is misused when applied to NCSPs with non-isolated solutions. Since point-wise techniques inherently assume the isolation of solutions, the interval splitting process they use for branching steps is sometimes prematurely stopped, as soon as a solution is detected within a box. This leads to poorly informative approximations of the solution sets, as shown in the following example. The first example, **WP**, is a two-dimensional simplification of the design model for a kinematic pair consisting of a wheel and a pawl. The constraints determine the regions where the pawl can touch the wheel without blocking its motion.

$$\mathbf{WP} \equiv \langle 20 < \sqrt{x^2+y^2} < 50,\ \frac{12y}{\sqrt{(x-12)^2+y^2}} < 10;\ x \in [-50,50],\ y \in [0,50]\rangle.$$

Figure 4.2 shows the exact solution set (the grey region) and the outcome of a point-wise constraint solver (six large boxes) when the existence of point-wise solutions is abusively assumed. The outcome was computed by using a simple combination of `IloGenerateBounds` and `IloSplit` in ILOG Solver 5.3 in the recommended way [ILOG 2003]:

```
Solver.setDefaultPrecision(P);
Solver.extract(WP);
```

```
Solver.startNewSearch(IloGenerateBounds(Env, X, P) && IloSplit(Env, X));
while (Solver.next()){
    IlcFloatVarArray S = Solver.getFloatVarArray(X);
    reportSolution(S);
}
Solver.endSearch();
```

where

- `WP` is a C^{++} object of the type `IloModel`, which is added with the problem **WP**;
- `Solver` is a C^{++} object of the type `IloSolver`;
- `Env` is a C^{++} object of the type `IloEnv`;
- `X` is a C^{++} object of the type `IloNumVarArray`, which is the vector of the variables x, y;
- `P` is the predefined precision, which is set to the value 2. ♣

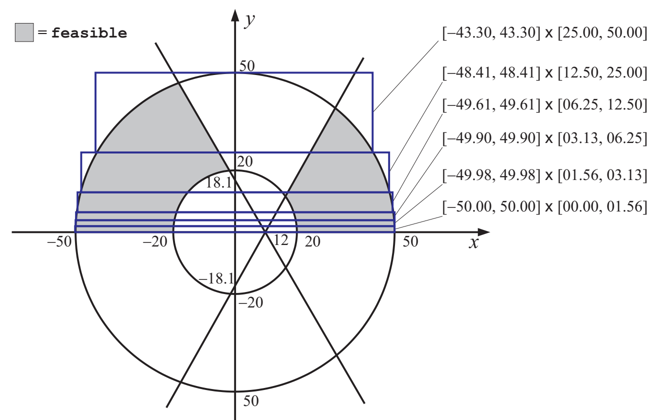

Figure 4.2. An example of a poorly informative representation: the solution set of the problem **WP** is poorly covered by six boxes, when the precision `P` is set to the value 2.

Example 4.2. The second example to be considered is an NCSP that has four nonlinear inequality constraints involving three variables:

$$\mathbf{P3} \equiv \begin{cases} x^2 \leq y; \\ \ln y + 1 \geq z; \\ xz \leq 1; \\ x^{3/2} + \ln(1.5z + 1) \leq y + 1; \\ x \in [0, 15];\ \ y \in [1, 200];\ \ z \in [0, 10]. \end{cases}$$

The solving process in Example 4.1 fails to solve the problem **P3** in reasonable time. Using an efficient implementation of classical bisection techniques,[1] the solving process does not terminate after ten hours and produces more than 260000 small boxes (see Section 4.5). Our technique, the **UCA6**$^+$ algorithm, reduces the complete output to 1376 boxes and terminates after 1.41 seconds on the same computer (see Table 4.1 and Table 4.2). ♣

The result described in Example 4.2 is one of our most successful results and hence does not objectively illustrate the power of our technique. However, it clearly illustrates how the point-wise approach can be inadequate to solve NCSPs with continuums of solutions.

[1] The pruning technique is based on `IloGenerateBounds` in ILOG Solver 5.3, but the branching technique is rewritten to reflect the ideas of **DMBC** techniques.

4.3. Representation of Non-isolated Solutions

4.3.1. Inner and Outer Approximations

In case the solution set is empty or consists of isolated points, its representation is usually simple. The representation of the solution set of a numerical CSP (NCSP) is not simple in other cases, especially when the solution set contains continuums of solutions. The solution set of an NCSP is, in general, a relation on $\mathbb{R}^n$, where n is the number of variables in the considered NCSP. A relation can be theoretically approximated by a superset and/or subset.

Definition 4.3 (Inner Approximation). Given a relation, $S \subseteq \mathbb{R}^n$, a set $S^- \subseteq \mathbb{R}^n$ is called an *inner approximation* of S if it is contained in S; that is, $S^- \subseteq S$.

Definition 4.4 (Outer Approximation). Given a relation, $S \subseteq \mathbb{R}^n$, a set $S^+ \subseteq \mathbb{R}^n$ is called an *outer approximation* of S if it contains S; that is, $S^+ \supseteq S$.

When a relation on $\mathbb{R}^n$, such as the solution set of a numerical CSP, is approximated by inner and/or outer approximations. The former is a *complete approximation* (i.e., it contains all solutions), but may contain some points that are not solutions. Conversely, the latter is a *sound approximation* (i.e., it only contains solutions), but may lose some solutions. The concepts of inner and outer approximations are depicted in Figure 4.3. Given an exact representation $\mathcal{R}$ of a relation $S \subseteq \mathbb{R}^n$, we denote by pts($\mathcal{R}$) the set of points in S.

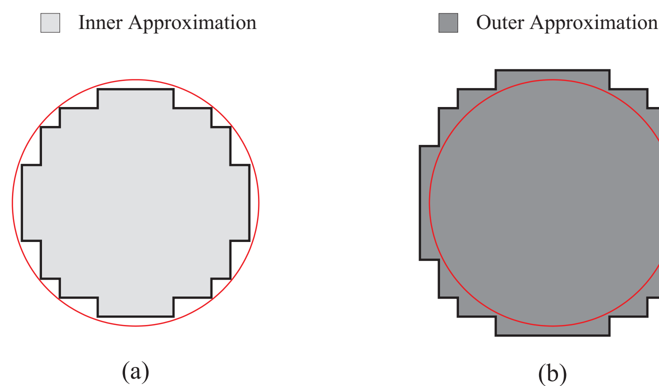

Figure 4.3. An example of inner and outer approximations of a circle with interior: (a) the light grey region is an inner approximation; (b) the dark grey region is an outer approximation.

4.3.2. Union Approximations

A box is the Cartesian product or a vector of real intervals (see Definition 2.78). Because the computational time for querying a point in a box is constant, one often approximates a relation $S \subseteq \mathbb{R}^n$ by a collection of *pairwise disjoint boxes*, where two boxes are said to be *disjoint* if they have no common points. Such a collection is called a *collection of disjoint boxes*, for short. The representation of a collection of disjoint boxes that enumerates these boxes and that stores their coordinates is called the *disjoint box representation* (DBR) [Aguilera 1998].

Among the approximations by boxes, the following three approximations attract the most attention in practice, because of their simplicity. ***The disjointness condition on boxes can be relaxed, depending on specific applications***: the results in this chapter still hold for boxes that may have common points on their facets/boundaries, but not in their interiors.

Definition 4.5 (Inner Union Approximation, $\boxplus^{\mathcal{I}}$). Given a relation $S \subseteq \mathbb{R}^n$. An *inner union approximation* of S, denoted by $\boxplus^{\mathcal{I}}[S]$, is a collection of (disjoint) boxes in $\mathbb{I}^n_\circ$ such that $S \supseteq \mathrm{pts}(\boxplus^{\mathcal{I}}[S])$.

Definition 4.6 (Outer Union Approximation, $\boxplus^{\mathcal{O}}$). Given a relation $S \subseteq \mathbb{R}^n$. An *outer union approximation* of S, denoted by $\boxplus^{\mathcal{O}}[S]$, is a collection of (disjoint) boxes in $\mathbb{I}^n_\circ$ such that $S \subseteq \mathrm{pts}(\boxplus^{\mathcal{O}}[S])$.

Definition 4.7 (Boundary Union Approximation, $\boxplus^{\mathcal{B}}$). Given a relation $S \subseteq \mathbb{R}^n$. A *boundary union approximation*, denoted by $\boxplus^{\mathcal{B}}[S]$, of S (with respect to an inner union approximation $\boxplus^{\mathcal{I}}[S]$ and an outer union approximation $\boxplus^{\mathcal{O}}[S]$) is a collection of (disjoint) boxes in $\mathbb{I}^n_\circ$ such that $\mathrm{pts}(\boxplus^{\mathcal{B}}[S]) = \mathrm{pts}(\boxplus^{\mathcal{O}}[S]) \setminus \mathrm{pts}(\boxplus^{\mathcal{I}}[S])$.

Note 4.8. Note that $\boxplus^{\mathcal{X}}$ is not a function, where $\mathcal{X} \in \{\mathcal{I}, \mathcal{O}, \mathcal{B}\}$. In this thesis, we will always refer to $\boxplus^{\mathcal{B}}[S]$ with respect to some $\boxplus^{\mathcal{I}}[S]$ and some $\boxplus^{\mathcal{O}}[S]$, even not explicitly.

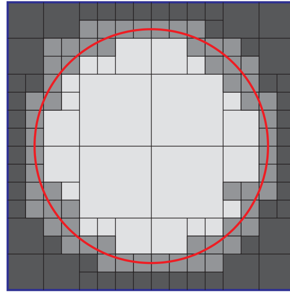

Figure 4.4. An example of inner/outer/boundary union approximations of a circle with interior: the collection of the light grey boxes is an inner union approximation ($\boxplus^{\mathcal{I}}$); the collection of the dark grey boxes is a boundary union approximation ($\boxplus^{\mathcal{B}}$); the collection of the light and dark grey boxes is an outer union approximation ($\boxplus^{\mathcal{O}}$).

The concepts of union approximations are depicted in Figure 4.4. Note that we always have the identity $\mathrm{pts}(\boxplus^{\mathcal{I}}[S]) \cap \mathrm{pts}(\boxplus^{\mathcal{B}}[S]) = \emptyset$. In practice, we often compute $\boxplus^{\mathcal{I}}[S]$ and $\boxplus^{\mathcal{B}}[S]$ first, and then obtain $\boxplus^{\mathcal{O}}[S]$ simply by $\boxplus^{\mathcal{O}}[S] \equiv \boxplus^{\mathcal{I}}[S] \cup \boxplus^{\mathcal{B}}[S]$, which is a *cover* of S (Definition B.4).

The worst-case query time of a *box-tree* in the space $\mathbb{R}^d$ is $\Theta(N^{1-1/d} + k)$, where N is the number of boxes and k is the number of boxes intersecting the *query range* (see [Agarwal *et al.* 2001a]). Therefore, it is useful to construct inner and/or outer union approximations of a relation S in the form of a *bounding-box tree*. That is, the box represented by an arbitrary

node **N** of the tree is contained in the box represented by the parent of **N** and has no common points with the boxes represented by the siblings of **N**. Fortunately, this is an inherent property enjoyed by branch-and-prune methods that use boxes as variable domains.

Several authors have recently addressed the issue of computing inner and/or outer union approximations of the solution set of a factorable NCSP. In [Jaulin 1994], a recursive dichotomous split is performed on variable domains (given as a box). Each box obtained by the split is tested for inclusion by using interval arithmetic (see Section 3.1.6). The obtained boxes are hierarchically structured in the form of a 2^k-*tree*. This technique has shown its practical usefulness in robotics, automation and robust control. Sam-Haroud and Faltings [1996] also proposed a similar technique. However, only binary and ternary constraints, which are obtained by ternarizing the initial NCSP (see Section 2.1.3.5), are considered when performing the split. This means that, for a general factorable NCSP, only *quadtrees* or *octrees* need to be constructed. The approach is restricted to the class of NCSPs with convexity properties. The technique proposed in [Garloff and Graf 1999] constructs outer union approximations algebraically by using Bernstein polynomials, which makes it possible to use algebraic inclusion tests for boxes. The approach is however restricted to polynomial constraints. Collavizza *et al.* [1999] proposed a technique to extend known consistent domains for inequality constraints (e.g., $f(x) \leq 0$) using *associated equalities* ($f(x) = 0$). Unfortunately, their results do not hold for general constraints as claimed in their Proposition 1 and Proposition 2. Most recently, Benhamou and Goualard [2000] devised a method to solve universally quantified constraints by working on the *negation of constraint* (see Section 4.4.2), and Silaghi *et al.* [2001] extended this method to compute inner union approximations of NCSPs with classic numerical constraints.

4.3.3. Qualification of Union Approximations

Most interval-based constraint solvers stop splitting a box, which represents the domains of the currently considered subproblem, as soon as the size of this box is smaller or equal to a given positive precision ε (hence, this box is called an ε-*bounded box*). Some other solvers may attempt to apply a pruning technique or test to ε-bounded boxes before classifying them as *undiscernible*; hence, the name *undiscernible box* has come out.

In general, different constraint solvers use different criteria for leaving ε-bounded boxes unprocessed. If a technique that is applied to ε-bounded boxes before claiming them as undiscernible is used by a solver, then it can be used for the other solvers as well. Therefore, the comparison of solvers should be based on the same criteria of classifying ε-bounded boxes as *undiscernible*. Here, we propose a definition of a checker for this purpose.

Definition 4.9 (Feasibility Checker, `FC`). Consider a sequence X of n real variables. A *feasibility checker* is a function `FC` which takes as input a box $\mathbf{x} \in \mathbb{I}_\circ^n$ and a finite set $\mathcal{C}$ of constraints on a subsequence Y of X, and which returns either `feasible`, `infeasible`, or `unknown` such that:

(*i*) If $\mathtt{FC}(\mathbf{x}, \mathcal{C}) = \mathtt{feasible}$, then every point in $\mathbf{x}$ satisfies all the constraints in $\mathcal{C}$.

(*ii*) If $\mathtt{FC}(\mathbf{x}, \mathcal{C}) = \mathtt{infeasible}$, then no point in $\mathbf{x}$ satisfies all the constraints in $\mathcal{C}$.

(*iii*) If $\mathtt{FC}(\mathbf{x}, \mathcal{C}) = \mathtt{unknown}$, then $\mathtt{FC}(\mathbf{x}', \mathcal{C}) = \mathtt{FC}(\mathbf{x}', \mathcal{C} \cup \mathcal{C}') = \mathtt{unknown}$ holds for every box $\mathbf{x}'$ and every finite set $\mathcal{C}'$ of constraints on Y such that

$$\mathbf{x}[Y] \subseteq \mathbf{x}'[Y] \subseteq \bigcap_{C \in \mathcal{C}'} C.$$

We recall that the notation $\mathbf{x}[Y]$ denotes the projection of $\mathbf{x}$ on the sequence Y of variables (see Definition 2.24). A trivial feasibility checker is the function that always returns `unknown`. It is easy to prove by contradiction way that

- If $\mathtt{FC}(\mathbf{x}, \mathcal{C}) = \mathtt{feasible}$, then $\mathtt{FC}(\mathbf{x}', \mathcal{C}) = \mathtt{FC}(\mathbf{x}', \mathcal{C} \cup \mathcal{C}') = \mathtt{feasible}$ holds for every finite set $\mathcal{C}'$ of constraints on Y and every nonempty box $\mathbf{x}'$ such that $\mathbf{x}'[Y] \subseteq \mathbf{x}[Y] \subseteq \bigcap_{C \in \mathcal{C}'} C$;

- If $\mathtt{FC}(\mathbf{x}, \mathcal{C}) = \mathtt{infeasible}$, then $\mathtt{FC}(\mathbf{x}', \mathcal{C}) = \mathtt{FC}(\mathbf{x}', \mathcal{C} \cup \mathcal{C}') = \mathtt{infeasible}$ holds for every finite set $\mathcal{C}'$ of constraints on Y and every box $\mathbf{x}'$ such that $\mathbf{x}'[Y] \subseteq \mathbf{x}[Y] \subseteq \bigcap_{C \in \mathcal{C}'} C$.

Therefore, we say that a feasibility checker is monotonic. A feasibility checker is very similar to an inclusion test (see Section 3.1.6), except that it enjoys the monotonicity.

Definition 4.10 (Interval-Based Precision). Given an NCSP $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, a precision (vector) ε, and a feasibility checker `FC`. A solution algorithm that computes inner and boundary union approximations is said to be *of the precision* ε *w.r.t. the feasibility checker* `FC` if the boundary union approximation equals (w.r.t. the set union) to a collection $\mathcal{U}$ of disjoint ε-bounded boxes in $\mathbb{I}^n_\circ$ such that

$$\forall \mathbf{x} \in \mathcal{U} : \mathtt{FC}(\mathbf{x}, \mathcal{C}) = \mathtt{unknown}. \tag{4.1}$$

4.4. Exhaustive Search for CSPs with Non-isolated Solutions

Essentially, recent interval-based search techniques for NCSPs are of the form of *dichotomous maintaining bounds by consistency* (**DMBC**). However, **DMBC** techniques often generates verbose inner and outer union approximations. The first reason is that the orthogonal splitting policy they use often generates a significant number of nearly aligned boxes near the boundaries of constraints. The second reason is that entirely feasible boxes might be unnecessarily split. The improvements to this search strategy (i.e., **DMBC**) we propose are presented in the sections 4.4.4, 4.4.5, and 4.4.6. They require, however, to generalize or modify several previously existing concepts. These modifications will be presented in the next three subsections.

4.4.1. Domain Reduction Operators

First, we define the concept of a domain reduction operator as follows.

Definition 4.11 (Domain Reduction Operator, `DR`). Given a sequence X of n real variables associated with domains $\mathcal{D}$. A *domain reduction operator* `DR` for numerical constraints is a function which takes as input a box $\mathbf{x} \in \mathbb{I}^n_\circ$ contained in $\mathcal{D}$ and a finite set $\mathcal{C}$ of constraints on X, and which returns a box in $\mathbb{I}^n_\circ$, denoted by $\mathtt{DR}(\mathbf{x}, \mathcal{C})$, satisfying the following properties:

$$\text{(Contractiveness)} \qquad \mathtt{DR}(\mathbf{x}, \mathcal{C}) \subseteq \mathbf{x}, \tag{4.2}$$

$$\text{(Correctness)} \qquad \mathtt{DR}(\mathbf{x}, \mathcal{C}) \supseteq \mathbf{x} \cap \bigcap_{C \in \mathcal{C}} C. \tag{4.3}$$

A domain reduction operator has also been referred to as a narrowing operator, contracting operator, or contractor in literature. We eventually adopt the name *domain reduction operator*, because it is mnemonic and the terminology *domain reduction* has been widely accepted in many fields, not only in contraint programming.

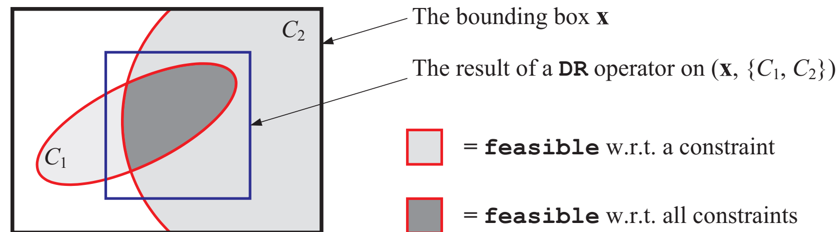

Figure 4.5. A domain reduction (DR) operator applied to a box $\mathbf{x}$ and a constraint set $\{C_1, C_2\}$.

In constraint programming, domain reduction operators are usually constructed by enforcing box consistency, hull consistency, or kB-consistency (see Section 2.3.3 and Section 3.2.2). These particular domain reduction operators enjoy the monotonicity (see Definition 2.44 and Definition 4.18). Many domain reduction operators do not enjoy the monotonicity, but they are still very efficient in practice. The concept of a domain reduction operator is depicted in Figure 4.5. Other examples of domain reduction operators are depicted in Figure 4.7. The following property of domain reduction operators is obvious but important for constraint solving.

Theorem 4.12. Given a set $\mathcal{C}$ of constraints on a sequence of n real variables associated with domains $\mathcal{D}$. Suppose $\mathbf{x} \in \mathbb{I}_{\circ}^{n}$ is a box contained in $\mathcal{D}$. If there exists a domain reduction operator DR that maps $(\mathbf{x}, \mathcal{C})$ to an empty set (i.e., $\mathtt{DR}(\mathbf{x}, \mathcal{C}) = \emptyset$), then $\mathcal{C}$ is inconsistent in $\mathbf{x}$; that is,

$$\mathbf{x} \subseteq \neg\mathcal{C} \equiv \mathcal{D} \setminus \bigcap_{C \in \mathcal{C}} C. \tag{4.4}$$

Proof. It follows from the correctness of domain reduction operators, as defined in Definition 4.11, that $\mathbf{x} \cap \bigcap_{C \in \mathcal{C}} C = \emptyset$. Hence, we have $\mathbf{x} \subseteq \neg\mathcal{C}$. ■

4.4.2. Complementary Boxing Operators

We now consolidate the technique for constructing inner approximations of universally quantified constraints [Benhamou and Goualard 2000] and the technique for constructing outer approximations of numerical constraints [Silaghi *et al.* 2001] with a new notation that facilitates the presentation of our techniques in Section 4.4.4.

Let X be a sequence of n real variables associated with domains $\mathcal{D}$ and $\mathcal{C}$ a finite set of constraints on X. Consider a box $\mathbf{x} \subseteq \mathcal{D}$. A *negation test* amounts to performing a kind of domain reduction operator on $(\mathbf{x}, \neg\mathcal{C})$ to get a box $\mathbf{x}'$. If $\mathbf{x}'$ is an empty set, then every point in $\mathbf{x}$ satisfies $\mathcal{C}$ (see Theorem 4.12); otherwise, use a kind of splitting operator, proposed in [Benhamou and Goualard 2000] under the name **ICAb3**$_c$, to split $\mathbf{x}$ around $\mathbf{x}'$ and then dichotomize $\mathbf{x}'$. A negation-based search algorithm, called **ICAb5**, recursively performs **ICAb3**$_c$ on a selected *running constraint*[2] until a predefined interval-based precision is reached. This

[2] A running constraint is a constraint that is currently still under consideration.

procedure was originally proposed for universally quantified constraints of the form $\forall t : C(t, x)$ by Benhamou and Goualard [2000], and was later adapted to solving numerical constraints by Silaghi *et al.* [2001]. In [Silaghi *et al.* 2001], an operator, called $\mathbf{B_q}$ therein, similar to a negation test is applied to numerical inequality constraints. The search algorithm in [Silaghi *et al.* 2001], called **UCA6**, employs the $\mathbf{B_q}$ operator to enclose the negations of all individual constraints and then choose the best result to guide the domain splitting during search. In addition, the **UCA6** algorithm memorizes old $\mathbf{B_q}$'s for reuse when computing new $\mathbf{B_q}$'s. It also uses some heuristics to choose inequalities for the $\mathbf{B_q}$ operator and domain splitting.

For convenience, we define a kind of operator to generalize the concept of a negation test, and then give several important properties. The generalized operator is called the *complementary boxing operator*, and the corresponding splitting operator is called the *box splitting operator*. Hereafter is the definition of a complementary boxing operator.

Definition 4.13 (Complementary Boxing Operator, CB). Given a sequence X of n real variables associated with domains $\mathcal{D}$. A *complementary boxing operator* is a function $\mathtt{CB}$ that takes as input a box $\mathbf{x} \in \mathbb{I}_\circ^n$ contained in $\mathcal{D}$ and a finite set $\mathcal{C}$ of constraints on X, and that returns a box in $\mathbb{I}_\circ^n$, denoted by $\mathtt{CB}(\mathbf{x}, \mathcal{C})$, satisfying the following properties:

$$\text{(Contractiveness)} \qquad \mathtt{CB}(\mathbf{x}, \mathcal{C}) \subseteq \mathbf{x}, \tag{4.5}$$

$$\text{(Complementariness)} \qquad \mathbf{x} \setminus \mathtt{CB}(\mathbf{x}, \mathcal{C}) \subseteq \bigcap_{C \in \mathcal{C}} C. \tag{4.6}$$

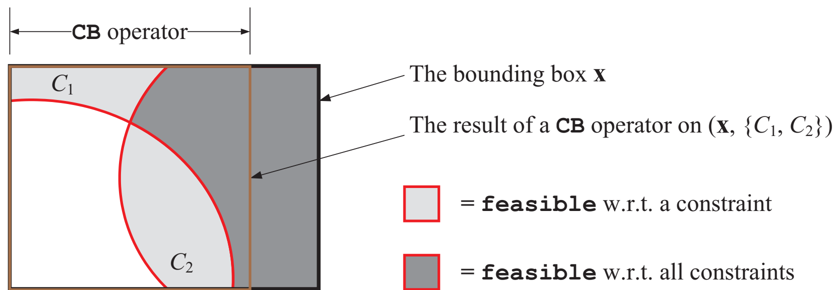

Figure 4.6. An example of a complementary boxing ($\mathtt{CB}$) operator applied to a box $\mathbf{x}$ and a set $\{C_1, C_2\}$ of two constraints.

A box resulting from the application of a complementary boxing operator to a bounding box $\mathbf{x}$ and a set $\mathcal{C}$ of constraints is called a *complementary box* of $\mathcal{C}$ within $\mathbf{x}$. The term *complementary boxing* refers to the process of computing a complementary box. The concept of a complementary boxing operator is depicted in Figure 4.6. Additionally, Figure 4.7 illustrates the behavior of domain reduction operators and complementary boxing operators when applied to the same bounding boxes, in some typical situations.

The complementariness of complementary boxing operators means that complementary boxing makes it possible to isolate certain regions, namely $\mathbf{x} \setminus \mathtt{CB}(\mathbf{x}, \mathcal{C})$, of which the points entirely satisfy all the constraints in $\mathcal{C}$. Especially, if the application of a complementary boxing operator to a box and a constraint results in an empty set, then the box completely satisfies that constraint. Similarly, if the application of a complementary boxing operator to a box with

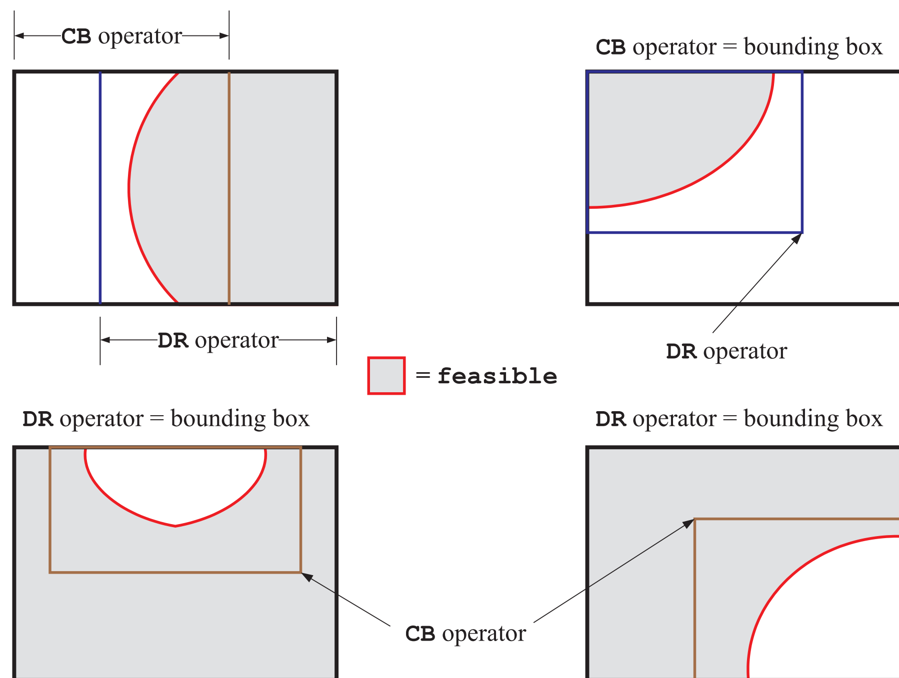

Figure 4.7. Examples of domain reduction (DR) operators and complementary boxing (CB) operators.

the whole set of constraints results in an empty set, then the box is completely feasible. The following theorem states that property formally.

Theorem 4.14. Given a set $\mathcal{C}$ of constraints on a sequence of n real variables associated with domains $\mathcal{D}$. Suppose $\mathbf{x}$ is a box contained in $\mathcal{D}$. If there exists a complementary boxing operator CB that maps $(\mathbf{x}, \mathcal{C})$ to an empty set (i.e., $\mathtt{CB}(\mathbf{x}, \mathcal{C}) = \emptyset$), then $\mathcal{C}$ is satisfied with every point in $\mathbf{x}$; that is, $\mathbf{x} \subseteq \bigcap_{C \in \mathcal{C}} C$.

An complementary boxing operator can be constructed from a domain reduction operator as stated in the following theorem.

Theorem 4.15. Given a domain reduction operator DR. The function f defined as $f(\mathbf{x}, \mathcal{C}) \equiv \mathtt{DR}(\mathbf{x}, \neg\mathcal{C})$ is a complementary boxing operator.

Proof. By definition $\mathbf{x}_f = f(\mathbf{x}, \mathcal{C}) = \mathtt{DR}(\mathbf{x}, \neg\mathcal{C})$. The contractiveness of domain reduction operators implies that $\mathbf{x}_f \subseteq \mathbf{x}$. That is, f enjoys the contractiveness of complementary boxing operators. In addition to that, the correctness of domain reduction operators implies that

$$\mathbf{x} \cap \neg\mathcal{C} \subseteq \mathbf{x}_f \tag{4.7}$$

It follows from (4.7) that, for all $x \in \mathbf{x} \setminus \mathbf{x}_f$, we have $x \notin \mathbf{x} \cap \neg\mathcal{C}$, thus, $x \notin \neg\mathcal{C} \equiv \mathcal{D} \setminus \bigcap_{C \in \mathcal{C}} C$, and $x \in \bigcap_{C \in \mathcal{C}} C$ because $x \in \mathbf{x} \subseteq \mathcal{D}$. That is, we have $\mathbf{x} \setminus \mathbf{x}_f \subseteq \bigcap_{C \in \mathcal{C}} C$. This means that f enjoys the complementariness of complementary boxing operators. ∎

By a similar argument, we also have the following result.

Theorem 4.16. Given a complementary boxing operator CB. The function f defined as $f(\mathbf{x}, \mathcal{C}) \equiv \texttt{CB}(\mathbf{x}, \neg\mathcal{C})$ is a domain reduction operator.

Proof. By definition $\mathbf{x}_f = f(\mathbf{x}, \mathcal{C}) = \texttt{CB}(\mathbf{x}, \neg\mathcal{C})$. The contractiveness of complementary boxing operators implies that $\mathbf{x}_f \subseteq \mathbf{x}$; that is, f has the contractiveness of domain reduction operators. In addition to that, the complementariness of complementary boxing operators implies that

$$\mathbf{x} \setminus \mathbf{x}_f \subseteq \neg\mathcal{C} = \mathcal{D} \setminus \bigcap_{C \in \mathcal{C}} C \tag{4.8}$$

It follows from (4.8) that, for all $x \in \mathbf{x}_f$, we have $x \notin \mathbf{x} \cap \neg\mathcal{C}$, thus, $x \notin \mathcal{D} \setminus \bigcap_{C \in \mathcal{C}} C$ and $x \in \bigcap_{C \in \mathcal{C}} C$ because $x \in \mathbf{x}_f \subseteq \mathcal{D}$. That is, we have $\mathbf{x} \setminus \mathbf{x}_f \subseteq \mathcal{C}$. This means that f enjoys the complementariness of complementary boxing operators. ■

It follows from Theorem 4.15 and Theorem 4.16 that complementary boxing operators can be constructed from domain reduction operators and vice versa. Next, we refine the general definition of the monotonicity of domain reduction operators and complementary boxing operators. In particular, let $\mathcal{C} = \{C_1, \ldots, C_k\}$ be set of k constraints. A complementary boxing operator can be constructed by $\texttt{CB}(\mathbf{x}, \mathcal{C}) := \texttt{DR}(\mathbf{x}, \neg\mathcal{C}) = \texttt{DR}(\mathbf{x}, C_0)$, where C_0 is the disjunction of constraints $\neg C_1, \ldots, \neg C_k$. In a system that does not accept disjunctive constraints, we can relax it by taking the union of complementary boxes, as stated in following theorem.

Theorem 4.17. Consider a sequence X of n real variables associated with domains $\mathcal{D}$. Let $\mathcal{C} = \{C_1, \ldots, C_k\}$ be a set of constraints on X and $\{\texttt{DR}_1, \ldots, \texttt{DR}_k\}$ a set of domain reduction operators for X. Suppose $\{C'_1, \ldots, C'_k\}$ is a set of constraints on X such that $\neg C_i \equiv \mathcal{D} \setminus C_i \subseteq C'_i$ for all $i = 1, \ldots, k$. Then the operator defined by the following rule is a complementary boxing operator:

$$\forall \mathbf{x} \in \mathbb{I}^n_\circ, \mathbf{x} \subseteq \mathcal{D} : f(\mathbf{x}, \mathcal{C}) \equiv \Box \bigcup_{i=1}^{k} \texttt{DR}_i(\mathbf{x}, C'_i), \tag{4.9}$$

Proof. The contractiveness of f is obvious because $\texttt{DR}_i(\mathbf{x}, C'_k) \subseteq \mathbf{x}$. We now prove the complementariness. For every $x \in \mathbf{x} \setminus \bigcup_{i=1}^{k} \texttt{DR}_i(\mathbf{x}, C'_i)$ and every $i \in \{1, \ldots, k\}$, we have

$$\begin{aligned} & x \notin \texttt{DR}_i(\mathbf{x}, C'_i) \supseteq \mathbf{x} \cap C'_i \\ \Rightarrow\ & x \notin \mathbf{x} \cap C'_i \\ \Rightarrow\ & x \notin C'_i \supseteq \neg C_i \\ \Rightarrow\ & x \notin \mathcal{D} \setminus C_i \\ \Rightarrow\ & x \in C_i \qquad \text{(since } x \in \mathcal{D}\text{),} \end{aligned}$$

hence, $x \in \bigcap_{i=1}^{k} C_i$. Therefore, we have

$$\mathbf{x} \setminus f(\mathbf{x}, \mathcal{C}) \subseteq \mathbf{x} \setminus \bigcup_{i=1}^{k} \texttt{DR}_i(\mathbf{x}, C'_i) \subseteq \bigcap_{i=1}^{k} C_i.$$

This is the complementariness as required. ■

The negation $\neg C$ of a numerical constraint C of the form $f(x) \diamond 0$; where $\diamond$ is either $\leq$, $<$, $\geq$, $>$, $=$, or $\neq$; is the constraint $f(x) \,\tilde{\diamond}\, 0$ with $\tilde{\diamond}$ is either $>$, $\geq$, $<$, $\leq$, $\neq$, or $=$, respectively. In practice, some implementations of domain reduction operators only accept constraints that are defined with the relations $\leq$ and $\geq$, but not with the relations $<$ and $>$. For example, a constraint C_i of the form $C_i \equiv (f(x) \leq 0)$ has the negation of the form $\neg C_i \equiv (f(x) > 0)$, which is not accepted in some implementations. Fortunately, we can safely use $C'_i \equiv (f(x) \geq 0)$ in the complementary boxing operator defined by (4.9) because $\neg C_i \subseteq C'_i$ holds.

Definition 4.18 (Monotonicity). Given a sequence X of variables associated with domains $\mathcal{D}$. A domain reduction (or complementary boxing) operator μ is called *monotonic* if for every set $\mathcal{C}$ of constraints on X we have

$$\forall \mathbf{x}, \mathbf{x}' \in \mathbb{I}_\circ^n, \mathbf{x} \subseteq \mathcal{D}, \mathbf{x}' \subseteq \mathcal{D} : \mathbf{x} \subseteq \mathbf{x}' \Rightarrow \mu(\mathbf{x}, \mathcal{C}) \subseteq \mu(\mathbf{x}', \mathcal{C}). \tag{4.10}$$

The following theorem gives a way to construct a feasibility checker from domain reduction operators and complementary boxing operators that enjoys the monotonicity.

Theorem 4.19 (FC from DR and CB). Given a set $\{\mathtt{DR}_1, \ldots, \mathtt{DR}_n\}$ of n domain reduction operators and a set $\{\mathtt{CB}_1, \ldots, \mathtt{CB}_n\}$ of n complementary boxing operators, where $\mathtt{DR}_k$ and $\mathtt{CB}_k$ are defined in k-dimensional space ($k = 1, \ldots, n$). Assume that all the operators $\mathtt{DR}_k$ and $\mathtt{CB}_k$ are monotonic. Let X be a sequence of n real variables. Suppose f is a function which takes as input a box $\mathbf{x} \in \mathbb{I}_\circ^n$ and a finite set $\mathcal{C}$ of constraints on a subsequence Y of size k of X, and which returns either `feasible`, `infeasible`, or `unknown` such that

$$f(\mathbf{x}, \mathcal{C}) = \mathtt{infeasible} \quad \Leftrightarrow \quad \mathtt{DR}_k(\mathbf{x}[Y], \mathcal{C}) = \emptyset, \tag{4.11}$$

$$f(\mathbf{x}, \mathcal{C}) = \mathtt{feasible} \quad \Leftrightarrow \quad \mathtt{CB}_k(\mathbf{x}[Y], \mathcal{C}) = \emptyset. \tag{4.12}$$

Then f is a feasibility checker.

Proof. The result follows Definition 4.9, Theorem 4.12, and Theorem 4.14 easily. ■

4.4.3. Domain Splitting Operators

First, we recall the concept of a bisection, where a side (i.e., an interval) of a box is dichotomized into two equal parts.

Definition 4.20 (Dichotomous Splitting Operator, DS). A *dichotomous splitting* (DS) operator is a function $\mathtt{DS} : \mathbb{I}_\circ^n \rightarrow 2^{\mathbb{I}_\circ^n}$ that takes as input a box in $\mathbb{I}_\circ^n$, and that returns two (disjoint) boxes in $\mathbb{I}_\circ^n$ resulting from splitting a side of the input box into two halves.

Example 4.21. Consider a box $\mathbf{x} \equiv (\mathbf{x}_1, \ldots, \mathbf{x}_n)^{\mathrm{T}} \in \mathbb{I}_\circ^n$, where $\mathbf{x}_i = [\underline{x}_i, \overline{x}_i[$. A dichotomous splitting operator DS may split $\mathbf{x}$ into two disjoint boxes: $\mathbf{x}' = (\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, \mathbf{x}'_i, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_n)^{\mathrm{T}}$ and $\mathbf{x}'' = (\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, \mathbf{x}''_i, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_n)^{\mathrm{T}}$, where $\mathbf{x}'_i = [\underline{x}_i, \mathrm{mid}(\mathbf{x})[$ and $\mathbf{x}''_i = [\mathrm{mid}(\mathbf{x}), \overline{x}_i[$. Note that $\mathbf{x}'_i \cap \mathbf{x}''_i = \emptyset$, thus, $\mathbf{x}' \cap \mathbf{x}'' = \emptyset$. ♣

Second, we define the concept of a box splitting operator, which splits around a complementary box in order to isolate *feasible regions* w.r.t. a set of constraints.

Definition 4.22 (Box Splitting Operator, BS). A *box splitting* (BS) operator is a function $\mathtt{BS} : \mathbb{I}^n_\circ \times \mathbb{I}^n_\circ \rightarrow 2^{\mathbb{I}^n_\circ}$ which takes as input two boxes such that the former contains the latter, and which sequentially splits the outer box along some facets of the inner one. The output is a sequence of disjoint boxes.

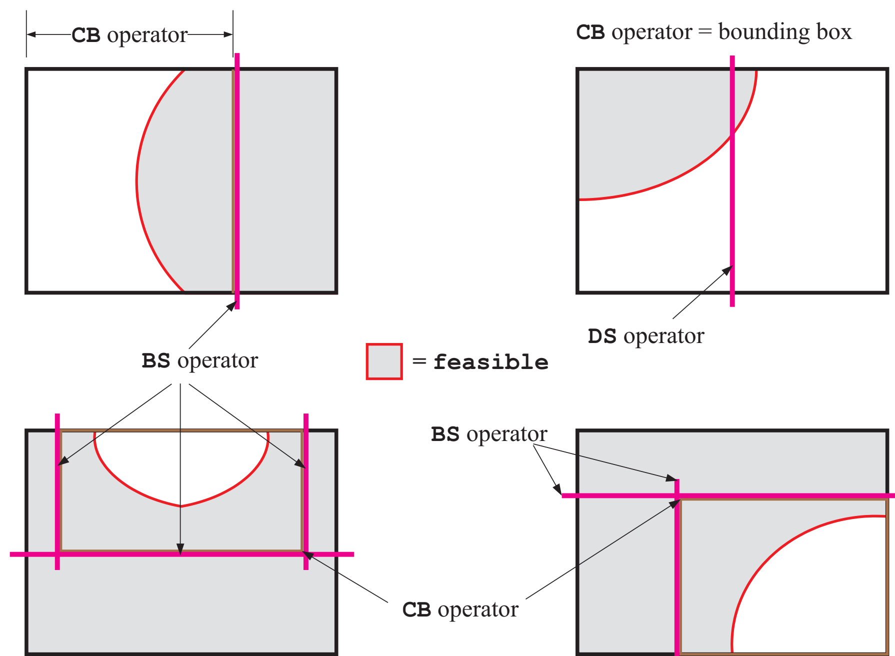

Figure 4.8. Examples of box splitting (BS) and dichotomous splitting (DS) operators. In box splitting, all boxes excepted the complementary box are feasible w.r.t. the considered constraints.

In fact, the concept of a box splitting operator is a slight generalization of the splitting operator proposed in [Van Iwaarden 1996]. The original splitting operator gives a way to split a region surrounding a box, provided that this box contains at most one optimal solution to a considered optimization problem.

In our algorithm, a box splitting operator which takes as input a domain box and a complementary box resulting from the application of a complementary boxing operator is applied in combination with a dichotomous splitting operator. The dichotomous splitting operator is used when either the complementary boxing operator produces no reduction or the box splitting operator results in too small boxes. Figure 4.8 illustrates the concept of a box splitting operator used for this purpose.

4.4.4. Controlling the Reduction of Small Domains

We first observe that a better alignment of boxes near boundary regions of the solution set can be obtained by finely controlling the application of domain reduction operators during search. In particular, if the i-th component $\mathbf{x}_i$ of a box $\mathbf{x} \in \mathbb{I}^n_\circ$, which represents the vector of domains

of the considered subproblem, is bounded by ε_i, then the domain reduction to be applied to the corresponding variable x_i is not necessary to be enabled. This is to obtain better alignments of contiguous boxes and the computational performance. Here is the formal definition of the condition for the context.

Definition 4.23 (Active/Inactive Variable). Given a sequence X of n real variables $(x_1, \ldots, x_n)$, and a vector $\varepsilon = (\varepsilon_1, \ldots, \varepsilon_n)^{\mathrm{T}} \in \mathbb{R}^n_+$. Consider a box $\mathbf{x} = (\mathbf{x}, \ldots, \mathbf{x}_n)^{\mathrm{T}} \in \mathbb{I}^n_\circ$ and a finite set $\mathcal{C}$ of constraints on subsequences of X. The variable x_i is called an *active variable in* $\mathbf{x}$ *w.r.t.* $\mathcal{C}$ *and* ε if it is a variable of (in other words, it occurs in) at least one constraint in $\mathcal{C}$ and $\mathrm{w}(\mathbf{x}_i) \equiv \sup(\mathbf{x}_i) - \inf(\mathbf{x}_i) > \varepsilon_i$ holds. The variable x_i is called an *inactive variable in* $\mathbf{x}$ *w.r.t.* $\mathcal{C}$ *and* ε if it is not an active variable in $\mathbf{x}$ w.r.t. $\mathcal{C}$ and ε.

A domain reduction operator (respectively, a complementary boxing operator) that only works on active variables is called a *restricted-dimensional domain reduction operator* (respectively, a *restricted-dimensional complementary boxing operator*). In other words, a restricted-dimensional operator stops reducing the domains of inactive variables in its implementation. We denote by $\mathtt{DR}_{\mathtt{rd}}$ (respectively, $\mathtt{CB}_{\mathtt{rd}}$) the restricted-dimensional domain reduction operator (respectively, the restricted-dimensional complementary boxing operator) obtained from the normal domain reduction operator `DR` (respectively, the normal complementary boxing operator `CB`) by enforcing the domain reduction only on active variables.

When local consistency notions are enforced in order to obtain the effect of domain reduction (i.e., they play the role of domain reduction operators), restricted-dimensional operators amount to enforcing the local consistency notions only for active variables. The recent algorithms for achieving local consistency (such as box consistency, hull consistency, and kB-consistency) in the sections 3.2.2.1, 3.2.2.2, 3.2.2.3, 3.2.2.4, and 3.2.2.5 can be easily modified to adopt the idea of only working on active variables (by ignoring any procedure involving inactive variables). In case a domain reduction operator cannot be modified to adopt the idea of working only on active variables, we can apply it normally and then restore the domains of inactive variables to the input domains/intervals. In this case, the gain is only in the (better) alignment of contiguous boxes, but not in the performance.

Fortunately, the implementation of box consistency in a well-known product, ILOG Solver [ILOG 2003], supports the idea of working only on active variables. It can be done by simply passing only active variables (`X`) to the function `IloGenerateBounds` when we need to generate narrower bounds on `X`.

An illustration of the difference between the effect of a normal domain reduction (`DR`) operator and the effect of a restricted-dimensional domain reduction ($\mathtt{DR}_{\mathtt{rd}}$) operator in the solving process is given in Figure 4.9. In this example, although the normal `DR` operator produces more accurate output than the $\mathtt{DR}_{\mathtt{rd}}$ operator does, it has to spend much time on making the boundary region narrower than the allowed tolerance ε_1. This is, in practice, unnecessary since real world applications mainly focus on the inner boxes, the boxes near the boundary are often unsafe for the further exploration in the applications, as shown in our experiments (see Section 4.5). Moreover, the number boxes (15 inner boxes + 8 undiscernible boxes) resulting from the application of the `DR` operator is higher than the number boxes (3 inner boxes + 8 undiscernible boxes) resulting from the application of the $\mathtt{DR}_{\mathtt{rd}}$ operator, in general. Moreover, the contiguous boxes obtained by using the $\mathtt{DR}_{\mathtt{rd}}$ operator are often aligned; hence, a geometrically compacting technique can work on them efficiently to get a concise representation of the solution set, as shown in the next section.

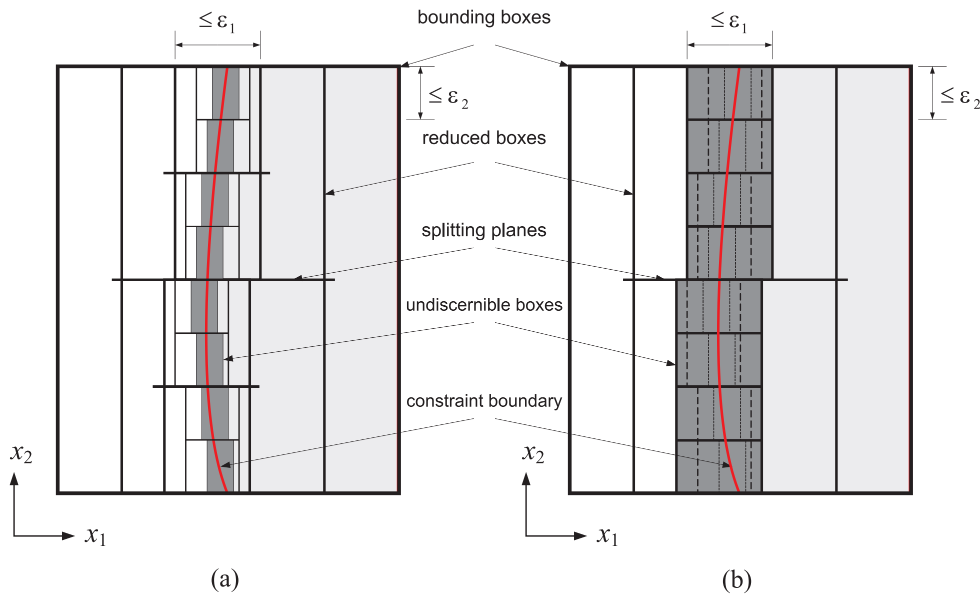

Figure 4.9. An example of normal domain reductions and restricted-dimensional domain reductions at different levels: (a) all variables (x_1 and x_2) are considered for the over-reduction; (b) only the active variable (x_2) is considered for the reduction. The light grey regions are inner boxes, the dark grey regions are undiscernible boxes.

4.4.5. Compact Representation of Solutions

Once the effect of better alignments is obtained, the question is how such a set of aligned boxes can be compacted into a smaller set. We propose to use the *extreme vertex representation* (EVR) of orthogonal polyhedra for this purpose. The extreme vertex representation was first proposed by Aguilera and Ayala [1997] for the three-dimensional space, and was later extended to the n-dimensional space in [Bournez and Maler 2000; Bournez *et al.* 1999]. The basic idea is that the union of disjoint boxes delivered by a box covering solver defines an *orthogonal polyhedron* for which an improved representation can be generated. An orthogonal polyhedron can be naturally represented as the union of disjoint boxes (by enumerating the boxes and their vertices). That representation is called the *disjoint box representation* (DBR) in computational geometry. The EVR is a way of compacting the DBR (see [Aguilera 1998] and [Bournez and Maler 2000; Bournez *et al.* 1999]). Roughly speaking, in the EVR, the extreme vertices of an orthogonal polyhedron are identified and stored in a compact manner such that no information is lost w.r.t. the representation of the polyhedra. Moreover, when converting from EVR back to DBR, the obtained DBR are often more compact than the initial DBR.

We now recall some basic concepts in the theory of extreme vertex representation. The reader can find more details in [Bournez and Maler 2000; Bournez *et al.* 1999]. These concepts are sufficient to be presented for *griddy polyhedra*[3] because the results on general orthogonal polyhedra can be easily obtained from the results on griddy polyhedra by mapping the multidimensional array of vertex indices of the orthogonal polyhedra to the multidimensional array of vertices of griddy polyhedra (see Figure 4.10). In fact, they do not depend on the orthogonality of the underlying basis.

[3] A griddy polyhedron is the union of some unit hypercubes with integer-valued vertices [Bournez *et al.* 1999].

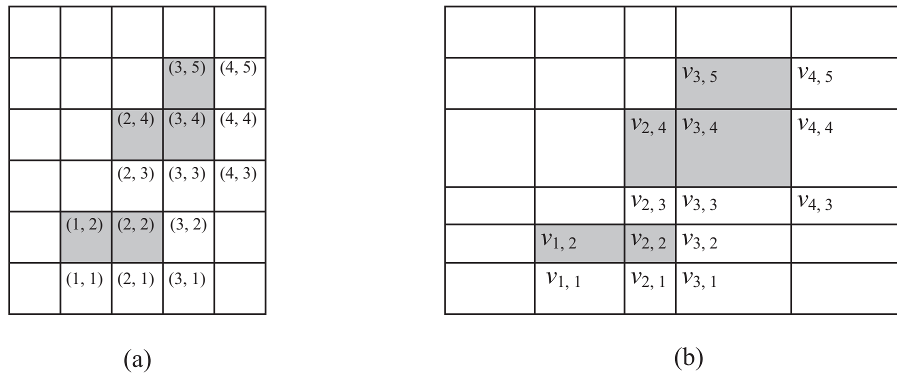

Figure 4.10. Examples of a griddy polyhedron and an orthogonal polyhedron: (a) a griddy polyhedron made of the vertex indices of (b) an orthogonal polyhedron.

For simplicity, polyhedra are assumed to live in $\mathbf{X} = [0, m[^d \subseteq \mathbb{R}^d$ (the results also hold for $\mathbf{X} = \mathbb{R}^d_+$). Let $\mathcal{G} = (0, 1, \ldots, m-1)^d \subseteq \mathbb{N}^d$ be a grid of integer points. For every point $x \in \mathbf{X}$, $\lfloor x \rfloor$ denotes the grid point corresponding to the (componentwise) integer part of x. The unit box associated with a grid point $x = (x_1, \ldots, x_d)^{\mathrm{T}} \in \mathcal{G}$ is the box $\mathbf{B}(x) = [x_1, x_1 + 1[\times \cdots \times [x_d, x_d + 1[$. The set of all unit boxes is denoted by $\mathcal{B}$. A griddy polyhedron P is the set closure of the union of some unit boxes, or can be viewed as a set of grid points.

Definition 4.24 (Color Function). Let P be a griddy polyhedron. The *color function* $\mathsf{color} : \mathbf{X} \to \{0, 1\}$ is defined as follows: if x is a grid point then $\mathsf{color}(x) = 1 \Leftrightarrow \mathbf{B}(x) \subseteq P$; otherwise, $\mathsf{color}(x) = \mathsf{color}(\lfloor x \rfloor)$.

We say that a grid point x is black (respectively, white) and that $\mathbf{B}(x)$ is full (respectively, empty) when $\mathsf{color}(x) = 1$ (respectively, $\mathsf{color}(x) = 0$). Figure 4.11a illustrates the color function for griddy polyhedra. In Figure 4.11b, the concept of a *forward cone* based at $x \in \mathcal{G}$ is also depicted: $x^{\triangleleft} \equiv \{y \in \mathcal{G} \mid x \leq y\}$.

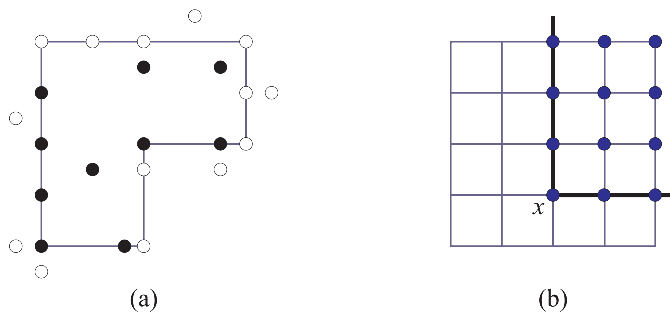

Figure 4.11. A griddy polyhedron: (a) sample colors defined by the color function; (b) the forward cone $x^{\triangleleft} \equiv \{y \in \mathcal{G} \mid x \leq y\}$.

A *canonical representation* scheme for $2^{\mathcal{B}}$ (or $2^{\mathcal{G}}$) is a set $\mathcal{E}$ of syntactic objects such that there is some bijective function $\Psi : \mathcal{E} \to 2^{\mathcal{B}}$; that is, every polyhedron has a unique representation. The most simple representation scheme is to explicitly enumerate the values of the color function on every grid point; hence, it needs a d-dimensional array of bits with m^d

entries. Another simple representation is the vertex representation that consists of the set $\{(x, \mathsf{color}(x)) \mid x \text{ is a vertex}\}$. This is however still verbose. Hence, it is desired to store only important vertices only. The following definition identifies these important vertices.

Definition 4.25 (Extreme Vertex). A grid point x is called an *extreme vertex* of a griddy polyhedron P if the number of black grid points in $\mathcal{N}(x) = \{x_1 - 1, x_1\} \times \cdots \times \{x_d - 1, x_d\}$ is odd (called the *neighborhood* of x).

Let $\oplus$ denotes the *addition modulo 2*, known also as the exclusive-or (XOR) operation in Boolean algebra, $p \oplus q = (p \wedge \neg q) \vee (p \wedge \neg q)$. The $\oplus$ operation on sets is defined by $A \oplus B = \{x \mid (x \in A) \oplus (x \in B)\}$. The set of extreme vertices (together with the $\oplus$ operation) makes an canonical representation of griddy polyhedra as follows [Bournez and Maler 2000].

Theorem 4.26 (Extreme Vertex Representation). For any griddy polyhedron P, there is a unique set V of grid points in $\mathcal{G}$ such that $P = \bigoplus_{x \in V} x^{\triangleleft}$. Moreover, the set V is the set of all extreme vertices of P and this is a canonical representation.

Proof. See the proofs of Theorem 1 and Theorem 2 in [Bournez and Maler 2000]. ■

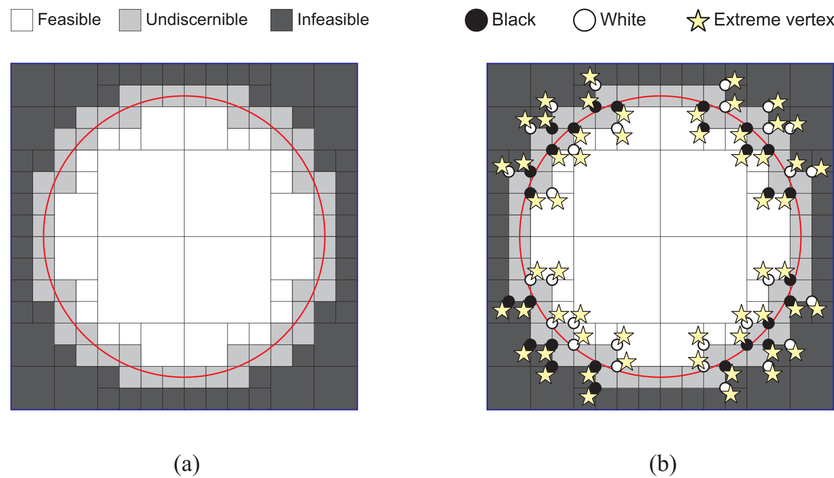

Figure 4.12. An example of (a) DBR and (b) extreme vertices of union approximations

Figure 4.12 illustrates the concept of EVR applied to union approximations. Theorem 4.26 shows that any griddy polyhedron can be canonically represented by the set of its extreme vertices (and their colors). The extreme vertex representation improves the space required for storing orthogonal polyhedra by an order of magnitude [Aguilera 1998; Bournez and Maler 2000; Bournez *et al.* 1999]. It also enables the design of efficient algorithms for fundamental operations on orthogonal polyhedra, such as the membership query and the set-theoretic operations (see [Aguilera 1998] and [Bournez and Maler 2000; Bournez *et al.* 1999]).

An effective transformation between DBR and EVR was proposed for low dimensional or small-size (i.e., m is small) polyhedra [Aguilera 1998; Bournez and Maler 2000]. For example, in the three-dimensional space, the average experimental time complexity of converting an

EVR to a DBR is far less than quadratic but slightly greater than linear in the number of extreme vertices [Aguilera 1998]. Results in [Bournez and Maler 2000] also imply that, in a fixed dimension, the time complexity of converting a DBR to an EVR by using XOR operations is linear in the number of boxes in DBR. We then propose to exploit the ideas of these effective transformation schemes to produce a compact representation of contiguous aligned boxes, using the following **Combination** procedure:

1. Produce a better alignment of the boxes along the boundaries of constraints. This is done by preventing the unnecessary application of contracting operators over inactive variables. Figure 4.9 shows the effect of better alignment obtained for a set of nearly aligned boxes of an undiscernible approximation. The original set of eight small boxes (Figure 4.9a) reduces to two groups of four aligned boxes (Figure 4.9b) without altering the predefined interval-based precision.

2. The set of aligned boxes in each group, $\mathcal{S}_1$, is converted to EVR and then back to DBR to get a set of combined boxes, $\mathcal{S}_2$ (containing only one box in this case). Due to the properties of EVR, $\mathcal{S}_2$ is more concise than $\mathcal{S}_1$. Figure 4.9b shows how this conversion procedure reduces the two groups of four boxes to two (dark gray) boxes.

The above conversion procedure can theoretically be applied in any dimension. Due to the efficiency of EVR in low dimension, we however restrict its application to very low dimensional and small-size regions of the search space in our implementation (see Section 4.4.6). The running time for this conversion is hence near zero.

4.4.6. Search Algorithms

First, for convenience in comparing the previously available algorithm with our revised algorithm, we present in Algorithm 4.1 a slightly generalized and improved version of the **UCA6** algorithm – a search technique proposed by Silaghi *et al.* [2001]. Basically, this version is the same as the original version, but it is improved by changing the order of the pruning phase (Function **solveQuickly**) such that each box is pruned before being put into the waiting list. This change reduces the number of subproblems in the waiting list because some inconsistent subproblems can be discarded sooner than that in the original version in [Silaghi *et al.* 2001]. This version is general enough to be used with the heuristics in [Silaghi *et al.* 2001]. Those heuristics are represented by a generic function, called getSplitType, at Line 1 in Algorithm 4.1. Moreover, in this version, we make the stop condition more explicit by Function **isEpsilonBox**. It is important (for gaining in performance) to emphasize that checking if a domain (i.e., an interval) is not wider than the predefined precision ε is only performed on variables that occur in some running constraint, because some constraints have become redundant. This detail has been omitted in both [Silaghi *et al.* 2001] and [Silaghi 2002, Section 5.2.3].

Notation 4.27. The notations to be used in Algorithm 4.1 and Algorithm 4.4 follow the following conventions:

- The notations $\mathbf{B}$, $\mathbf{B}_i$ and $\mathbf{B}'$ denote relevant bounding boxes in $\mathbb{I}_\circ^n$, that is, the vectors of domains of the considered NCSPs.
- The notations $\mathcal{C}$, $\mathcal{C}_i$, $\mathcal{C}'$ and $\mathcal{C}''$ denote relevant sets of constraints.
- The notation $\mathbf{CB}_c$ denote a complementary box w.r.t. a constraint, c.

- The notation $\boxplus^{\mathcal{I}}$ and $\boxplus^{\mathcal{B}}$ denote the global lists that accumulate computed boxes of inner and boundary union approximations, respectively.
- The notations DR, CB, and FC denote some domain reduction operator, complementary boxing operator, and feasibility checker, respectively.
- The notations $\mathtt{DR_{rd}}$ and $\mathtt{CB_{rd}}$ denote the restricted-dimensional version of some domain reduction operator and complementary boxing operator, respectively.

Algorithm 4.1: The **UCA6** algorithm – a slightly improved version

```
Input: a bounding box B_0, a constraint set C_0, ε ∈ R^n_+, a feasibility checker FC.
Output: an inner union approximation ⊞^I, a boundary union approximation ⊞^B.
⊞^I := ∅; ⊞^B := ∅; WaitingList := ∅;
if solveQuickly(B_0, C_0, {B_0, ..., B_0}, ε, FC, WaitingList) then return ;   ◀ Page 120.
while WaitingList ≠ ∅ do        ▼/* A set {CB_c | c ∈ C} of memorized complementary boxes. */
    (B, C, {CB_c | c ∈ C}) := getNext(WaitingList);
    foreach c ∈ C do
        CB_c := CB(B ∩ CB_c, c);
        if CB_c = ∅ then C := C \ {c};        ◀ c is now redundant in B (Theorem 4.14).
    end
    if C = ∅ then
        ⊞^I := ⊞^I ∪ {B}; continue while;
    end
1   Splitter := getSplitType();           ◀ Get a splitting mode, heuristics can be used.
    if Splitter = BS then                 ◀ The splitting mode is box splitting.
        CB_b := chooseTheBest({CB_c | c ∈ C});
        (B_1, ..., B_k) := BS(B, CB_b);   ◀ If box splitting did not fail, then ∃B_i ⊇ CB_b.
        if BS failed then Splitter := DS;
    end
    if Splitter = DS then (B_1, ..., B_k) := DS(B);                   ◀ Bisect B.
    for i := 1, ..., k do
        C_i := C;
        if Splitter = BS and B_i ∩ CB_b = ∅ then
            C_i := C_i \ {b};                                         ◀ b is now redundant.
            if C_i = ∅ then
                ⊞^I := ⊞^I ∪ {B_i}; continue for;
            end
        end
        solveQuickly(B_i, C_i, {CB_c | c ∈ C_i}, ε, FC, WaitingList);  ◀ On page 120.
    end
end
```

Function solveQuickly(B, $\mathcal{C}$, $\{\mathbf{CB}_c \mid c \in \mathcal{C}\}$, ε, FC, WaitingList)

```
B' := DR(B, C);                                        ◀ Reduce domains.
if B' = ∅ then return true;          ◀ B is infeasible, the problem has been solved.
if isEpsilonBox(B', C, ε, FC) then return true;   ◀ The problem has been solved. Page 121.
put(WaitingList ← (B', C, {CB_c | c ∈ C}));   ◀ Put the current problem into the waiting list.
return false;                               ◀ The problem has not been solved yet.
```

Function isEpsilonBox($\mathbf{B}$, $\mathcal{C}$, ε, FC)

```
if the domain (in B) of some variable occurring in some running constraint is neither
bounded by ε nor canonical then return false;
if (RESULT := FC(B,C)) = feasible then          ◂ Identify the feasibility of B w.r.t. C.
|  ⊞^I := ⊞^I ∪ {B};                            ◂ B is feasible, store it into the list of inner boxes.
else if RESULT = unknown then
|  ⊞^B := ⊞^B ∪ {(B,C)};                        ◂ B is undiscernible, store it into the list of boundary boxes.
end
return true;
```

Second, we propose in Algorithm 4.4 a revised version of the improved version of the **UCA6** algorithm, which is called **UCA6**$^+$ (revised union-constructive approximation). It takes as input an NCSP, $\mathcal{P} \equiv (\mathcal{V}, \mathcal{C}, \mathcal{D})$, and returns an inner union approximation ($\boxplus^{\mathcal{I}}$) and a boundary union approximation ($\boxplus^{\mathcal{B}}$) the solution set of $\mathcal{P}$. The outer union approximation of the solution set can be computed by $\boxplus^{\mathcal{O}} := \boxplus^{\mathcal{I}} \cup \boxplus^{\mathcal{B}}$. Roughly speaking, the **UCA6**$^+$ algorithm uses restricted-dimensional versions of domain reduction and complementary boxing operators instead of normal versions to produce the effect of better alignment (and also to gain in performance), and then uses the conversion between the extreme vertex representation and disjoint box representation to get a compact representation of union approximations. The **UCA6**$^+$ algorithm proceeds by recursively repeating three main steps/processes:

(*i*) Use $\mathtt{DR}_{\mathtt{rd}}$ operators to reduce the current bounding box, playing the role of variable domains, to a narrower ones (see Function **solveQuickly**$^+$).

(*ii*) Use $\mathtt{CB}_{\mathtt{rd}}$ operators to get a list of complementary boxes w.r.t. each running constraint and the new bounding box obtained at Step (*i*) (Line 2 in Algorithm 4.4). The constraints that make empty complementary boxes are removed (Line 3 in Algorithm 4.4).

(*iii*) Combine dichotomous splitting (DS) operators with box splitting (BS) operators for the whole set of running constraints (see the lines 4, 5, and 7 in Algorithm 4.4).

Remark 4.28. In practice, equality constraints usually define surfaces, we then do not need to perform Step (*ii*) for them.

The **UCA6**$^+$ algorithm uses a waiting list, WAITINGLIST, to store the subproblems waiting to be processed further. The elements can be retrieved from, and be put to, WAITINGLIST by the functions getNext and put. WAITINGLIST can be handled as a queue or a stack. This allows for the breadth-first search in the former case and the depth-first search in the latter case.

The **UCA6**$^+$ algorithm does not compute complementary boxes for all running constraints as the **UCA6** algorithm does. Instead of this, it allows user to predefine a policy to choose a subset $\mathcal{C}''$ of $\mathcal{C}$, of which the constraints are enforced with restricted-dimensional complementary boxing ($\mathtt{CB}_{\mathtt{rd}}$) operators (see Line 1 in Algorithm 4.4). A simple policy is to choose either all the constraints of $\mathcal{C}$ or a fixed number of constraints in $\mathcal{C}$. A more complicated and dynamic policy based on the pruning efficiency can be used. The set of constraints to be considered for computing complementary boxes (by either using $\mathtt{CB}_{\mathtt{rd}}$ operators or intersecting with the memorized complementary boxes) is hence $\mathcal{C}' \cup \mathcal{C}''$, where $\mathcal{C}'$ is the set of constraints memorized together with its complementary boxes (see Line 2 in Algorithm 4.4).

Algorithm 4.4: The **UCA6$^+$** algorithm

```
Input: a bounding box B_0, a constraint set C_0, ε ∈ R^n_+, a feasibility checker FC, D_stop.
Output: an inner union approximation ⊞^I, a boundary union approximation ⊞^B.
⊞^I := ∅; ⊞^B := ∅; WaitingList := ∅;
if solveQuickly+(B_0, C'_0, ∅, ε, FC, WaitingList, D_stop) then return ;   ◄ On page 123.
while WaitingList ≠ ∅ do
      ▼/* A set {CB_c | c ∈ C} of complementary boxes that were optionally memorized. */
      (B, C, {CB_c | c ∈ C'}) := getNext(WaitingList);               ◄ C' ⊆ C, CB_c ⊂ B.
1     Choose an arbitrary set C'' ⊆ C;  ◄ C'' is a set of constraints to be used with the CB_rd operator.
2     foreach c ∈ C' ∪ C'' do
          if c ∈ C' ∩ C'' then
              CB_c := CB_rd(B ∩ CB_c, c);
          else if c ∈ C'' then                                        ◄ c ∉ C'.
              CB_c := CB_rd(B, c); C' := C' ∪ {c};
          else                                                        ◄ c ∈ C', c ∉ C''.
              CB_c := B ∩ CB_c;
          end
          if CB_c = ∅ then C := C \ {c};         ◄ c is now redundant in B (Theorem 4.14).
          if CB_c = ∅ or CB_c = B then  C' := C' \ {c};
      end
3     if C = ∅ then
          ⊞^I := ⊞^I ∪ {B}; continue while;
      end
4     Splitter := getSplitType();          ◄ Get a splitting mode, heuristics can be used.
5     if Splitter = BS then                      ◄ The splitting mode is box splitting.
6         CB_b := chooseTheBest({CB_c | c ∈ C'});
          (B_1, ..., B_k) := BS(B, CB_b);   ◄ If box splitting did not fail, then ∃B_i ⊇ CB_b.
          if C' = ∅ or BS failed then Splitter := DS;
      end
7     if Splitter = DS then (B_1, ..., B_k) := DS(B);                        ◄ Bisect B.
      for i := 1, ..., k do
          C_i := C; C'_i := C';
          if Splitter = BS and B_i ∩ CB_b = ∅ then
              C_i := C_i \ {b}; C'_i := C'_i \ {b};                ◄ b is now redundant.
              if C_i = ∅ then
                  ⊞^I := ⊞^I ∪ {B_i}; continue for;
              end
          end
8         foreach c ∈ C'_i do  if B_i ⊆ CB_c then  C'_i := C'_i \ {c};
          solveQuickly+(B_i, C_i, {B_i ∩ CB_c | c ∈ C'_i}, ε, FC, WaitingList, D_stop);
      end
end
```

Function getSplitType returns a mode of splitting the current domains, which are represented by a box. The current splitting mode can be inferred from the history of the current box (e.g., the splitting mode at the parent subproblem). In contrast to **DMBC**, the **UCA6** and **UCA6$^+$** algorithms only allow DS operators to dichotomize the domains of active variables,

Function solveQuickly⁺$(\mathbf{B}, \mathcal{C}, \{\mathbf{CB}_c \mid c \in \mathcal{C}'\}, \varepsilon, \texttt{FC}, \textsc{WaitingList}, D_{\text{stop}})$

```
   B' := DR_rd(B, C);                                   ◀ Restricted-dimensional domain reduction.
   if B' = ∅ then return true;                          ◀ B is infeasible, the problem has been solved.
   if isEpsilonBox+(B', C, ε, FC) then return true;     ◀ It has been solved. Page 123.
 9 if there are at most D_stop active variables in B' then     ▼/* Resort to another technique. */
10 |  (⊞^I(B', C), ⊞^B(B', C)) := DimStopSolver(B', C, ε, FC, DR_rd, CB_rd);
   |    ▼/* Combination(.) does the conversions DBR → EVR → DBR in a D_stop-dimensional space. */
   |  ⊞^I := ⊞^I ∪ Combination(⊞^I(B', C));          ◀ Store in the global list of feasible boxes.
   |  ⊞^B := ⊞^B ∪ Combination(⊞^B(B', C));          ◀ Store in the global list of undiscernible boxes.
   |  return true;                                     ◀ The problem has been solved.
   end
11 foreach c ∈ C' do if B' ⊆ CB_c then C' := C' \ {c};
   put(WaitingList ← (B', C, {B' ∩ CB_c | c ∈ C'}));   ◀ Put the problem into the waiting list.
   return false;                                      ◀ The problem has not been solved yet.
```

Function isEpsilonBox⁺$(\mathbf{B}, \mathcal{C}, \varepsilon, \texttt{FC})$

```
   if there exists an active variable in B w.r.t. C and ε then return false;   ◀ Def. 4.23.
   if (Result := FC(B, C)) = feasible then          ◀ Identify the feasibility of B w.r.t. C.
   |  ⊞^I := ⊞^I ∪ {B};                             ◀ B is feasible, store it into the list of inner boxes.
   else if Result = unknown then
   |  ⊞^B := ⊞^B ∪ {(B, C)};                        ◀ B is undiscernible, store it into the list of boundary boxes.
   end
   return true;
```

not the domains of inactive variables (see Line 7 in Algorithm 4.4). This avoids resulting in a huge number of tiny boxes. The reason is that, in the **UCA6⁺** algorithm, constraints are removed from consideration whenever empty complementary boxes are computed w.r.t. them (see Line 3 in Algorithm 4.4) and, maybe, some variables no longer appear in any running constraints. The interval (domain) with the greatest width is preferred for `DS` operators.

For efficiency, the `BS` operators (at Line 5 in Algorithm 4.4) split along some facet of a complementary box only if that produces sufficiently large boxes, the complementary box itself excepted. This estimation is done by using a predetermined parameter, fragmentation ratio. After the splitting phase (after Line 7 in Algorithm 4.4), if the box splitting operator was chosen and it did not fail (i.e., Splitter = `BS`), then there is a unique resulting box intersecting the best complementary box $\mathbf{CB}_b$, where $\mathbf{CB}_b$ was returned by Function chooseTheBest. In this case, the constraint b is redundant in every box that has empty intersection with $\mathbf{CB}_b$. This is due to the complementariness of $\texttt{CB}_{\texttt{rd}}$ operators.

The **UCA6⁺** algorithm use the same Function chooseTheBest (Line 6 in Algorithm 4.4) as the **UCA6** algorithm does. That is to choose the *best* complementary box $\mathbf{CB}_b$ (and the respective constraint b) based on some criteria to maximize the space surrounding the complementary box. The other complementary boxes can be memorized for improving the complementary boxing of subproblems. However, the memorization should be made optional because it may make the computation slow. Unlike the **UCA6** algorithm, the **UCA6⁺** algorithm only memorizes complementary boxes that do not contain the corresponding bounding box (see Line 8 in Algorithm 4.4 and Line 11 in Function **solveQuickly⁺**).

Function **solveQuickly**$^+$ (on page 123) attempts to apply a $\mathtt{DR_{rd}}$ operator to the input subproblem in a way similar to Function **solveQuickly** does. If it cannot solve this subproblem by using the $\mathtt{DR_{rd}}$ operator, it then checks if the subproblem has at most D_{stop} active variables. If the answer is yes, it resorts to a secondary solution technique, **DimStopSolver**, to solve the current subproblem, provided that **DimStopSolver** provides an output with good alignments. A good candidate for **DimStopSolver** is a search technique with the following branching strategies: *simple gridding*[4], 2^k-trees [Lottaz 2000; Sam-Haroud and Faltings 1996], and the bisection with generalized inclusion tests (see Definition 3.33), such as in the **Exclusion** algorithm (Algorithm 3.1, page 69) and in the **Inclusion** algorithm (Algorithm 3.2, page 77). Variants of **DMBC** or **UCA6** that use the restricted-dimensional operators can alternatively be used. For a given subproblem, **DimStopSolver** constructs two respective inner and boundary union approximations (see Line 10 of Function **solveQuickly**$^+$). These two union approximations are naturally represented in DBR (or a bounding-box tree). They are converted to EVR and then back to DBR in order to combine each group of contiguous aligned boxes into a bigger equivalent box. This operation is performed by Function **Combination**.

Function **isEpsilonBox**$^+$ (on page 123) checks if the input subproblem has no active variable. If so, it uses a feasibility checker, called `FC`, to check if the subproblem is either `infeasible`, `feasible`, or `unknown`. If `FC` returns `infeasible`, the subproblem is discarded. If `FC` returns `unknown`, the subproblem is classified as *undiscernible* w.r.t. ε and `FC`. In the other case, every point in the domains of the subproblem is a solution. Although the feasibility checker `FC` in our implementation is a combination of `DR` and `CB` operators as described in Theorem 4.19, it is however not restricted to this kind of feasibility checker.

Theorem 4.29. Given a feasibility checker, `FC`. The algorithm **UCA6**$^+$ terminates and computes the inner, boundary and outer union approximations at an arbitrary interval-based precision ε w.r.t. `FC` (see Definition 4.10).

Proof. The conclusion is easily deduced from the properties:

(*i*) The union approximations $\boxplus^{\mathcal{I}}$ and $\boxplus^{\mathcal{B}}$ are disjoint, and $\boxplus^{\mathcal{O}} := \boxplus^{\mathcal{I}} \cup \boxplus^{\mathcal{B}}$.

(*ii*) No solution is lost due to the correctness of $\mathtt{DR_{rd}}$ operators;

(*iii*) All points in the inner boxes (i.e., the boxes in $\boxplus^{\mathcal{I}}$) are sound solutions. That is due to complementariness of $\mathtt{CB_{rd}}$ operators (Definition 4.13) and Theorem 4.14;

(*iv*) The union of disjoint boxes $\boxplus^{\mathcal{B}}$ is equivalent to a union of the boxes (before applying EVR-DBR conversions by Function **Combination**) each of which have no active variable and each of which cannot be classified as `feasible` or `infeasible` by using the feasibility checker `FC` (see Function **isEpsilonBox**$^+$). That is due to the properties of EVR-DBR conversion and due to the fact that each box that has no active variable has the same feasibility (under feasibility checkers) with its projection – an ε-bounded box – on the space defined by all the variables of the running constraints. ∎

[4] A simple gridding/cell subdivision solver splits variable domains into a grid and then solves each subproblem in each grid cell.

4.5. Experiments

We now present a preliminary evaluation on the following small set of problems with non-isolated solutions (with different properties of constraints and the solution set): **WB**, **TD**, **P2**, **P3**, **FD**, **CD**. The reader can find their detailed descriptions in Section C.1.

For the purpose of evaluation, we have implemented the three search algorithms **DMBC**, **UCA6**, and **UCA6**$^+$ with different options, using the same data structures and the same domain reduction operators. We have also implemented a direct adaptation, called **UCA5**, of the **ICAb5** algorithm in [Benhamou and Goualard 2000] to solve NCSPs and a version of the **DMBC** algorithm to include a negation test (i.e., a complementary boxing operator). This enhanced version, called **DMBC**$^+$, of **DMBC** is a branch-and-prune method (see Section 3.2.2). It can check whether a box (playing the role of domains) is completely feasible or not, similarly to the **Inclusion** algorithm in Section 3.1.6. However, the **DMBC**$^+$ not only checks if a box is `infeasible` or `feasible`, but also allows narrowing the box in case the return value is `unknown`. In fact, the **DMBC**$^+$ algorithm is very similar to the **BnPSearch** algorithm (see Algorithm 6.5 in Section 6.6), except that the set of constraints is unchanged throughout the solving process. Our experiments discarded **DMBC** as a reasonable candidate for NCSPs with non-isolated solutions, particularly when the solution set contains continuums of solutions, because **DMBC** usually produces a huge number of boxes, each is ε-bounded, in very long running time.

The source codes of the above algorithms (**DMBC**, **DMBC**$^+$, **UCA5**, **UCA6**, and **UCA6**$^+$) are open in the BCS (*box covering solver*) module which is downloadable at the official web site of the COCONUT project, `http://www.mat.univie.ac.at/coconut-environment/`.

Table 4.1. The running time results for search algorithms.

Prob.	ε	**DMBC**$^+$ (DS), ¬MEM	**UCA6** (DS), MEM	**UCA6**$^+$ (DS), MEM	**UCA5** (BS+DS), ¬MEM	**UCA6** (BS+DS), MEM	**UCA6**$^+$ (BS+DS), ¬MEM
WP	0.1	22.07s	6.73s	5.98s	5.11s	4.77s	3.97s
TD	0.01	81.53s	26.45s	26.01s	14.96s	13.43s	3.59s
P3	0.1	>10h	615.98s	530.16s	87.09s	135.28s	1.41s
P2	0.1	>10h	4959.76s	4433.06s	281.51s	293.09s	7.32s
FD	0.1	3878.47s	1278.82s	740.05s	394.29s	439.07s	211.91s
CD	0.01	1308.63s	468.21s	389.12s	538.21s	493.57s	352.96s

Table 4.2. The numbers of boxes in inner union approximations (on the left) and boundary union approximations (on the right).

Prob. ▼	ε	**DMBC**$^+$ (DS), ¬MEM		**UCA6** (DS), MEM		**UCA6**$^+$ (DS), MEM		**UCA5** (BS+DS), ¬MEM		**UCA6** (BS+DS), MEM		**UCA6**$^+$ (BS+DS), ¬MEM	
WP	0.1	2753	2620	2753	2620	2489	2147	1738	2788	1573	2791	1176	1585
TD	0.01	3900	2917	3900	2917	3895	1970	2832	3270	1313	3496	53	50
P3	0.1	>110000	150000	33398	38006	30418	28229	10784	29888	12113	38808	406	970
P2	0.1	>120000	180000	108701	100027	106320	78755	21872	55901	18722	55063	1873	3225
FD	0.1	42178	66940	42178	66940	42084	47138	51882	65536	26378	70170	10341	35126
CD	0.01	8957	22132	8873	21974	8354	14857	12729	25079	9922	25344	2826	12922

Table 4.3. The ratios of the total volumes of inner union approximations to that of outer union approximations.

Prob.	ε	DMBC$^+$ (DS), ¬MEM	UCA6 (DS), MEM	UCA6$^+$ (DS), MEM	UCA5 (BS+DS), ¬MEM	UCA6 (BS+DS), MEM	UCA6$^+$ (BS+DS), ¬MEM
WP	0.1	0.992	0.992	0.991	0.994	0.994	0.993
TD	0.01	0.997	0.997	0.997	0.999	0.999	0.998
P3	0.1	n/a	0.907	0.912	0.980	0.980	0.919
P2	0.1	n/a	0.973	0.974	0.995	0.995	0.975
FD	0.1	0.987	0.987	0.984	0.992	0.992	0.986
CD	0.01	0.638	0.638	0.619	0.830	0.828	0.616

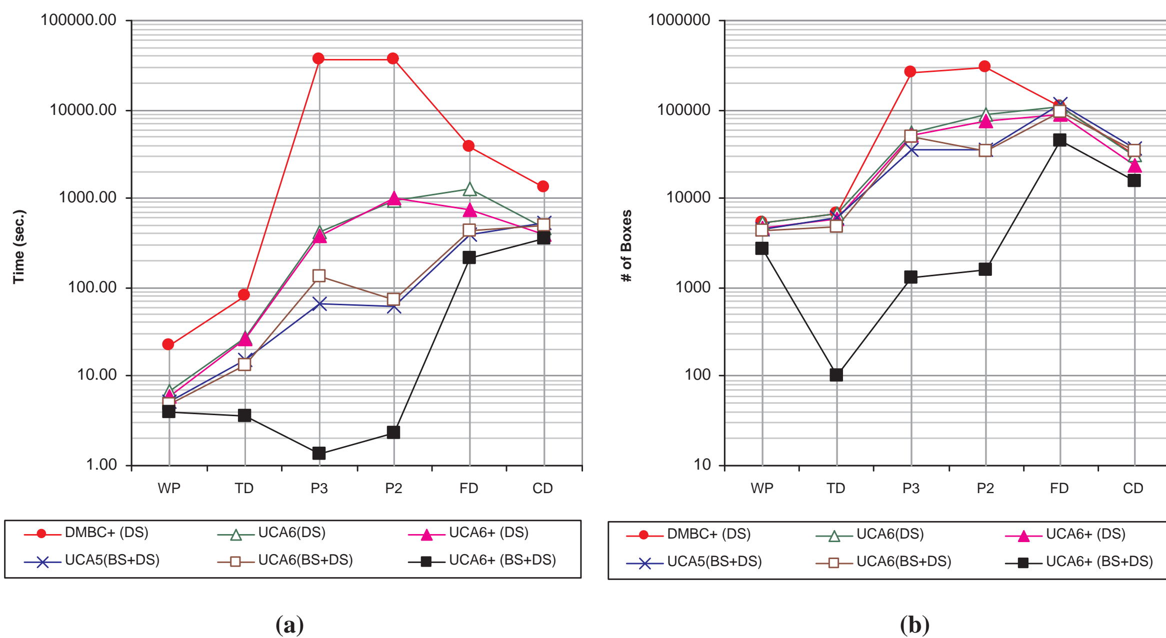

Figure 4.13. Logarithmic charts for: (a) the running time; and (b) the total number of boxes.

The empirical results are shown in Table 4.1, Table 4.2, and Table 4.3. A graphic overview of the results are shown in Figure 4.13. These results were obtained with the settings fragmentation ratio $= 0.25$, $D_{\text{stop}} = 1$, and with the feasibility checker FC being constructed as described in Theorem 4.19 (where the precision set to 1). The domain reduction operators (DR and $\mathtt{DR}_{\mathtt{rd}}$) were implemented by using the function `IloGenerateBounds` in ILOG Solver 5.3 [ILOG 2003]. The default precision for this function is set to the value 1, for optimal performance.[5] Let ε be the interval-based precision to stop at. For simplicity, we assume that all components of the precision vector ε are equal. The secondary search technique used in the **UCA6$^+$** algorithm is a simple gridding solver.[6]

[5] Our experiments on different values of the default precision for the function `IloGenerateBounds` showed that the value 1 seems to be the most suitable.

[6] The 2^k-tree based solver would serve better this purpose, but it requires more time for integration.

The terms (DS) and (BS+DS) in Table 4.1, Table 4.2, and Table 4.3 indicate the splitting strategies used in the respective search algorithms. The term MEM means the memorization of complementary boxes. Table 4.1 shows the running times of the search algorithms (in seconds and hours). Table 4.2 shows the numbers of boxes in inner and boundary union approximations delivered by the respective algorithms. Table 4.3 shows the ratios of the total volumes of inner union approximations to that of outer union approximations.

Together with other experiments on tens of similar or larger problems (including the experiments in Chapter 6 and Chapter 7), our experiments showed that the best gains, in the running time and the number of boxes, of the algorithms **UCA5**, **UCA6** and **UCA6$^+$** over **DMBC$^+$** are obtained for the cases the arities of individual constraints are less than the arity of the problem (e.g., **P2**, **P3**). In all the tests, the **UCA6$^+$** algorithm with the option (BS+DS) and with either the option MEM or the option ¬MEM is better than the other algorithms in the running time and the number of boxes. The best gains are obtained when (nonlinear) constraints contain a large percentage of nearly axis-parallel regions (e.g., **P2**, **P3**, **TD**). The **UCA6$^+$** algorithm with the option (BS+DS) is slightly less accurate than the **UCA5** algorithm and the **UCA6** algorithm in the volume measure, but the situation is improved when ε is reduced (i.e., in higher precision). However, this is hardly a matter for real world applications because no one could ever use all solutions when a very large percentage of sound solutions has been found.

In summary, the **UCA6$^+$** algorithm shows to be the most adaptive search algorithm among the above-considered search algorithms for NCSPs with continuums of solutions. For NCSPs with isolated solutions, all these search algorithms should be equal, in general, because they are assumed to use the same splitting strategy – the bisection. Its characteristics are clearly more suitable for real world applications than the other search algorithms.

The experiments are therefore encouraging enough to investigate other combinations of the control parameters and higher values of D_{stop} in combination with solvers with good alignments, such as 2^k-tree solvers [Lottaz 2000; Sam-Haroud 1995; Sam-Haroud and Faltings 1996], the **Exclusion** algorithm (Algorithm 3.1, page 69), and the **Inclusion** algorithm (Algorithm 3.2, page 77), in the place of the secondary solution technique (**DimStopSolver**).

4.6. Conclusion

Our contribution in this chapter is a new search strategy, called **UCA6$^+$**, which overcomes the limitations of the previously available search strategies: **DMBC** and **UCA6**. The **UCA6$^+$** algorithm revise the branching steps. The new contribution in the **UCA6$^+$** algorithm is twofold:

1. The **UCA6$^+$** algorithm allows better controlling domain reductions by restricting the domain reduction effort to active variables (see Definition 4.23). This feature is supported by many local consistency notions which are achieved by search such as box consistency, hull consistency, kB-consistency, and their variants (see Section 2.3.3 and Section 3.2.2). Moreover, when a subproblem has at most D_{stop} active variables, the **UCA6$^+$** algorithm resorts to a solver, which is efficient for small-size and low dimensional problems, with better alignments in the output. The gain is then in both the running time and the alignment of boxes.

2. The **UCA6$^+$** algorithm allows resorting to a geometrically compacting technique, such as the extreme vertex representation, to combine each group of locally aligned boxes resulting in the previous stage into a larger equivalent box. Hence, the representation

of the solution set is more concise. This hence potentially accelerates the query on the explicit representation of the solution set. To be efficient, the value of D_{stop} – which is considered as the dimension of the input problem for compacting techniques – should be small, for instance, $D_{\text{stop}} = 1$ or $D_{\text{stop}} = 2$.

In general, the **UCA6$^+$** algorithm provides improvements to previously available search techniques in case the solution set contains continuums of solutions, often by an order of magnitude, while keeping the same procedure/performance as bisection-based techniques (i.e., **DMBC**) in case solutions are isolated. In the former case, the **UCA6$^+$** algorithm provides inner and outer approximations w.r.t. a predefined interval-based precision ε and a feasibility checker (Definition 4.10). Moreover, a large percentage of provided approximations are often proved to be sound solutions (see Definition 2.47). Our experiments show that the new technique – the **UCA6$^+$** algorithm – improves the efficiency as well as the conciseness of the solution representation.

Additionally, we have also presented a general view on existing techniques such as feasibility checkers, domain reduction operators, complementary boxing operators, and splitting operators (see the sections 4.3.3, 4.4.1, 4.4.2, 4.4.3). That brings out a uniform view on existing search techniques and the position of domain reduction techniques in complete methods.

Chapter 5

Modification and Abstraction of Inclusion Techniques

Note: This chapter includes the research conducted jointly with Djamila Sam-Haroud, Hermann Schichl, and Boi Faltings in [Vu *et al.* 2004b,c]. This is necessary for Chapter 6 and Chapter 7.

5.1. Introduction

In this chapter, we present some modifications and improvements to the standard versions of interval arithmetic and affine arithmetic for the purpose of constraint propagation in Chapter 6 and Chapter 7. We also propose an abstract inclusion concept in order to present a novel generic scheme for constraint propagation concisely and uniformly in Chapter 7.

First, we presents in Section 5.2 a consistent way to extend the functions that are only defined on subsets of the real set $\mathbb{R}$ for specific situations in constraint propagation. We then revise, in Section 5.3, the concept of an interval form for extended functions. An example on the revision of the division in interval arithmetic is also given in Section 5.3. The revised version of interval arithmetic allows obtaining slightly tighter enclosures in constraint propagation.

Second, we revise, in Section 5.4, the standard version of affine arithmetic for the purpose of constraint propagation. Affine forms are revised such that the number of noise variables will not increase during computations; hence, it is potentially useful for long-running computations. We point out that Kolev generalized affine arithmetic (Section 2.2.3.4) on revised affine forms is about two times faster than that on Kolev affine forms. Moreover, we propose a new formula for the product of two affine forms, which reduces the complexity $\mathcal{O}(n^2)$ of the multiplication in affine arithmetic and Kolev generalized affine arithmetic to $\mathcal{O}(n)$. This formula provides tighter enclosures than the most recent version in Kolev affine arithmetic (Section 2.2.3.4) while keeping the same number of real operations. We also propose, in Section 5.4, a generic procedure to compute Chebyshev affine approximations in a rigorous manner. When applied to elementary operations, rigorous affine approximations can be obtained for virtually any factorable function which is composed of the elementary operations.

Finally, we propose, in Section 5.5, an abstract inclusion concept, called the *real inclusion representation*. This abstract concept does not result in new inclusion techniques, but provides a common view into inclusion techniques. It facilitates presenting a novel generic scheme for constraint propagation proposed in Chapter 7. Some common properties of inclusion techniques are also drawn out formally in Section 5.5.

5.2. Extended Functions

In practice, we sometimes encounter functions of the form $f : D \to \mathbb{R}^m$, where $D \subset \mathbb{R}^n$. For example, a division by zero, such as $1 \div 0$, is not defined. Consequently, the division of two intervals is not defined in (standard) interval arithmetic when the denominator contains zero. In such cases, many implementations of interval arithmetic give, by convention, the universe interval $[-\infty, \infty]$ as the result. This is an extension of (standard) interval arithmetic for all purposes, in order to preserve the inclusion property. If we use these implementations to evaluate the range of the function $f : D \subset \mathbb{R}^n \to \mathbb{R}^m$, we will often get unnecessarily overestimated bounds of the form $[-\infty, \infty]$ if the denominators of f contain zero. In order to avoid such an overestimation when using interval arithmetic, we have to extend functions in accordance with the use in specific computations. Hereafter, we give a way, in accordance with the concept of a *multifunction*, to extend functions that are only defined on subsets of $\mathbb{R}^n$. First, we recall the definition of a multifunction from [Singh *et al.* 1997, p. 34].

Definition 5.1 (Multifunction, Values, Fibers). Let X and Y be two sets. A *multifunction* F from X to Y, denoted by $F : X \to Y$, is a subset $F \subseteq X \times Y$. The inverse of F is a multifunction $F^{-1} : Y \to X$ defined by the rule: $(y, x) \in F^{-1} \Leftrightarrow (x, y) \in F$. We define the *values* of F at x be $F(x) \equiv \{y \in Y \mid (x, y) \in F\}$, and the *fibers* of F for $y \in Y$ be $F^{-1}(y) \equiv \{x \in X \mid (x, y) \in F\}$.

In Definition 5.1, if for some $x \in X$ there is no $y \in Y$ such that $(x, y) \in F$, we assume that $F(x) \equiv \emptyset$. From Definition 5.1 we can see that a function is, in fact, a special multifunction that is single-valued. Second, the concepts of an image and an inverse image (of a set) under a multifunction are similar to those under a normal function.

Definition 5.2 (Image, Inverse Image). Let X and Y be two sets, $F : X \to Y$ a multifunction. The *image of a subset $A \subseteq X$ under F* is defined and denoted by

$$F(A) \equiv \bigcup_{x \in A} F(x) \equiv \{y \in F^{-1} \mid F^{-1}(y) \cap A \neq \emptyset\}. \tag{5.1}$$

The *inverse image of a subset $B \subseteq Y$ under F* is defined and denoted by

$$F^{-1}(B) \equiv \bigcup_{y \in B} F^{-1}(y) \equiv \{x \in F \mid F(x) \cap B \neq \emptyset\}. \tag{5.2}$$

Next, we define a special class of multifunctions.

Definition 5.3 (Extended Function). Let f be a function from a set X to a set Y, X' a superset of X, and Z a set of some subsets of Y possibly including $\emptyset$. A *Z-extended function over X' of f* is a multifunction $F : X' \to Y$ such that

$$\forall x \in X : F(x) = \{f(x)\}, \tag{5.3}$$

$$\forall x \in X' \setminus X : F(x) \in Z. \tag{5.4}$$

Note 5.4. When we do not care about Z in Definition 5.3, we just call F an extended function over X' of f.

Notation 5.5. For simplicity, in Definition 5.3, for all $x \in X$ we write $F(x) = f(x)$ when no confusion may arise.

A Z-extended function F, as defined in Definition 5.3, corresponds to a function $g : X' \to Y \cup Z$ defined as

$$g(x) \equiv \begin{cases} f(x) & \text{if } x \in X, \\ F(x) & \text{otherwise.} \end{cases} \tag{5.5}$$

If $X' = X$, then $f(x) = g(x)$ for all x in X. The following theorem follows from Definition 5.3.

Theorem 5.6. Let f, F and other notations be as in Definition 5.3. Then, for every subset S of X', we have

$$f(S) \equiv \{f(x) \mid x \in S \cap X\} \subseteq F(S). \tag{5.6}$$

Consider the case $X = D \subseteq \mathbb{R}^n, Y = \mathbb{R}^m$. It is easy to see that, for any function $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$, there is only one Z-extended function from $\mathbb{R}^n$ to $\mathbb{R}^m$ if Z has only one element, for example, when Z is either $\{\emptyset\}$ or $\{\mathbb{R}\}$.

Example 5.7. The domain of the standard division $x \div y$ is $D_\div = \{(x, y) \in \mathbb{R}^2 \mid y \neq 0\}$. The unique $\{\emptyset\}$-extended function over $\mathbb{R}$ of the standard division is defined as

$$\div_\emptyset(x, y) \equiv x \div_\emptyset y \equiv \begin{cases} x/y & \text{if } y \neq 0, \\ \emptyset & \text{otherwise.} \end{cases} \tag{5.7}$$

The unique $\{\mathbb{R}\}$-extended function over $\mathbb{R}$ of the standard division is defined as

$$\div_\mathbb{R}(x, y) \equiv x \div_\mathbb{R} y \equiv \begin{cases} x/y & \text{if } y \neq 0, \\ \mathbb{R} & \text{otherwise.} \end{cases} \tag{5.8}$$

The following is a $\{\emptyset, \mathbb{R}\}$-extended function over $\mathbb{R}$ of the standard division:

$$\div_\star(x, y) \equiv x \div_\star y \equiv \begin{cases} x/y & \text{if } y \neq 0, \\ \emptyset & \text{if } x \neq 0, y = 0, \\ \mathbb{R} & \text{otherwise.} \end{cases} \tag{5.9}$$

♣

Example 5.8. The domain of the standard square root $\sqrt{x}$ is $D_{\sqrt{}} = [0, +\infty]$. The unique $\{\emptyset\}$-extended function over $\mathbb{R}$ of the standard square root is defined as

$$\sqrt{x}^\emptyset \equiv \begin{cases} \sqrt{x} & \text{if } x \geq 0, \\ \emptyset & \text{otherwise.} \end{cases} \tag{5.10}$$

The unique $\{\mathbb{R}\}$-extended function over $\mathbb{R}$ of the standard square root is defined as

$$\sqrt{x}^\mathbb{R} \equiv \begin{cases} \sqrt{x} & \text{if } x \geq 0, \\ \mathbb{R} & \text{otherwise.} \end{cases} \tag{5.11}$$

♣

5.3. Modification to Interval Arithmetic

The interval form concept for a function (Definition 2.80) is extended to that for a multifunction in the next definition. Consequently, the concept of an interval form is also defined for extended functions because an extended function is a special instance of multifunctions.

Definition 5.9 (Interval Form of Multifunctions). Let $F : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ be a multifunction. A function $[F] : \mathbb{I}^n \to \mathbb{I}^m$ is called an *interval form* of F (or for F) if the *inclusion property* holds; that is,

$$\forall x \in D, \forall \mathbf{x} \in \mathbb{I}^n : x \in \mathbf{x} \Rightarrow F(x) \subseteq [F](\mathbf{x}). \tag{5.12}$$

The *natural interval form* of f (see Section A.2.1) is an instance of interval forms. The following theorem sates the inclusion property of interval form of multifunctions.

Theorem 5.10. Let $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ be a function and $F : D' \supseteq D \to \mathbb{R}^m$ an extended function over D' of f. Then every interval form of F is an interval form of f.

Proof. Let $[F] : \mathbb{I}^n \to \mathbb{I}^m$ be an interval form of F. Then for every $x \in D$ and every box $\mathbf{x} \in \mathbb{I}^n$ containing x, we have $f(x) \in \{f(x)\} = F(x) \subseteq [F](\mathbf{x})$. ■

Definition 5.11 (Interval Division: $[\div_\emptyset]$, $[\div_\mathbb{R}]$, $[\div_\star]$). Let $\mathbf{x} = [\underline{x}, \overline{x}]$ and $\mathbf{y} = [\underline{y}, \overline{y}]$. Here we give three natural interval forms for the divisions defined by (5.7), (5.8) and (5.9), respectively:

$$[\div_\emptyset](\mathbf{x},\mathbf{y}) \equiv \mathbf{x}[\div_\emptyset]\mathbf{y} \equiv \begin{cases} \emptyset & \text{if } \mathbf{y} = [0,0], \\ [0,0] & \text{else if } \mathbf{x} = [0,0], \\ \mathbf{x} \div \mathbf{y} & \text{else if } 0 \notin \mathbf{y} \text{ (see (2.24))}, \\ [\underline{x}/\overline{y}, +\infty] & \text{else if } \underline{x} \geq 0 \wedge \underline{y} = 0, \\ [-\infty, \underline{x}/\underline{y}] & \text{else if } \underline{x} \geq 0 \wedge \overline{y} = 0, \\ [-\infty, \overline{x}/\overline{y}] & \text{else if } \overline{x} \leq 0 \wedge \underline{y} = 0, \\ [\overline{x}/\underline{y}, +\infty] & \text{else if } \overline{x} \leq 0 \wedge \overline{y} = 0, \\ [-\infty, +\infty] & \text{otherwise;} \end{cases} \tag{5.13}$$

$$[\div_\mathbb{R}](\mathbf{x},\mathbf{y}) \equiv \mathbf{x}[\div_\mathbb{R}]\mathbf{y} \equiv \begin{cases} \mathbf{x} \div \mathbf{y} & \text{if } 0 \notin \mathbf{y}, \\ [-\infty, +\infty] & \text{otherwise;} \end{cases} \tag{5.14}$$

$$[\div_\star](\mathbf{x},\mathbf{y}) \equiv \mathbf{x}[\div_\star]\mathbf{y} \equiv \begin{cases} \mathbf{x}[\div_\emptyset]\mathbf{y} & \text{if } 0 \notin \mathbf{x} \vee 0 \notin \mathbf{y}, \\ [-\infty, +\infty] & \text{otherwise.} \end{cases} \tag{5.15}$$

In literature, for instance [Hickey *et al.* 2001], we can find the tightest range of the division of two intervals; however, the result is not always an interval.

Theorem 5.12. For any two intervals $\mathbf{x}$ and $\mathbf{y}$ in $\mathbb{I}$, we have $\mathbf{x}[\div_\emptyset]\mathbf{y} \subseteq \mathbf{x}[\div_\star]\mathbf{y} \subseteq \mathbf{x}[\div_\mathbb{R}]\mathbf{y}$.

Proof. It follows from Definition 5.11. ■

Theorem 5.13. Let x, y, and z be three real numbers living in three intervals $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ in $\mathbb{I}$, respectively. Then we have

$$x = y * z \quad \Rightarrow \quad z \in \mathbf{x} \diamond \mathbf{y} \text{ for all } \diamond \in \{[\div_\star], [\div_\mathbb{R}]\}, \tag{5.16}$$

$$z = x/y \quad \Rightarrow \quad z \in \mathbf{x} \diamond \mathbf{y} \text{ for all } \diamond \in \{[\div_\emptyset], [\div_\star], [\div_\mathbb{R}]\}. \tag{5.17}$$

Proof. The proof follows from Definition 5.11, with noticing that from a given equality $x = y*z$ we can deduce that

(i) If $y \neq 0$, then $z = x/y$;

(ii) If $y = 0$, then $x = 0$ and z can take an arbitrary value. ■

On one hand, Theorem 5.13 shows that, when given the relation $x = y * z$, it is safe to use the domain reduction $\mathbf{z} := \mathbf{x} \diamond \mathbf{y}$ for any $\diamond \in \{[\div_\star], [\div_\mathbb{R}]\}$. On the other hand, if the equality $z = x/y$ is given, we can safely reduce the domain of z by the rule $\mathbf{z} := \mathbf{x} \diamond \mathbf{y}$ for any $\diamond \in \{[\div_\mathbb{R}], [\div_\star], [\div_\emptyset]\}$, because the case $y = 0$ is not admitted by definition.

Theorem 5.12 and Theorem 5.13 show that the tightness of a natural interval form of a function defined on a subset of $\mathbb{R}^n$ might be dependent of the underlying extended function. In turn, the extended functions to be used depend on the context of computations. Unfortunately, some interval implementations, such as the one in [Walster *et al.* 2000], use the division $[\div_\mathbb{R}]$, while it is safe to use the division $[\div_\emptyset]$ in some computations such as forward evaluation and use the division $[\div_\star]$ in some computations such as backward propagation, as described in Section 6.3 (see also Section 3.2.2.2).

5.4. Revised Affine Arithmetic

In the standard version, affine arithmetic (defined on affine forms) has several limitations in performance when used in long-running computations. The well-known limitation of affine forms and affine arithmetic is that the number of noise variables grows quickly during computations since each nonlinear operation adds a new noise variable. In general, the cost of operations in affine arithmetic heavily depends on the number of noise variables (e.g., linearly or quadratically). The computations in which affine arithmetic is used to generate linear relaxations and linear programming is used to reduce domains even suffer from this limitation much more than the operations in affine arithmetic do. Another obvious limitation of affine forms is that they are not capable of handling half-lines of the form $[-\infty, a]$ and $[a, +\infty]$, while this is needed in many computation methods such as constraint propagation and exhaustive search. For the purpose of constraint propagation and search, we propose some slight modifications and improvements to affine form and affine arithmetic. It is to make affine arithmetic more suitable for specific types of computations. These modifications and improvements are very useful for the constraint propagation technique proposed in Chapter 7. Hereafter, we present these modifications and improvements.

5.4.1. Revised Affine Form

Inspired by the ideas of Messine affine arithmetic and Kolev generalized affine arithmetic (see Section 2.2.3.4), we propose a kind of affine form, called *revised affine form*, which is similar

to the affine form (2.35); namely, it is similar to

$$\hat{z} \equiv z_0 + z_1\epsilon_1 + \cdots + z_n\epsilon_n + z_{\text{new}}\epsilon_{\text{new}} \tag{5.18}$$

but the new term $z_{\text{new}}\epsilon_{\text{new}}$ is replaced with a symmetric interval $e_z[-1,1]$, called the *accumulative error*, that bounds the maximum error of non-affine operations. Namely, a *revised affine form* of a real-valued quantity/variable x is defined as

$$\widehat{x} = x_0 + x_1\epsilon_1 + \cdots + x_n\epsilon_n + e_x[-1,1]. \tag{5.19}$$

This form consists of two separated parts: an affine part of length n and an interval part; thus, it is said to be of length n. This form is similar to, but more concise than, the Messine affine form (2.45). The magnitude of the accumulative error, $e_x \geq 0$, is identified by the interval part. We write $x \in \widehat{x}$ if, for each real value x of the quantity $\widehat{x}$, there exist $\epsilon_x \in [-1,1]$ and $\epsilon_i \in [-1,1]$ (for $i = 1,\ldots,n$) such that $x = x_0 + x_1\epsilon_1 + \cdots + x_n\epsilon_n + e_x\epsilon_x$.

In fact, revised affine form (5.19) is a special case of Kolev affine/interval form (see Definition 2.83). The two parts of a revised affine form are computed separately in an affine operation. For example, an affine operation on two revised affine forms are now defined as

$$\widehat{z} \equiv \alpha\widehat{x} + \beta\widehat{y} + \gamma = (\alpha x_0 + \beta y_0 + \gamma) + \sum_{i=1}^{n}(\alpha x_i + \beta y_i)\epsilon_i + (|\alpha|e_x + |\beta|e_y)[-1,1]. \tag{5.20}$$

Therefore, during long-running computations the lengths of revised affine forms will not exceed the number of noise symbols at the beginning (which equals the number of variables of the input constraint system). Note that we should implement special affine operations separately to gain in speed. In rigorous computing, e_z is used to accumulate the rounding errors in floating-point arithmetic; namely, (5.20) can be interpreted as follows:

$$z_0 = \langle \alpha x_0 + \beta y_0 + \gamma \rangle \pm e_0, \tag{5.21a}$$
$$z_i = \langle \alpha x_i + \beta y_i \rangle \pm e_i, \tag{5.21b}$$
$$e_z = \lceil |\alpha|e_x + |\beta|e_y + \sum_{i=0}^{n} e_i \rceil \tag{5.21c}$$

Theorem 5.14. The affine operation defined by (5.20) or (5.21) is an interval form of its counterpart (i.e., the real-valued operation).

Proof. This theorem is obvious; hence, the proof is omitted. ∎

We propose to associate each quantity $\widehat{x}$ with a data field $x_\infty \in \{-1, 0, +1\}$ to represent the half-lines of the form $[-\infty, a]$ and $[a, +\infty]$. The *revised affine form* is then interpreted as follows:[1]

$$\widehat{x} \equiv \begin{cases} [-\infty, +\infty] & \text{if } e_x = +\infty, \\ [-\infty, x_0] & \text{else if } x_\infty = -1, \\ [x_0, +\infty] & \text{else if } x_\infty = +1, \\ x_0 + x_1\epsilon_1 + \cdots + x_n\epsilon_n + e_x[-1,1] & \text{otherwise.} \end{cases} \tag{5.22}$$

[1] For simplicity, we allow zero coefficients in the formulae here, however in implementation one should keep only nonzero coefficients and their indices.

Remark 5.15. In an operation, if the domain of a variable is unbounded (i.e., in the first three cases of (5.22)), the other variables are converted into interval forms for that operation performed in interval arithmetic, then the result is converted back to revised affine form. Therefore, we only need to discuss about the last case of (5.22) in the rest of this thesis.

Notation 5.16. We denote by $\hat{\mathbb{A}}$ the set of all affine forms and by $\mathbb{A}$ the set of all revised affine forms of the form (5.22).

5.4.2. Multiplication

Since every revised affine form of the form (5.19) is a special case of Kolev affine/interval forms (see Definition 2.83), we can apply Kolev generalized affine arithmetic to revised affine forms. Indeed, by letting the radius (v_i) of all noise variables in (2.43) be 1, the multiplication of revised affine forms in Kolev generalized affine arithmetic, $\widehat{z} := \widehat{x} * \widehat{y}$, is defined as follows:

$$z_0 \equiv x_0 y_0 + \frac{1}{2}\sum_{i=1}^{n} x_i y_i, \tag{5.23a}$$

$$z_i \equiv x_0 y_i + y_0 x_i \quad (\text{for } i = 1, \ldots, n), \tag{5.23b}$$

$$e_z \equiv e_x e_y + |x_0| e_y + |y_0| e_x + e_y \sum_{i=1}^{n} |x_i| + e_x \sum_{i=1}^{n} |y_i| + \sum_{1 \le i,j \le n;\ i \ne j} |x_i y_j| + \frac{1}{2}\sum_{i=1}^{n} |x_i y_i|. \tag{5.23c}$$

The number of real additions in (5.23) is $n^2 + 4n + 2$, which is the same as that in (2.43). However, the number of real multiplications is $n^2 + 2n + 8$, which is less than that number, $3n^2 + 4n + 8$, of (2.43). The total number of operations is $2n^2 + 6n + 10$; hence, it is about two times less than that number, $4n^2 + 8n + 10$, of (2.43). Roughly speaking, the multiplication in Kolev generalized affine arithmetic is about two times slower than its simplification (5.23) for revised affine forms.

Moreover, we propose a faster *multiplication* in which the product of two revised affine forms $\widehat{x} = x_0 + x_1\epsilon_1 + \cdots + x_n\epsilon_n + e_x[-1, 1]$ and $\widehat{y} = y_0 + y_1\epsilon_1 + \cdots + y_n\epsilon_n + e_y[-1, 1]$ is another revised affine form $\widehat{z} = z_0 + z_1\epsilon_1 + \cdots + z_n\epsilon_n + e_z[-1, 1]$ defined as follows:

$$S_x \equiv \sum_{i=1}^{n} |x_i|, \quad S_y \equiv \sum_{i=1}^{n} |y_i|, \quad S_1 \equiv 0.5 \sum_{i=1;\ x_i y_i \ge 0}^{n} x_i y_i, \quad S_2 \equiv 0.5 \sum_{i=1;\ x_i y_i < 0}^{n} x_i y_i, \tag{5.24a}$$

$$z_0 \equiv x_0 y_0 + (S_1 + S_2), \tag{5.24b}$$

$$z_i \equiv x_0 y_i + y_0 x_i \quad (\text{for } i = 1, \ldots, n), \tag{5.24c}$$

$$e_z \equiv e_x e_y + e_y(|x_0| + S_x) + e_x(|y_0| + S_y) + S_x S_y - (S_1 - S_2). \tag{5.24d}$$

The number of real additions in (5.24) is $4n + 5$, the number of real multiplications in (5.24) is $3n + 7$. The total number of real operations in (5.24) is $7n + 12$. This is the same as in Kolev's multiplication (2.55) (see Section 2.2.3.4). It however provides tighter enclosures than what provided by Kolev's multiplication (2.55) (see Section 2.2.3.4), because $|P| = 0.5|\sum_{i=1}^{n} x_i y_i| \le 0.5\sum_{i=1}^{n} |x_i y_i| = (S_1 - S_2)$. Note that in [Vu *et al.* 2004b], we proposed another version of the

multiplication (5.24) with the same tightness and the cost $8n + 10$, but with simpler formulas. **These improvements can be easily transferred to other variants of affine arithmetic such as Kolev generalized affine arithmetic** in Section 2.2.3.4. In rigorous computations, we use the following formulas with error rounding controls:

$$S_x \equiv \lceil \sum_{i=1}^{n} |x_i| \rceil, \quad S_y \equiv \lceil \sum_{i=1}^{n} |y_i| \rceil, \tag{5.25a}$$

$$S_1 \equiv \langle 0.5 \sum_{i=1;\ x_i y_i \geq 0}^{n} x_i y_i \rangle \pm e_{n+1}, \quad S_2 \equiv \langle 0.5 \sum_{i=1;\ x_i y_i < 0}^{n} x_i y_i \rangle \pm e_{n+2}, \tag{5.25b}$$

$$z_0 \equiv \langle x_0 y_0 + (S_1 + S_2) \rangle \pm e_0, \tag{5.25c}$$

$$z_i \equiv \langle x_0 y_i + y_0 x_i \rangle \pm e_i \quad (\text{for } i = 1, \ldots, n), \tag{5.25d}$$

$$e_z \equiv \lceil e_x e_y + e_y(|x_0| + S_x) + e_x(|y_0| + S_y) + S_x S_y + (S_2 - S_1) + \sum_{i=0}^{n+2} e_i \rceil. \tag{5.25e}$$

Theorem 5.17. The multiplication defined by (5.24) (or by (5.25)) is an interval form of the real multiplication; that is, $\forall x \in \widehat{x}, \forall y \in \widehat{y} : xy \in \widehat{z} \equiv \widehat{x}\widehat{y}$.

Proof. Let $x \in \widehat{x}$ and $y \in \widehat{y}$ be two revised affine forms as defined in (5.24) or (5.25). By definition, there exist two real numbers $e_x, e_y \in [-1, 1]$ such that

$$x = x_0 + \sum_{i=1}^{n} x_i \epsilon_i + e_x \epsilon_x, \quad y = y_0 + \sum_{i=1}^{n} y_i \epsilon_i + e_y \epsilon_y.$$

Let $\mathbf{e} \equiv [-1, 1]$. Since $a\epsilon_i^2 \in \frac{1}{2}(a + |a|\mathbf{e})$ for all $a \in \mathbb{R}$ and $i \in \{1, \ldots, n\}$, we have

$$\begin{aligned}
xy &= x_0 y_0 + \sum_{i=1}^{n} (x_0 y_i + x_i y_0)\epsilon_i + x_0 e_y \epsilon_y + y_0 e_x \epsilon_x + e_x e_y \epsilon_x \epsilon_y + \\
&\quad e_y \sum_{i=1}^{n} x_i \epsilon_y \epsilon_i + e_x \sum_{i=1}^{n} y_i \epsilon_x \epsilon_i + \sum_{1 \leq i,j \leq n;\ i \neq j} x_i y_j \epsilon_i \epsilon_i + \sum_{i=1}^{n} x_i y_i \epsilon_i^2 \\
&\in x_0 y_0 + \sum_{i=1}^{n} (x_0 y_i + x_i y_0)\epsilon_i + |x_0| e_y \mathbf{e} + |y_0| e_x \mathbf{e} + e_x e_y \mathbf{e} + \\
&\quad e_y \sum_{i=1}^{n} |x_i| \mathbf{e} + e_x \sum_{i=1}^{n} |y_i| \mathbf{e} + \sum_{1 \leq i,j \leq n;\ i \neq j} |x_i y_j| \mathbf{e} + \frac{1}{2} \sum_{i=1}^{n} (x_i y_i + |x_i y_i| \mathbf{e}) \\
&= (x_0 y_0 + \frac{1}{2} \sum_{i=1}^{n} x_i y_i) + \sum_{i=1}^{n} (x_0 y_i + x_i y_0)\epsilon_i + \\
&\quad \mathbf{e} \times \left(e_x e_y + e_y \sum_{i=0}^{n} |x_i| + e_x \sum_{i=0}^{n} |y_i| + \sum_{i=1}^{n} |x_i| \sum_{i=1}^{n} |y_i| - \frac{1}{2} \sum_{i=1}^{n} |x_i y_i| \right) \\
&\subseteq z_0 + \sum_{i=1}^{n} z_i \epsilon_i + e_z \mathbf{e} = \widehat{z} \qquad \text{(It follows from (5.24) or (5.25)).}
\end{aligned}$$

That is, $\forall x \in \widehat{x}, \forall y \in \widehat{y} : xy \in \widehat{z} \equiv \widehat{x} \times \widehat{y}$. The proof is hence completed. ∎

5.4.3. Division

The division $\hat{x}/\hat{y}$ can be written as $\hat{x} * (1/\hat{y})$, hence can be computed by a reciprocal and a multiplication. It is worth mentioning that Kolev [2002] proposed an improvement for computing the reciprocal $1/\hat{y}$, hence for computing $\hat{x}/\hat{y} := \hat{x} * (1/\hat{y})$. This has an interesting property that $\hat{x}/\hat{x} = 1$, which does not hold for interval arithmetic. Miyajima *et al.* [2003] also proposed new methods to compute $\hat{x}/\hat{y}$. However, these methods are too complicated to be presented here. The reader should find the details in [Kolev 2002; Miyajima *et al.* 2003].

5.4.4. Non-Affine Unary Operations

First, we recall the fundamental result in affine arithmetic, which is a result in *Chebyshev approximation theory* (see [Stolfi and de Figueiredo 1997, Theorem 2]).

Theorem 5.18. Let f be a bounded and *twice differentiable function* defined on some interval $[a, b]$ of which the second derivative f'' does not change sign inside $[a, b]$, where $a < b$. Let $f^{\mathrm{a}}(x) = \alpha x + \beta$ be its minimax affine approximation in $[a, b]$. Then

- The coefficient α is $(f(b) - f(a))/(b - a)$, the slope of the line $l(x)$ that interpolates the points $(a, f(a))$ and $(b, f(b))$;
- The maximum absolute error will occur twice (with the same sign) at the endpoints a and b of the range, and once (with the opposite sign) at every interior point c of $[a, b]$, where $f'(c) = \alpha$;
- The independent term β is such that $\alpha c + \beta = (f(c) + l(c))/2$ and the maximum absolute error is $\delta = |f(c) - l(c)|/2$.

Second, we propose the following constructive theorem, which is inspired by Theorem 5.18 and the related procedures in [Stolfi and de Figueiredo 1997], that serves as a basis to compute affine approximations of elementary univariate functions in a rigorous manner.

Theorem 5.19 (Chebyshev Affine Approximation). Let f be a *differentiable function* on $[a, b]$, where a and b are real numbers and $a \le b$; that is, $f \in \mathcal{C}^1([a, b])$. Denote $d_\alpha(x) \equiv f(x) - \alpha x$. Then

1. (a) If $\forall x \in [a, b] : \alpha \ge f'(x)$, then $\forall x \in [a, b] : \alpha x + d_\alpha(b) \le f(x) \le \alpha x + d_\alpha(a)$;
   (b) If $\forall x \in [a, b] : \alpha \le f'(x)$, then $\forall x \in [a, b] : \alpha x + d_\alpha(a) \le f(x) \le \alpha x + d_\alpha(b)$.
2. If f' is *continuous* and *monotone increasing* on $[a, b]$, we have
   (a) $\forall \alpha \in [f'(a), f'(b)], \exists c \in [a, b] : f'(c) = \alpha$;
   (b) Let $g : \mathbb{R} \to \mathbb{R}$ be a function such that $g(\alpha) = d_\alpha(c)$, then
   $$\forall x \in [a, b] : \alpha x + g(\alpha) \le f(x) \le \alpha x + \max\{d_\alpha(a), d_\alpha(b)\}.$$
3. If f' is *continuous* and *monotone decreasing* on $[a, b]$, we have
   (a) $\forall \alpha \in [f'(b), f'(a)], \exists c \in [a, b] : f'(c) = \alpha$;
   (b) Let $g : \mathbb{R} \to \mathbb{R}$ be a function such that $g(\alpha) = d_\alpha(c)$, then
   $$\forall x \in [a, b] : \alpha x + \min\{d_\alpha(a), d_\alpha(b)\} \le f(x) \le \alpha x + g(\alpha).$$

Proof. Hereafter, we prove the results for the cases 1a and 2. The proof is analogous for the other cases. Considering the case 1a, we have

$$\begin{aligned}
& f(x) - (\alpha x + d_\alpha(b)) \\
&= f(x) - (\alpha x + f(b) - \alpha b) \\
&= (f(x) - f(b)) - \alpha(x - b) \\
&= f'(\xi)(x-b) - \alpha(x-b) && \text{for some } \xi \in [x,b] \text{ (by the mean value theorem)} \\
&= (x-b)(f'(\xi) - \alpha) \geq 0 && \text{since } x \leq b, f'(\xi) \leq \alpha
\end{aligned}$$

By a similar argument, we also have $f(x) - (\alpha x + d_\alpha(a)) \leq 0$. Hence, the proof of the case 1a is completed.

Because $\alpha \in [f'(b), f'(a)]$ and f' is continuous on $[a,b]$ then there exists $c \in [a,b]$ as required by the case 2a. The proof is then completed for the case 2a. Here we give the proof of the case 2b. For all $x \in [a,b]$ we have

$$\begin{aligned}
& f(x) - (\alpha x + g(\alpha)) \\
&= f(x) - (\alpha x + f(c) - \alpha c) \\
&= (f(x) - f(c)) - \alpha(x - c) \\
&= f'(\xi)(x-c) - \alpha(x-c) && \text{for some } \xi \text{ between } x \text{ and } c \text{ (the mean value theorem)} \\
&= (x-c)(f'(\xi) - f'(c)) && \text{since } \alpha = f'(c) \\
&\geq 0 && \text{since } \xi \text{ is between } x \text{ and } c\text{, and } f' \text{ is increasing}
\end{aligned}$$

Moreover, if $x \in [a,c]$, we have

$$\begin{aligned}
& f(x) - (\alpha x + d_\alpha(a)) \\
&= f(x) - (\alpha x + f(a) - \alpha a) \\
&= (f(x) - f(a)) - \alpha(x - a) \\
&= f'(\eta)(x-a) - \alpha(x-a) && \text{for some } \eta \in [a,x] \text{ (by the mean value theorem)} \\
&= (x-a)(f'(\eta) - f'(c)) && \text{since } \alpha = f'(c) \\
&\leq 0 && \text{since } \eta \leq c,\ f' \text{ is monotone increasing}
\end{aligned}$$

By a similar argument, we have $f(x) - (\alpha x + d_\alpha(b)) \leq 0$ for all $x \in [c,b]$. Hence, we have $f(x) \leq \alpha x + \max\{d_\alpha(a), d_\alpha(b)\}$ for all $x \in [a,b]$. The proof is then completed. ■

To illustrate the usefulness of Theorem 5.19, we give in Table 5.1 the *derivative* f' and function g for some elementary operations. In Algorithm 5.1, we also propose an algorithm, called **SafeChebyshevApprox**$^\uparrow$, to find a safe Chebyshev affine approximation of a function $f \in \mathcal{C}^1([a,b])$ such that f' is monotone, when given the function g satisfying the conditions in Theorem 5.19. The following theorem guarantees the rigor of the **SafeChebyshevApprox**$^\uparrow$ algorithm, even in the presence of rounding errors.

Theorem 5.20. Let $\alpha\widehat{x} + \beta + \delta[-1,1]$ be the revised affine form produced by the **SafeChebyshevApprox**$^\uparrow$ algorithm in Algorithm 5.1, where $[a,b]$ is a closed interval containing $\widehat{x} \in \mathbb{A}$. Suppose $f \in \mathcal{C}^1([u,v])$ and f' is monotone on $[u,v]$, where $[u,v] \supseteq [a,b]$, such that $f'(v) \geq \lceil f'(b) \rceil$ if f' is monotone increasing or such that $f'(u) \geq \lceil f'(a) \rceil$ if f' is monotone decreasing. Then

$$\forall x \in \widehat{x} : f(x) \in \alpha\widehat{x} + \beta + \delta[-1,1]. \tag{5.26}$$

Table 5.1. Examples of functions $f \in \mathcal{C}^1([a,b])$ satisfying the conditions of Theorem 5.19.

$f(x)$	$[a,b]$ is a subset of	$f'(x)$	f'	$g(\alpha)$
x^2	$[-\infty,+\infty]$	$2x$	$\uparrow$	$-\alpha^2/4$
$\sqrt{x}$	$[0,+\infty]$	$1/(2\sqrt{x})$	$\downarrow$	$1/(4\alpha) : \alpha > 0$
e^x	$[-\infty,+\infty]$	e^x	$\uparrow$	$\alpha(1-\log\alpha) : \alpha > 0$
$\log x$	$]0,+\infty]$	$1/x$	$\downarrow$	$-(1+\log\alpha) : \alpha > 0$
$1/x$	$[-\infty,0[$	$-1/x^2$	$\downarrow$	$-2\sqrt{-\alpha} : \alpha < 0$
$1/x$	$]0,+\infty]$	$-1/x^2$	$\uparrow$	$2\sqrt{-\alpha} : \alpha < 0$
$x^n : n \geq 2$ is even	$[-\infty,+\infty]$	nx^{n-1}	$\uparrow$	$(1-n)\sqrt[n-1]{(\alpha/n)^n}$
$x^n : n \geq 3$ is odd	$[-\infty,0]$	nx^{n-1}	$\downarrow$	$(n-1)\sqrt[n-1]{(\alpha/n)^n} : \alpha \geq 0$
$x^n : n \geq 3$ is odd	$[0,+\infty]$	nx^{n-1}	$\uparrow$	$(1-n)\sqrt[n-1]{(\alpha/n)^n} : \alpha \geq 0$
$1/x^n : n \geq 2$ is even	$[-\infty,0[;\ ]0,+\infty]$	$-n/x^{n+1}$	$\uparrow$	$(n+1)\sqrt[n+1]{(-\alpha/n)^n}$
$1/x^n : n \geq 1$ is odd	$[-\infty,0[$	$-n/x^{n+1}$	$\downarrow$	$-(n+1)\sqrt[n+1]{(-\alpha/n)^n} : \alpha < 0$
$1/x^n : n \geq 1$ is odd	$]0,+\infty]$	$-n/x^{n+1}$	$\uparrow$	$(n+1)\sqrt[n+1]{(-\alpha/n)^n} : \alpha < 0$
$x^r : r \notin [0,1]$	$]0,+\infty]$	rx^{r-1}	$\uparrow$	$(1-r)(\alpha/r)^{(r/(r-1))} : \alpha r > 0$
$x^r : r \in (0,1)$	$]0,+\infty]$	rx^{r-1}	$\downarrow$	$(1-r)(\alpha/r)^{(r/(r-1))} : \alpha > 0$

Algorithm 5.1: The **SafeChebyshevApprox**$^{\uparrow}$ algorithm

Input: $\widehat{x} \in \mathbb{A}$, $f \in \mathcal{C}^1([a,b])$, f', g as defined in Theorem 5.19.
Output: a revised affine form $\alpha\widehat{x} + \beta + \delta[-1,1]$.

```
f_a := ⌊f(a)⌋; f_b := ⌈f(b)⌉; α := ⌈(f_b − f_a)/(b − a)⌉;
if f′ is monotone increasing on [a, b] then
    d_a := ⌈f(a)⌉ − ⌊αa⌋;
1   if α > ⌈f′(b)⌉ then
        d_min := ⌊f(b)⌋ − ⌈αb⌉; d_max := d_a;
2   else
        d_min := ⌊g(α)⌋; d_max := max{d_a, f_b − ⌊αb⌋};
    end
else                                    ◀ f′ is monotone decreasing on [a, b]
    d_b := ⌊f(b)⌋ − ⌈αb⌉;
    if α > ⌈f′(a)⌉ then
        d_min := d_b; d_max := ⌈f(a)⌉ − ⌊αa⌋;
    else
        d_min := min{f_a − ⌈αa⌉, d_b}; d_max := ⌈g(α)⌉;
    end
end
β := mid([d_min, d_max]); δ := rad([d_min, d_max]);
```

Proof. By the mean value theorem, there exists $c^* \in [a,b]$ such that

$$\begin{aligned} \alpha^* &\equiv f'(c^*) = (f(b) - f(a))/(b-a) \\ \Rightarrow \quad \alpha^* &\leq \lceil(\lceil f(b)\rceil - \lfloor f(a)\rfloor)/(b-a)\rceil = \alpha \\ \Rightarrow \quad \alpha &\geq f'(c^*) \geq \min\{f'(a), f'(b)\} \qquad \text{(since } f' \text{ is monotone).} \end{aligned}$$

Hereafter, we give a proof for the case that f' is monotone increasing. The proof for the case that f' is monotone decreasing is similar, where (a,u) and (b,v) exchange their roles with each

other. In case $\alpha > \lceil f'(b) \rceil$, we have $\alpha > f'(x)$ for all $x \in [a, b]$. Hence, according to the case 1a of Theorem 5.19, for all $x \in [a, b]$, we have

$$\begin{aligned} & \alpha x + (f(b) - \alpha b) = \alpha x + d_\alpha(b) \le f(x) \le \alpha x + d_\alpha(a) = \alpha x + (f(a) - \alpha a) \\ \Rightarrow\ & \alpha x + d_{\min} \le \alpha x + d_\alpha(b) \le f(x) \le \alpha x + d_\alpha(a) \le \alpha x + d_{\max}, \end{aligned}$$

because $d_{\min} = \lfloor f(b) \rfloor - \lceil \alpha b \rceil$ and $d_{\max} = \lceil f(a) \rceil - \lfloor \alpha a \rfloor$ (see Line 1 in Algorithm 5.1). In case $\alpha \le f'(b)$, it follows from the case 2 of Theorem 5.19 and Line 2 in Algorithm 5.1 that

$$\alpha x + d_{\min} \le \alpha x + g(\alpha) \le f(x) \le \max\{d_\alpha(a), d_\alpha(b)\} \le \alpha x + d_{\max}.$$

In the rest, we consider the case $f'(b) < \alpha \le \lceil f'(b) \rceil$. According to the case 1a of Theorem 5.19, for all $x \in [a, b]$, we have $\alpha x + d_\alpha(b) \le f(x) \le \alpha x + d_\alpha(a)$. Moreover, applying the case 2 of Theorem 5.19 to $[b, v]$, we have

$$\begin{aligned} & x \in [b, v] : \alpha x + g(\alpha) \le f(x) \\ \Leftrightarrow\ & x \in [b, v] : g(\alpha) \le f(x) - \alpha x \\ \Rightarrow\ & g(\alpha) \le f(b) - \alpha b = d_\alpha(b). \end{aligned}$$

Therefore, for all $x \in [a, b]$, we have

$$\begin{aligned} & \alpha x + g(\alpha) \le f(x) \le \alpha x + d_\alpha(a) \le \max\{\alpha x + d_\alpha(a), \alpha x + d_\alpha(b)\} \\ \Rightarrow\ & \alpha x + d_{\min} \le f(x) \le \alpha x + d_{\max}. \text{ (See Line 2 in Algorithm 5.1.)} \end{aligned}$$

As a result, in all cases, we have $\alpha x + d_{\min} \le f(x) \le \alpha x + d_{\max}$ for all $x \in [a, b]$, thus, $\forall x \in \widehat{x} : f(x) \in \alpha \widehat{x} + \beta + \delta[-1, 1]$. The proof is hence completed. ■

Algorithm 5.2: The **SafeChebyshevApprox**$^\downarrow$ algorithm

Input: $\widehat{x} \in \mathbb{A}$, $f \in \mathcal{C}^1([a, b])$, f', g as defined in Theorem 5.19.
Output: a revised affine form $\alpha \widehat{x} + \beta + \delta[-1, 1]$.

```
    f_a := ⌈f(a)⌉; f_b := ⌊f(b)⌋; α := ⌊(f_b − f_a)/(b − a)⌋;
    if f' is monotone increasing on [a, b] then
        d_b := ⌈f(b)⌉ − ⌊αb⌋;
1       if α < ⌊f'(a)⌋ then
            d_min := ⌊f(a)⌋ − ⌈αa⌉; d_max := d_b;
2       else
            d_min := ⌊g(α)⌋; d_max := max{f_a − ⌊αa⌋, d_b};
        end
    else                                        ◄ f' is monotone decreasing on [a, b]
        d_a := ⌊f(a)⌋ − ⌈αa⌉;
        if α < ⌊f'(b)⌋ then
            d_min := d_a; d_max := ⌈f(b)⌉ − ⌊αb⌋;
        else
            d_min := min{d_a, f_b − ⌈αb⌉}; d_max := ⌈g(α)⌉;
        end
    end
    β := mid([d_min, d_max]); δ := rad([d_min, d_max]);
```

The rigor of the **SafeChebyshevApprox**$^\uparrow$ algorithm requires that $f'(v) \ge \lceil f'(b) \rceil$ if f' is monotone increasing, or $f'(u) \ge \lceil f'(a) \rceil$ if f' is monotone decreasing (see Theorem 5.20). In

very special cases, the domain of f may not be extended in the required side, we can use the **SafeChebyshevApprox**$^{\downarrow}$ algorithm in Algorithm 5.2 as an alternative. These two algorithms are sufficient for the standard elementary operations.

The following theorem states the rigor of the **SafeChebyshevApprox**$^{\downarrow}$ algorithm.

Theorem 5.21. Let $\alpha\widehat{x}+\beta+\delta[-1,1]$ be the revised affine form produced by the **SafeChebyshevApprox**$^{\downarrow}$ algorithm in Algorithm 5.2, where $[a,b]$ is a closed interval containing $\widehat{x} \in \mathbb{A}$. Suppose $f \in \mathcal{C}^1([u,v])$ and f' is monotone on $[u,v]$, where $[u,v] \supseteq [a,b]$, such that $f'(u) \leq \lfloor f'(a) \rfloor$ if f' is monotone increasing or such that $f'(v) \leq \lfloor f'(b) \rfloor$ if f' is monotone decreasing. Then

$$\forall x \in \widehat{x} : f(x) \in \alpha\widehat{x} + \beta + \delta[-1,1]. \tag{5.27}$$

Proof. The proof is similar to the proof of Theorem 5.20, but (a,u) and (b,v) exchange their roles with each other. For example, the interval $[b,v]$ is replaced with the interval $[u,a]$, in the last part of the proof of Theorem 5.20. Note that the first result is replaced with $\alpha \leq \alpha^* = f'(c^*) \leq \max\{f'(a), f'(b)\}$. ■

5.5. Abstraction of Inclusion Concepts

5.5.1. Inclusion Representations

Hereafter, we generalize the concepts related to the concept of an *interval form* (see the definitions 2.80, 2.81, and 5.9) into a common view in order to facilitate presenting a novel combination scheme for constraint propagation in Chapter 7.

Definition 5.22 (Inclusion Representation). Given a set $\mathcal{A}$. A couple $\mathcal{I} \equiv (\mathcal{R}, \mu)$, where $\mathcal{R}$ is a nonempty set and μ is a function from $\mathcal{R}$ to $2^{\mathcal{A}} \cup \{\emptyset\}$, is called an *inclusion representation* of $\mathcal{A}$ if there exists a function $\zeta : 2^{\mathcal{A}} \cup \{\emptyset\} \to \mathcal{R}$ such that $\mu(\zeta(\emptyset)) = \emptyset$ and, for all $S \subseteq \mathcal{A}$, we have $S \subseteq \mu(\zeta(S))$. In this case, each $T \in \mathcal{R}$ is called a *representation object* in $\mathcal{I}$, $\mathcal{R}$ is called the *representation set* in $\mathcal{I}$, ζ is called a *representing function* of $\mathcal{I}$, and μ is called the *evaluating function* of $\mathcal{I}$.

The function ζ in Definition 5.22 is to say that each subset of $\mathcal{A}$ can be "included" in at least one representation object in $\mathcal{R}$. The next definition identifies a special class of inclusion representations for use in practice, in particular, in Chapter 7.

Definition 5.23 (Real Inclusion Representation, Real Representation). Let $\mathcal{I} \equiv (\mathcal{R}, \mu)$ be an inclusion representation of $\mathbb{R}$. It is called a *real inclusion representation* (of the real set $\mathbb{R}$) if each representation object $T \in \mathcal{R}$ is a tuple consisting of real numbers, and the value of μ at T can be written as

$$\mu(T) \equiv \{f_T(V_T) \mid V_T \in D_T[V_T]\}, \tag{5.28}$$

where f_T is a real-valued function (with T as a tuple of parameters) on a finite sequence V_T of variables taking values in the projection $D_T[V_T]$ of D_T on V_T (Definition 2.24), where D_T is defined on V_T and other auxiliary variables. The representation (5.28) is called a *real representation* of μ; f_T is called the *real evaluation generator* of T.

The domains in D_T can be explicitly given by constant domains such as an interval $[a, b]$, or implicitly given by constraints. Note that the set of variables of D_T must contain V_T. If D_T is given by

$$D_T = \bigcap_{i=1}^{m} \{f_{i,T}(V_{i,T}) \mid V_{i,T} \in D_{i,T}[V_{i,T}]\}, \tag{5.29}$$

and f_T is an identity function, we then have

$$\mu(T) \equiv \{x \mid x \in D_T[x]\} = \bigcap_{i=1}^{m} \{f_{i,T}(V_{i,T}) \mid V_{i,T} \in D_{i,T}[V_{i,T}]\}. \tag{5.30}$$

This seems to be a generalized form of (5.28), but it is in fact equivalent to (5.28). Therefore, in some cases we can use the form (5.30) directly instead of (5.28), for short. Hereafter, we give some examples to illustrate the concept of a real inclusion representation.

Example 5.24. Obviously, the standard representation of reals is directly equivalent to, although not exactly, a real inclusion representation of $\mathbb{R}$; where $T = (x)$, $V_T = (x)$, f_T is an identity function, and $D_T = \{x\}$. ♣

Example 5.25. It is easy to see that the representation of intervals in the form

$$a \leq x \leq b \tag{5.31}$$

is equivalent to a real inclusion representation of the form (5.28), called the *interval representation*, by defining that

$$\begin{aligned} T &= (a, b) \in \mathbb{R}^2, && (5.32a) \\ V_T &= (x), && (5.32b) \\ D_T &= [a, b], && (5.32c) \\ f_T(V_T) &= x, && (5.32d) \\ \mu(T) &\equiv \{x \mid x \in [a, b]\}. && (5.32e) \end{aligned}$$

The function f_T in (5.32) is an identity function. ♣

Example 5.26. The union form of intervals can also be viewed as a real inclusion representation. For example, the union $\bigcup_{i=1}^{m} [a_i, b_i]$ can be interpreted by

$$\begin{aligned} T &= (a_1, b_1, \ldots, a_m, b_m), && (5.33a) \\ V_T &= (x), && (5.33b) \\ D_T &= \bigcup_{i=1}^{m} [a_i, b_i], && (5.33c) \\ f_T(V_T) &= x, && (5.33d) \\ \mu(T) &\equiv \left\{ x \;\middle|\; x \in \bigcup_{i=1}^{m} [a_i, b_i] \right\}. && (5.33e) \end{aligned}$$

The function f_T in (5.33) is an identity function. ♣

Example 5.27. The affine forms of the form (2.31), namely $\hat{x} \equiv x_0 + x_1\epsilon_1 + \cdots + x_n\epsilon_n$, can also be seen as a real inclusion representation of the form (5.28), called the *standard affine representation*, by defining that

$$T = (x_0, \ldots, x_n), \tag{5.34a}$$
$$V_T = (\epsilon_1, \ldots, \epsilon_n), \tag{5.34b}$$
$$D_T = [-1, 1]^n, \tag{5.34c}$$
$$f_T(V_T) = x_0 + \sum_{i=1}^{n} x_i\epsilon_i, \tag{5.34d}$$
$$\mu(T) \equiv \left\{ x_0 + \sum_{i=1}^{n} x_i\epsilon_i \;\middle|\; (\epsilon_1, \ldots, \epsilon_n) \in [-1, 1]^n \right\}. \tag{5.34e}$$

The function f_T in (5.34) is a *linear function* on V_T. Another real representation of the form (5.28) for the above real inclusion representation is defined as follows:

$$T = (x_0, \ldots, x_n), \tag{5.35a}$$
$$V_T = (x), \tag{5.35b}$$
$$D_T = \left\{ x_0 + \sum_{i=1}^{n} x_i\epsilon_i \;\middle|\; (\epsilon_1, \ldots, \epsilon_n) \in [-1, 1]^n \right\}, \tag{5.35c}$$
$$f_T(V_T) = x, \tag{5.35d}$$
$$\mu(T) \equiv \{x \mid x \in D_T\}, \tag{5.35e}$$

where D_T is implicitly given via its variables. Although both (5.34) and (5.35) represent the same set, in the latter, the function f_T is an identity function. ♣

Example 5.28. Similarly to the affine forms, the revised affine forms defined by (5.19) can also be seen as a real inclusion representation of the form (5.28), called the *revised affine representation*, by defining that

$$T = (x_0, \ldots, x_n, e_x), \tag{5.36a}$$
$$V_T = (\epsilon_1, \ldots, \epsilon_n, \epsilon_x), \tag{5.36b}$$
$$D_T = [-1, 1]^{n+1}, \tag{5.36c}$$
$$f_T(V_T) = x_0 + \sum_{i=1}^{n} x_i\epsilon_i + e_x\epsilon_x, \tag{5.36d}$$
$$\mu(T) \equiv \left\{ x_0 + \sum_{i=1}^{n} x_i\epsilon_i + e_x\epsilon_x \;\middle|\; (\epsilon_1, \ldots, \epsilon_n, \epsilon_x) \in [-1, 1]^{n+1} \right\}. \tag{5.36e}$$

The function f_T in (5.36) is a *linear function* on $(n + 1)$ variables in V_T. ♣

Example 5.29. The *Kolev affine form* (see Definition 2.83), a generalization of interval and affine form, is also a real inclusion representation. Namely, the Kolev affine form (2.40) can be defined as an affine function on the variables κ_i:

$$\tilde{x} = c_x + \sum_{i=1}^{n} x_i\kappa_i + \kappa_x, \quad \kappa_i \in \mathbf{v}_i, \; \kappa_x \in \mathbf{v}_x, \tag{5.37}$$

where $\mathbf{v}_i \equiv [-v_i, v_i]$ (for all $i = 1, \dots, n$) and $\mathbf{v}_x \equiv [-v_x, v_x]$ are symmetric intervals, x_i (for all $i = 1, \dots, n$) are real coefficients, and $c_x \in \mathbb{R}$. Similarly to revised affine forms, Kolev affine forms can also be interpreted as a real inclusion representation (5.28), called the *Kolev affine representation*, by defining that

$$T = (c_x, x_1, \dots, x_n, v_1, \dots, v_n, v_x), \tag{5.38a}$$
$$V_T = (\kappa_1, \dots, \kappa_n, \kappa_x), \tag{5.38b}$$
$$D_T = [-v_1, v_1] \times \cdots \times [-v_n, v_n] \times [-v_x, v_x], \tag{5.38c}$$
$$f_T(V_T) = c_x + \sum_{i=1}^{n} x_i \kappa_i + \kappa_x, \tag{5.38d}$$
$$\mu(T) \equiv \left\{ c_x + \sum_{i=1}^{n} x_i \kappa_i + \kappa_x \;\middle|\; (\kappa_1, \dots, \kappa_n, \kappa_x) \in D_T \right\}. \tag{5.38e}$$

The function f_T in (5.38) is a *linear function* on $(n+1)$ variables in V_T. ♣

Example 5.30. Hansen's generalized interval [Hansen 1975] is given as follows

$$\tilde{x} = [\underline{c}_x, \overline{c}_x] + \sum_{i=1}^{n} [\underline{x}_i, \overline{x}_i] \kappa_i, \tag{5.39}$$

where the notations are the same as in the Kolev affine form (5.37). Hansen's generalize interval can be interpreted as a real inclusion representation, called the *Hansen interval representation*, by defining that

$$T = (\underline{c}_x, \overline{c}_x, \underline{x}_1, \overline{x}_1, \dots, \underline{x}_n, \overline{x}_n, v_1, \dots, v_n), \tag{5.40a}$$
$$V_T = (\kappa_1, \dots, \kappa_n, c_x, x_1, \dots, x_n), \tag{5.40b}$$
$$D_T = [-v_1, v_1] \times \cdots \times [-v_n, v_n] \times [-v_x, v_x] \times [\underline{x}_1, \overline{x}_1] \times \dots [\underline{x}_n, \overline{x}_n], \tag{5.40c}$$
$$f_T(V_T) = c_x + \sum_{i=1}^{n} x_i \kappa_i, \tag{5.40d}$$
$$\mu(T) \equiv \left\{ c_x + \sum_{i=1}^{n} x_i \kappa_i \;\middle|\; (\kappa_1, \dots, \kappa_n, c_x, x_1, \dots, x_n) \in D_T \right\}. \tag{5.40e}$$

Note that the function f_T in (5.40) is a quadratic function on $(2n+1)$ variables in V_T. ♣

Example 5.31. Linear relaxations and (convex) polyhedral enclosures can also be viewed as real inclusion representations. Indeed, they are given as the intersection of m half-spaces

$$H_i \equiv \{(x_1, \dots, x_n) \mid a_{i0} + \sum_{j=1}^{n} a_{ij} x_j \le 0\} \quad (\text{for } i = 1, \dots, m),$$

and is often restricted to a domain $\mathbf{B}$ that is usually a box. We can therefore obtain a real inclusion representation (5.30), called the *linear relaxation representation*, by defining that

$$T = (a_{10}, \dots, a_{1n}, \dots, a_{m0}, \dots, a_{mn}), \tag{5.41a}$$
$$V_T = (x_k), \quad \text{for some } k \in \{1, \dots, n\} \tag{5.41b}$$
$$D_T = \mathbf{B} \cap \bigcap_{i=1}^{m} H_i, \tag{5.41c}$$

$$f_T(V_T) = x_k, \tag{5.41d}$$

$$\mu(T) \equiv \{x_k \mid x_k \in D_T[x_k]\} = \mathbf{B} \cap \bigcap_{i=1}^{m} H_i. \tag{5.41e}$$

This is the intersection of $\mathbf{B}$ and the convex polyhedron $\bigcap_{i=1}^{m} H_i$. ♣

By a similar argument, one could see that the new concept of an inclusion representation covers almost all existing inclusions for real numbers. However, it does not introduce a new inclusion technique, but provides an abstraction of different existing inclusion techniques.

Note 5.32. In practice, a representation object T in the above affine forms often contains many zero coefficients; hence, we should store only nonzero coefficients and their indices instead of all coefficients. For example, an affine form $0.1 + 2.1\epsilon_2 + 9.1\epsilon_9$ should be stored in T by $(0.1, 2.1, 9.1; 2, 9)$ instead of $(0.1, 0.0, 2.1, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 9.1)$.

The following theorems characterize properties of inclusion representations.

Theorem 5.33. Let $\mathcal{A}$ be a nonempty set and $\mathcal{A}'$ a subset of $\mathcal{A}$. Suppose $\mathcal{I} \equiv (\mathcal{R}, \mu)$ is an inclusion representation of $\mathcal{A}$. Then the couple $\mathcal{I}' \equiv (\mathcal{R}', \mu')$ is an inclusion representation of $\mathcal{A}'$, where

$$\mathcal{R}' := \{T \in \mathcal{R} \mid \mu(T) \neq \emptyset \Rightarrow \mu(T) \cap \mathcal{A}' \neq \emptyset\}, \tag{5.42a}$$

$$\mu'(T) := \mu(T) \cap \mathcal{A}' \quad \text{for all } T \in \mathcal{R}'. \tag{5.42b}$$

Proof. By Definition 5.22, there exists a representing function ζ of $\mathcal{I}$ that maps from $2^{\mathcal{A}} \cup \{\emptyset\}$ to $\mathcal{R}$. We define a function $\zeta' : 2^{\mathcal{A}'} \cup \{\emptyset\} \to \mathcal{R}$ simply by the rule $\zeta'(S) := \zeta(S)$ for all $S \in 2^{\mathcal{A}'} \cup \{\emptyset\}$. It follows from (5.42) that $\zeta'(S) \in \mathcal{R}'$ because $\mu(\zeta'(S)) = \mu(\zeta(S)) \supseteq S$ and $\mu(\zeta'(\emptyset)) = \mu(\zeta(\emptyset)) = \emptyset$. Hence, ζ' is a function from $2^{\mathcal{A}'} \cup \{\emptyset\}$ to $\mathcal{R}'$ as required. Moreover, for all $S \in 2^{\mathcal{A}'} \cup \{\emptyset\}$, we have $\mu'(\zeta'(S)) = \mu(\zeta(S)) \cap \mathcal{A}' \supseteq S$. ■

Theorem 5.34. Let $\mathcal{I} \equiv (\mathcal{R}, \mu)$ and $\mathcal{I}' \equiv (\mathcal{R}', \mu')$ be inclusion representations of two sets $\mathcal{A}$ and $\mathcal{A}'$, respectively. Then $\mathcal{I}'' \equiv (\mathcal{R} \times \mathcal{R}', (\mu, \mu')^{\mathrm{T}})$ is an inclusion representation of $\mathcal{A} \times \mathcal{A}'$. We denote $\mathcal{I}''$ by $\mathcal{I} \times \mathcal{I}'$.

Proof. By Definition 5.22, there exist two representing functions ζ and ζ' of $\mathcal{I}$ and $\mathcal{I}'$, respectively. The function $\zeta'' = (\zeta, \zeta')^{\mathrm{T}}$ is a representing function of $\mathcal{I}''$. ■

Notation 5.35. For simplicity, we will use $\mathbb{I}$, $\hat{\mathbb{A}}$, and $\mathbb{A}$ to refer to the representation sets in the interval inclusion representation defined by (5.32), the standard affine representation defined by (5.34), and the revised affine representation defined by (5.36), respectively, when no confusion may arise. As a result, we will use $0.1+2.1\epsilon_2+9.1\epsilon_9$ to refer the representation object $(0.1, 0.0, 2.1, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 9.1)$ in the standard affine representation $(\hat{\mathbb{A}}, \mu_{\hat{\mathbb{A}}})$ defined by (5.34). Similarly, we will also use the form $[1, 3]$ to refer to the representation object $(1, 3)$ in the interval representation $(\mathbb{I}, \mu_{\mathbb{I}})$ defined by (5.32).

5.5.2. Inclusion Functions

Next, we generalize the concept of an interval form to accept the concept of an inclusion representation.

Definition 5.36 (Inclusion Function of Functions). Given two nonempty sets X, Y and a function $f : X \to Y$. Let $\mathcal{I}_X \equiv (\mathcal{R}_X, \mu_X)$ and $\mathcal{I}_Y \equiv (\mathcal{R}_Y, \mu_Y)$ be two inclusion representations of two supersets $X^+ \supseteq X$ and $Y^+ \supseteq Y$, respectively. A function $F : \mathcal{R}_X \to \mathcal{R}_Y$ is called an *inclusion function* of f (or for f) if we have

$$\forall x \in X, \forall T \in \mathcal{R}_X : x \in \mu_X(T) \Rightarrow f(x) \in \mu_Y(F(T)). \tag{5.43}$$

One can see that the inclusion function defined in [Jaulin *et al.* 2001, p. 27] for intervals is a special case of the new inclusion function concept in Definition 5.36. The new inclusion function concept can also be extended to multifunctions similarly to Definition 5.9 and, in particular, can be applied to extended functions.

Definition 5.37 (Inclusion Function of Multifunctions). Given two nonempty sets X, Y and a multifunction $f : X \to Y$. Let $\mathcal{I}_X \equiv (\mathcal{R}_X, \mu_X)$ and $\mathcal{I}_Y \equiv (\mathcal{R}_Y, \mu_Y)$ be two inclusion representations of two supersets $X^+ \supseteq X$ and $Y^+ \supseteq Y$, respectively. A function $F : \mathcal{R}_X \to \mathcal{R}_Y$ is called an *inclusion function* of f (or for f) if we have

$$\forall x \in X, \forall T \in \mathcal{R}_X : x \in \mu_X(T) \Rightarrow f(x) \subseteq \mu_Y(F(T)). \tag{5.44}$$

It is easy to verify that both the condition (5.43) and the condition (5.44) can be replaced with an equivalent condition, called the *generalized inclusion property*,

$$\forall S \subseteq X, \forall T \in \mathcal{R}_X : S \subseteq \mu_X(T) \Rightarrow f(x) \subseteq \mu_Y(F(T)). \tag{5.45}$$

All interval forms of a function (or multifunction) f are inclusion functions of f when considering the interval representation (5.32) (see Section A.2 for typical interval forms). The new inclusion function concept can also be extended to relations in a way similar to Definition 2.81.

Definition 5.38 (Inclusion Relation). Let X be a nonempty set and $\mathcal{I}_X \equiv (\mathcal{R}_X, \mu_X)$ an inclusion representation of a superset $X^+ \supseteq X$. Suppose R is a relation on X; that is, $R \subseteq X$. A relation $R_X \subseteq \mathcal{R}_X$ satisfying the condition

$$\forall r \in R, \forall T \in \mathcal{R}_X : r \in T \Rightarrow T \in R_X \tag{5.46}$$

is called an *inclusion relation* of R.

The result of Theorem 2.82 can be extended to inclusion relations as follows.

Theorem 5.39. Let $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ be a function, $\mathbf{b} \in \mathbb{I}^m$, and $\mathcal{I} \equiv (\mathcal{R}, \mu)$ a real inclusion representation. If $F : \mathcal{R}^n \to \mathcal{R}^m$ is an inclusion function of f. Then $R_{\mathcal{I}} \equiv \{T \in \mathcal{R}^n \mid \mu(F(T)) \cap \mathbf{b} \neq \emptyset\}$ is an inclusion relation of the relation $R \equiv \{x \in D \mid f(x) \in \mathbf{b}\}$.

Proof. For all $r \in R$ and $T \in \mathcal{R}^n$, if $r \in T$, then $f(r) \in \mathbf{b}$, thus $\mu(F(T)) \cap \mathbf{b} \ni f(x)$. Hence, $T \in R_{\mathcal{I}}$. The notation $\mu(F(T))$ means performing μ in a componentwise manner. ∎

The result of Theorem 5.10 can also extended to inclusion functions as follows.

Theorem 5.40. Let $F : X \to Y$ be an extended function over X of a function $f : X' \subseteq X \to Y$. Let $\mathcal{I}_X \equiv (\mathcal{R}_X, \mu_X)$ and $\mathcal{I}_Y \equiv (\mathcal{R}_Y, \mu_Y)$ be inclusion representations of X and Y, respectively. Then each inclusion function $[F] : \mathcal{R}_X \to \mathcal{R}_Y$ of F is an inclusion function of f.

Proof. For every $x \in X'$ and every object $T \in \mathcal{R}_X$ such that $x \in \mu_X(T)$, we have $f(x) \in \{f(x)\} = F(x) \subseteq [F](T)$. Hence, $[F]$ is an inclusion function of f. ∎

In practice, one often extends real-valued functions in a natural way to evaluate the ranges of the real-valued functions. Hereafter, we define the concept of a natural extension for use with the inclusion function concept in Definition 5.36.

Definition 5.41 (Natural Extension). Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a factorable function using a finite set E of elementary operations defined over $\mathbb{R}$. Suppose $\mathcal{I} = (\mathcal{R}, \mu)$ is an inclusion representation of $\mathbb{R}$ such that there exists a set $E_{\mathcal{R}}$ of elementary operations defined over $\mathcal{R}$ and a bijection mapping $\eta : E \to E_{\mathcal{R}}$. We then call $\diamond \in E$ the real-valued counterpart of $\eta(\diamond)$ and vice versa. A function $\mathbf{f} : \mathcal{R}^n \to \mathcal{R}^m$ is called the *natural extension* of f in $\mathcal{I}$ (using operations in $E_{\mathcal{R}}$) if $\mathbf{f}$ is constructed from the recursive composition of f by replacing each real variable (resp., each constant) by a variable taking values (resp., a constant) in $\mathcal{R}$, and replacing each occurrence of an elementary operation $e \in E$ by the corresponding occurrence of $\eta(e)$. If $\mathbf{f}$ is also an inclusion function of f, we call $\mathbf{f}$ the *natural inclusion function* of f.

The natural interval form (see Definition A.1) is an example of the natural inclusion functions, when considering the interval representation (5.32).

Example 5.42. In the composition of the real-valued function $f(x) = x * (x - 1)$, the real variable x occurs twice. The set of elementary operations used in the composition of f is $E = \{-, *\}$. The exact range $f([0, 1]) = [-0.25, 0]$. The natural extension of f in the interval representation $(\mathbb{I}, \mu_{\mathbb{I}})$ defined by (5.32) is a function $\mathbf{f} : \mathbb{I} \to \mathbb{I}$ defined as $\mathbf{f}(\mathbf{x}) = \mathbf{x} * (\mathbf{x} - \mathbf{1})$, where $\mathbf{1} = [1, 1] = 1$. Consider the representation object $T = (0, 1)$ in the interval representation $(\mathbb{I}, \mu_{\mathbb{I}})$, which corresponds to the interval $[0, 1]$. In interval arithmetic, the value of $\mathbf{f}$ at T is

$$\mathbf{f}(T) = [0, 1] * ([0, 1] - \mathbf{1}) = [0, 1] * [-1, 0] = [-1, 0].$$

We have $f([0, 1]) \subseteq \mathbf{f}([0, 1])$. Moreover, we can prove that $\forall x \in \mathbf{x} \in \mathbb{I} : f(x) \in \mathbf{f}(\mathbf{x})$. Hence, $\mathbf{f}$ is the natural inclusion function of f in the interval arithmetic representation. ♣

Example 5.43. The natural extension of the real-valued function $f(x) = x * (x - 1)$ in the standard affine representation is the function $\hat{f}(\hat{x}) = \hat{x} * (\hat{x} - \hat{1})$, where $\hat{1} = 1$. A representation object $T = (0.5, 0.5)$ in the standard affine representation $(\hat{\mathbb{A}}, \mu_{\hat{\mathbb{A}}})$ defined by (5.34), which corresponds to the affine form $0.5 + 0.5\epsilon_1$, has the real evaluation at T

$$\mu_{\hat{\mathbb{A}}}(T) = \{0.5 + 0.5\epsilon_1 \mid \epsilon_1 \in [-1, 1]\} = [0, 1].$$

In affine arithmetic, the value of $\hat{f}$ at $T = 0.5 + 0.5\epsilon_1$ is

$$\begin{aligned}\hat{f}(T) &= (0.5 + 0.5\epsilon_1) * ((0.5 + 0.5\epsilon_1) - 1)\\ &= (0.5 + 0.5\epsilon_1) * (-0.5 + 0.5\epsilon_1)\\ &= -0.25 + 0.25\epsilon_{\text{new}},\end{aligned}$$

where ϵ_{new} is a new noise variable taking its value in $[-1, 1]$. Hence, $\hat{f}(T)$ has the real evaluation $\mu_{\hat{\mathbb{A}}}(-0.25 + 0.25\epsilon_{\text{new}}) = [-0.5, 0]$. We can prove that the natural extension $\hat{f}$ is the natural inclusion function of f in $(\hat{\mathbb{A}}, \mu_{\hat{\mathbb{A}}})$, w.r.t. affine arithmetic.

Note that in revised affine arithmetic, $\hat{f}(T) = -0.125 + 0.125\epsilon_{\text{new}}$. Hence, it has the real evaluation $\mu_{\mathbb{A}}(-0.125 + 0.125\epsilon_{\text{new}}) = [-0.25, 0]$. This evaluation provides a tighter enclosure than what provided by affine arithmetic. ♣

Example 5.44. The natural extension of the real-valued function $f(x) = x * (x - 1)$ in the revised affine representation is the function $\widehat{f}(\widehat{x}) = \widehat{x} * (\widehat{x} - \widehat{1})$, where $\widehat{1} = 1$. A representation object $T = (0.5, 0.5, 0.0)$ in the revised affine representation $(\mathbb{A}, \mu_{\mathbb{A}})$ defined by (5.36), which corresponds to the revised affine form $0.5 + 0.5\epsilon + 0[-1, 1]$, has the real evaluation at T

$$\mu_{\mathbb{A}}(T) = \{0.5 + 0.5\epsilon + 0[-1, 1] \mid \epsilon \in [-1, 1]\} = [0, 1].$$

In revised affine arithmetic, The value of $\widehat{f}$ at T is

$$\begin{aligned}\widehat{f}(T) &= (0.5 + 0.5\epsilon) * ((0.5 + 0.5\epsilon) - 1)\\ &= (0.5 + 0.5\epsilon) * (-0.5 + 0.5\epsilon)\\ &= -0.125 + 0.125[-1, 1].\end{aligned}$$

Hence, $\widehat{f}(T)$ has the real evaluation $\mu_{\mathbb{A}}(-0.125 + 0.125[-1, 1]) = [-0.25, 0]$. We can prove that the natural extension $\widehat{f}$ is the natural inclusion function of f in the revised affine representation, w.r.t. revised affine arithmetic. ♣

Next, we define a concept of converting from an inclusion representation to another inclusion representation without loss of the inclusion property.

Definition 5.45 (Inclusion Conversion). Let $\mathcal{I}_1 = (\mathcal{R}_1, \mu_1)$ and $\mathcal{I}_2 = (\mathcal{R}_2, \mu_2)$ be two inclusion representations of the same set. A function $c : \mathcal{R}_1 \to \mathcal{R}_2$ is called an *inclusion converter* from $\mathcal{I}_1$ to $\mathcal{I}_2$ if for all $T \in \mathcal{R}_1$ we have $\mu_1(T) \subseteq \mu_2(c(T))$.

Example 5.46. Converting the object $3 + 2\epsilon_1 + 1\epsilon_2$ of the standard affine representation into the interval representation, we get interval $[0, 6]$. When converting $[0, 6]$ into the standard affine representation; however, we get $3 + 3\epsilon_{\text{new}}$, where ϵ_{new} is a new noise variable. ♣

Theorem 5.47 (Composite Inclusion Function). Let $\mathcal{I}_X = (\mathcal{R}_X, \mu_X)$, $\mathcal{I}_Y = (\mathcal{R}_Y, \mu_Y)$ and $\mathcal{I}_Z = (\mathcal{R}_Z, \mu_Z)$ be inclusion representations of three sets X, Y and Z, respectively. If $F : \mathcal{R}_X \to \mathcal{R}_Y$ and $G : \mathcal{R}_Y \to \mathcal{R}_Z$ are inclusion functions of two functions $f : X \to Y$ and $g : Y \to Z$ respectively, then the composite function $G \circ F$ is an inclusion function of the composite function $g \circ f$.

Proof. Let $T \in \mathcal{R}_X$, $U = F(T)$, $V = G(U)$ and $S \subseteq X \cap \mu_X(T)$. By Definition 5.36, we have $f(S) \subseteq \mu_Y(U)$, thus, $f(S) \subseteq Y \cap \mu_Y(U)$. Therefore, also by Definition 5.36, we have $g(f(S)) \subseteq \mu_Z(G(U)) = \mu_Z(V)$. That is, $g \circ f(S) \subseteq \mu_Z(G \circ F(T))$ holds for every $S \subseteq X \cap \mu_X(T)$. That is, $G \circ F$ is an inclusion function of $g \circ f$. ■

Corollary 5.48. *Let $\mathcal{I} = (\mathcal{R}, \mu)$ be an inclusion representation of $\mathbb{R}$. If elementary operations defined over $\mathcal{R}$ are inclusion functions of their counterparts over $\mathbb{R}$, then all factorable functions built over $\mathcal{R}$ by using these elementary operations are also inclusion functions of their counterparts over $\mathbb{R}$.* ✠

Proof. Corollary 5.48 is an obvious consequence of Theorem 5.47. Therefore, the proof is omitted. ■

In implementation, the elementary operations in interval arithmetic and affine arithmetic are constructed to be inclusion functions of their real-valued counterparts. Therefore, it follows from Corollary 5.48 that all the factorable operations/functions defined in interval arithmetic (or affine arithmetic) by using these elementary operations are also inclusion functions of their real-valued counterparts.

5.6. Conclusion

Our contributions in this chapter include the following (minor and major) changes:

1. In Section 5.2, we present a consistent way to extend the functions that are only defined on subsets of the real set, for specific situations in constraint propagation. This serves as a basis for the results in Section 5.3, Section 5.5, Chapter 6, and Chapter 7.

2. In Section 5.3, we revise the concept of an interval form for extended functions. The revision of the division in interval arithmetic is given in Section 5.3 as an example. The revised version of interval arithmetic allows obtaining tighter enclosures at each stage in constraint propagation (see Chapter 6).

3. In Section 5.4, we revise and improve the standard version of affine arithmetic for the purpose of constraint propagation (see Chapter 7). In particular,

    (a) We revise affine forms such that the number of noise variables will not increase during computations; hence, it is potentially useful for long-running computations.

    (b) We point out that Kolev generalized affine arithmetic on revised affine forms is about two times faster than that on Kolev affine forms. Namely, it reduces the number of real operations in the multiplication, the most common but expensive operation, from $4n^2 + 8n + 10$ to $2n^2 + 6n + 10$; where n is the number of noise variables.

    (c) We propose a new formula for the product of two affine forms. The idea allows reducing the complexity $\mathcal{O}(n^2)$ of the tight multiplication in (revised) affine arithmetic and Kolev generalized affine arithmetic to $7n+12$ real operations. This formula also provides tighter enclosures than the most recent version in Kolev affine arithmetic while using the same number of real operations.

    (d) We propose a generic procedure to compute Chebyshev affine approximations in a rigorous manner. When applied to elementary operations/functions, rigorous affine approximations can be obtained for virtually any factorable function.

4. In Section 5.5, we propose an abstract inclusion concept, called the *real inclusion representation.* Although it does not results in new inclusion techniques, it provides a common view into inclusion techniques. This facilitates presenting a novel generic scheme for constraint propagation proposed in Chapter 7. Some common properties of inclusion techniques needed for constraint propagation are also presented formally.

Chapter 6

Numerical Constraint Propagation on Directed Acyclic Graphs

Note: This chapter includes the research conducted jointly with Djamila Sam-Haroud and Hermann Schichl in [Vu *et al.* 2004c]. This is a significant contribution of the thesis.

6.1. Introduction

Numerical constraints are often equalities or inequalities expressed in the factorable form (see Section 2.1.3.3). Many techniques have been proposed to solve numerical CSPs (see Chapter 3). They often exploit the factorability of numerical constraints. To achieve the full mathematical rigor when dealing with floating-point numbers, most solution techniques have been based on interval arithmetic or its variants (see Section 2.2.2 and Section 2.2.3).

During the last twenty years, a lot of work has been done to devise domain reduction techniques based on interval arithmetic (see Chapter 3). In the early nineties, an interesting approach, called *interval constraint propagation*, has been developed in constraint programming (see [Benhamou and Older 1992, 1997; Van Hentenryck 1997], Section 2.3.3, and Section 3.2.2). This approach associates constraint propagation techniques in artificial intelligence with interval analytic methods. Recently, a domain reduction technique, called **HC4**, was proposed by Benhamou *et al.* [1999], which performs forward evaluations and backward propagations on the tree representation of individual numerical constraints in combination with a simplification of arc consistency to obtain the effect of domain reduction (see Section 3.2.2.2). This technique is referred to as the *forward-backward propagation*. More recently, a fundamental framework for interval analysis on directed acyclic graphs (DAGs) has been proposed by Schichl and Neumaier [2004b]. Among the proposals, this framework shows that the above forward evaluation and backward propagation can also be performed on DAGs. Replacing trees by DAGs potentially reduces the number of computations in the forward-backward propagation.

This chapter presents a constraint propagation technique following the DAG-based framework proposed by [Schichl and Neumaier 2004b]. The contribution is twofold. First, we show how the DAG-based framework can be made efficient and practical for performing constraint propagation on DAGs (Section 6.3, Section 6.4, and Section 6.5). Second, we propose a new algorithm to coordinate constraint propagation and exhaustive search on DAGs (Section 6.6). In particular, we propose an algorithm to perform a kind of forward-backward propagation on DAGs, which is able to work on *partial DAG representations*. This algorithm restricts the

work to relevant *subsets of constraints* while keeping the DAG representation of the *initial* problem. Another specificity of our forward-backward propagation technique is that it makes it possible to flexibly choose different *interval forms* at different stages of the propagation. We also propose a solution technique, which coordinates our partial forward-backward propagation on DAGs and exhaustive search, in the branch-and-prune framework. The experiments carried out on various problems show that the new approach outperforms previously available propagation techniques by 1 to 2 orders of magnitude or more in speed, while being roughly the same quality w.r.t. enclosure properties for unbiasedly chosen benchmarks (see Section 6.7).

6.2. DAG Representations for Numerical CSPs

6.2.1. Basic Concepts of Directed Acyclic Graphs

For completeness, we recall hereafter some fundamental concepts in graph theory related to the concept of a DAG representation, which was originated in the foundational paper by Schichl and Neumaier [2004b], of a constraint system.

Definition 6.1 (Directed Multigraph). A *directed multigraph* $\mathcal{G} \equiv (V, E, f)$ consists of a finite set V of *vertices* (also called *nodes*), a finite set E of *edges* (also called *arcs*), and a mapping $f \equiv (f_{\mathrm{s}}, f_{\mathrm{t}})^{\mathrm{T}} : E \to V \times V$ such that for all $e \in E$ we have $f_{\mathrm{s}}(e) \neq f_{\mathrm{t}}(e)$. For every edge $e \in E$, we define the *source* of e as $f_{\mathrm{s}}(e)$ and the *target* of e as $f_{\mathrm{t}}(e)$.

In the above definition, if we replace f with a function that maps each edge to an unordered pair of vertices, we then obtain the definition of a *multigraph*. In addition to that, if we allow the source and target of an edge be the same, then the obtained one is called a *pseudograph*. We can also obtain the concept of a *directed pseudograph* in the same way.

Definition 6.2 (In-Edges, Out-Edges). Using the notations in Definition 6.1, we define the set of all *in-edges* of a vertex $v \in V$ as $\mathsf{in\text{-}edges}(v) \equiv \{e \mid f_{\mathrm{t}}(e) = v\}$. Similarly, we define the set of all *out-edges* of a vertex $v \in V$ as $\mathsf{out\text{-}edges}(v) \equiv \{e \mid f_{\mathrm{s}}(e) = v\}$.

In other words, $\mathsf{in\text{-}edges}(v)$ is the set of all edges having v as their target and $\mathsf{out\text{-}edges}(v)$ is the set of all edges having v as their source. Similarly to a tree, the concepts of a leaf and a root (node) in a directed multigraph are defined as follows.

Definition 6.3 (Leaf, Root). Consider a directed multigraph $\mathcal{G}$. A vertex v of $\mathcal{G}$ is called a *leaf* of $\mathcal{G}$ if $\mathsf{in\text{-}edges}(v) = \emptyset$. A vertex v of $\mathcal{G}$ is called a *root* of $\mathcal{G}$ if $\mathsf{out\text{-}edges}(v) = \emptyset$.

Unlike a (directed) tree, a directed multigraph may have many roots (and many leaves).

Definition 6.4 (Directed Path, Cycle, Directed Acyclic Multigraph). Consider a directed multigraph $\mathcal{G} \equiv (V, E, f)$, where $f \equiv (f_{\mathrm{s}}, f_{\mathrm{t}})^{\mathrm{T}}$. A *directed path* from a vertex $v_1 \in V$ to a vertex $v_2 \in V$ is a sequence $(e_i)_{i=1}^{n}$ of edges, where $n \geq 2$, such that $v_1 = f_{\mathrm{s}}(e_1)$, $v_2 = f_{\mathrm{t}}(e_n)$, and $f_{\mathrm{t}}(e_i) = f_{\mathrm{s}}(e_{i+1})$ for all $i = 1, \ldots, n-1$. This directed path is called a *cycle* if $v_1 = v_2$. $\mathcal{G}$ is called a *directed acyclic multigraph* if it does not contain a cycle.

Definition 6.5 (Parent, Child, Ancestor, Descendant). Consider a directed acyclic multigraph $\mathcal{G} \equiv (V, E, f)$. Let v_1 and v_2 be two vertices in V. We say that v_2 is a *parent* of v_1 and that v_1 is a *child* of v_2 if there exists an edge $e \in E$ such that $f(e) = (v_1, v_2)^{\mathrm{T}}$. The set of all parents (respectively, all children) of a vertex $v \in V$ is denoted by $\mathsf{parents}(v)$ (respectively, by $\mathsf{children}(v)$). We say that v_2 is an *ancestor* of v_1 and that v_1 is a *descendant* of v_2 if there exists a directed path from v_1 to v_2. The set of all ancestors (respectively, all descendants) of a vertex $v \in V$ is denoted by $\mathsf{ancestors}(v)$ (respectively, by $\mathsf{descendants}(v)$).

The following result is fundamental, which illustrates the precedence relationship of nodes in a directed acyclic multigraph.

Theorem 6.6. For every directed acyclic multigraph (V, E, f), there exists a total order $\preceq$ on the vertices V such that for every $v \in V$ and every $u \in \mathsf{ancestors}(v)$, we have $v \preceq u$.

Proof. See the proof of Proposition 4.7 in [Schichl 2003]; or see Procedure **NodeLevel** on page 163, which is is a simple algorithm for assigning a level to each node such that any sort in descending order of the obtained levels will result in a required order. ∎

Definition 6.7 (Directed Multigraph with Ordered Edges). A *directed multigraph with ordered edges* is a quadruple $(V, E, f, \leq)$ such that (V, E, f) is a directed multigraph and $(E, \leq)$ is a totally ordered set.

6.2.2. DAG Representations

As proposed by Schichl and Neumaier [2004b], a directed acyclic multigraph with ordered edges, abbreviated to *DAG*, can be used to represent a factorable numerical CSP of the form

$$f(x) \in \mathbf{b}, \tag{6.1}$$

where x is a vector of n real variables taking values in a box $\mathbf{x} \in \mathbb{I}^n$, $\mathbf{b} \equiv (\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_m)^{\mathrm{T}}$ is an interval vector in $\mathbb{I}^m$, and $f = (f_1, f_2, \ldots, f_m)^{\mathrm{T}}$ is a function from $D \subseteq \mathbb{R}^n$ to $\mathbb{R}^m$. Since the system (6.1) is factorable, the function f can be composed of a sequence of elementary operations/functions such as $+$, $*$, $/$, log, exp, sqr, and sqrt. In this composition, each variable is represented by a leaf. Each elementary operation/function $\phi : D \subseteq \mathbb{R}^k \to \mathbb{R}$ that takes as input k subexpressions $x_1, \ldots, x_k$ is represented by a node $\mathbf{N}$ with k edges, each of which runs from the node representing x_i to the node $\mathbf{N}$, where $1 \leq i \leq k$. These k edges represent the computational flow in the natural composition of the operation ϕ. The obtained representation is called the *DAG representation* of the considered problem.

Notation 6.8. Each node $\mathbf{N}$ in a DAG representation is associated with an interval, denoted by $\tau_{\mathbf{N}}$ and called the *node range* of $\mathbf{N}$, in which the exact range of the associated subexpression must lie. The node $\mathbf{N}$ is also associated with a real variable, denoted by $\vartheta_{\mathbf{N}}$, that represents the value of the subexpression represented by $\mathbf{N}$.

For efficiency and compactness, the standard elementary operations in a DAG representation are replaced with more general operations. For example, multiple applications of binary elementary operations of the forms in $\{x+y,\ x-y,\ x+a,\ a+x,\ x-a,\ a-x,\ ax\}$ are replaced with a k-ary operation $a_0+a_1x_1+\cdots+a_kx_k$, which is interpreted as a k-ary operation $+$ (see Figure 6.1a), where $1 \le k \in \mathbb{N}$. Similarly, multiple applications of the binary multiplication $x*y$ are replaced with a k-ary multiplication (or product) $a_0 * x_1 * \cdots * x_k$, which is interpreted as the k-ary operation $*$ (see Figure 6.1b), where $2 \le k \in \mathbb{N}$. In general, each edge of a DAG representation is associated with a respective coefficient of the operation represented by its target. When not specified in figures, this coefficient equals to 1. The other constants involving an operation are stored at the node representing the operation (see Figure 6.1). At a result, the DAG representation does not contain nodes representing constants as the corresponding tree representation does (see Section 3.2.2.2). Much more detailed descriptions of DAG representations can be found in Section 4.1 and Section 5.3 of [Schichl 2003].

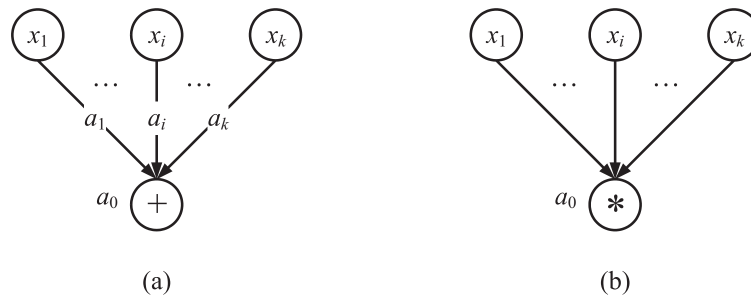

Figure 6.1. A node and its computational flows in a DAG representation.

In practice, we have to use multigraphs instead of simple graphs for DAG representations because some special operations can take the same input more than once. For example, the expression x^x can be represented by the power operation x^y. Hence, we do not need a univariate operation x^x for this purpose. In all cases, a normal directed acyclic graph is sufficient to represent a numerical CSP (NCSP) if we allow introducing new elementary operations such as the univariate function x^x. The ordering of edges is needed for non-commutative operations like the division. For convenience, a virtual ground node, called **G**, is added to each DAG representation to be the parent of all nodes that represents constraints. In fact, the ground node can be interpreted as the logical `AND` operation.

Example 6.9. Consider the following constraint system

$$\begin{cases} \sqrt{x} + 2\sqrt{xy} + 2\sqrt{y} \le 7, \\ 0 \le x^2\sqrt{y} - 2xy + 3\sqrt{y} \le 2, \\ x \in [1,16],\ y \in [1,16], \end{cases}$$

which can be written into the form (6.1) as follows

$$\begin{cases} \sqrt{x} + 2\sqrt{xy} + 2\sqrt{y} \in [-\infty, 7], \\ x^2\sqrt{y} - 2xy + 3\sqrt{y} \in [0,2], \\ x \in [1,16],\ y \in [1,16]. \end{cases} \tag{6.2}$$

The DAG representation of the constraint system (6.2) is depicted in Figure 6.2. Two constraints of the system (6.2) are represented by two nodes $\mathbf{N}_9$ and $\mathbf{N}_{10}$. Two variables, x and

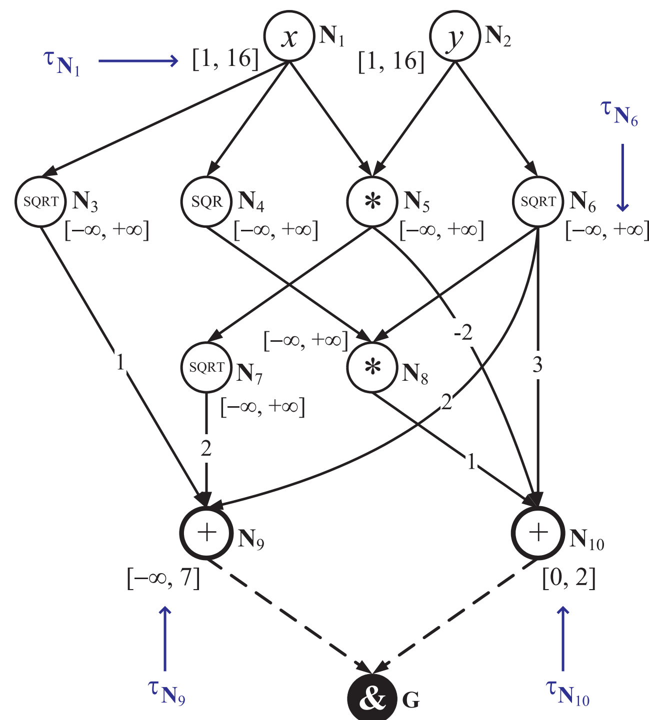

Figure 6.2. The DAG representation of the constraint system (6.2).

y, are represented by two nodes $\mathbf{N}_1$ and $\mathbf{N}_2$, respectively. The sequence $(\mathbf{N}_1, \mathbf{N}_2, \ldots, \mathbf{N}_{10})$ of nodes given in Figure 6.2 is an example of an ordering as stated in Theorem 6.6. ♣

For the same constraint system, the DAG representation is clearly more concise than the tree representation described in Section 3.2.2.2.

6.3. Forward-Backward Propagation on DAG Representations

When proposing the concept of a DAG representation, Schichl and Neumaier [2004b] suggested to adapt the concepts of a *forward evaluation* and a *backward propagation* defined on tree representations of individual constraints of a constraint system (see Section 3.2.2.2) to those defined on DAG representations of whole constraint systems. The *forward evaluation* at a node, $\mathbf{N}$, is to evaluate the node range of $\mathbf{N}$ based on the node ranges of the children of $\mathbf{N}$. The *backward propagation* at a node, $\mathbf{N}$, is concerned with reducing the node range of each child of $\mathbf{N}$ based on the node range of $\mathbf{N}$ and on the node ranges of the other children of $\mathbf{N}$.

The original forward-backward propagation algorithm, called **HC4**, in [Benhamou *et al.* 1999] was designed to use *natural interval forms* in two processes, a recursive forward evaluation and a recursive backward propagation, as described in Section 3.2.2.2. Each of these two processes is performed at all nodes of the tree representation of a constraint at a time rather than at a single node at a time. In this section, we refine the concepts of a forward evaluation and a backward propagation to adaptively use them at individual nodes. This allows using different interval forms at different stages of the propagation process. Moreover, this refinement serves as a basis for a new constraint propagation algorithm, called **FBPD**, we propose in Section 6.5. Note that, in in this section, we use the notations defined in Notation 6.8.

6.3.1. Forward Evaluation on DAG Representations

Consider the DAG representation of a factorable constraint system of the form (6.1). Let $\mathbf{N}$ be a node that is not the ground node and that has k children: $\mathbf{C}_1, \ldots, \mathbf{C}_k$. The operation represented by $\mathbf{N}$ is a function $h : D_h \subseteq \mathbb{R}^k \to \mathbb{R}$. The relation between $\mathbf{N}$ and its children is given by $\vartheta_{\mathbf{N}} = h(\vartheta_{\mathbf{C}_1}, \ldots, \vartheta_{\mathbf{C}_k})$. We define the forward evaluation at a node as follows.

Definition 6.10 (Forward Evaluation Rule). Consider the node $\mathbf{N}$ and its operation, h, as above described. Let $[h]$ be an interval form of the $\{\emptyset\}$-extended function over $\mathbb{R}$ of h. The *forward evaluation using the interval form* $[h]$ *at node* $\mathbf{N}$ is defined and denoted by

$$\texttt{FE}(\mathbf{N}, [h]) \equiv (\tau_{\mathbf{N}} := \tau_{\mathbf{N}} \cap [h](\tau_{\mathbf{C}_1}, \ldots, \tau_{\mathbf{C}_k})). \tag{6.3}$$

Example 6.11. Consider the node $\mathbf{N}_7$ in Figure 6.2, $h(z) \equiv \sqrt{z}$, where $z \equiv \vartheta_{\mathbf{N}_5}$. We can use any interval form $[h]$ of the $\{\emptyset\}$-extended function over $\mathbb{R}$ of h (i.e., the function $\sqrt{z}^{\emptyset}$ defined by (5.10)) for the forward evaluation (6.3). For example, we can use the natural interval form $\mathbf{h}(\mathbf{z}) \equiv \sqrt{\mathbf{z}}$ in place of $[h]$ in (6.3). ♣

Remark 6.12. We can also replace the interval form $[h]$ in (6.3) with an interval form of the recursive subexpression of which the variables are user's ones. For instance, we can replace the interval form $[h]$ of the node $\mathbf{N}_7$ in Figure 6.2 with the natural interval form of the recursive subexpression ($\sqrt{xy}$) composed of the nodes $\mathbf{N}_7$, $\mathbf{N}_4$, $\mathbf{N}_1$, and $\mathbf{N}_2$. That is, we can replace $[h]$ with the bivariate interval function $\sqrt{\mathbf{xy}}$.

In our implementation, we use the natural interval form for simplicity. The natural interval form of the function $h(x_1, \ldots, x_k) = a_0 + a_1 x_1 + \cdots + a_k x_k$ is the function $\mathbf{h}(\mathbf{x}_1, \ldots, \mathbf{x}_k) = a_0 + a_1\mathbf{x}_1 + \cdots + a_k\mathbf{x}_k$.[1] Similarly, the natural interval form of the function $h(x_1, \ldots, x_k) = a x_1 \ldots x_k$ is the function $\mathbf{h}(\mathbf{x}_1, \ldots, \mathbf{x}_k) = a\mathbf{x}_1 \ldots \mathbf{x}_k$. The division of two reals has not only one but many natural interval forms because it is originally not defined when the denominator is zero (see Section 5.2 and Section 5.3). In Definition 5.11, we have pointed out three versions that can be called the natural interval form of the standard real division: $[\div_{\emptyset}]$, $[\div_{\star}]$, and $[\div_{\mathbb{R}}]$. They all can be used in the forward evaluation (6.3) if h is the division.

The following theorem states the correctness of the forward evaluation in Definition 6.10.

Theorem 6.13. Consider the DAG representation of a factorable numerical CSP of the form (6.1). The forward evaluation defined in Definition 6.10, when applied to any node, never discard a solution of the considered problem.

Proof. For every solution of the considered problem, there exists an assignment of values from the intervals $\tau_{\mathbf{N}}, \tau_{\mathbf{C}_1}, \ldots, \tau_{\mathbf{C}_k}$ to the variables $\vartheta_{\mathbf{N}}, \vartheta_{\mathbf{C}_1}, \ldots, \vartheta_{\mathbf{C}_k}$, respectively, such that $\vartheta_{\mathbf{N}} = h(\vartheta_{\mathbf{C}_1}, \ldots, \vartheta_{\mathbf{C}_k})$. Because $[h]$ is an interval form of the $\{\emptyset\}$-extended function over $\mathbb{R}$ of h, it follows from Theorem 5.10 that $h(\vartheta_{\mathbf{C}_1}, \ldots, \vartheta_{\mathbf{C}_k}) \in [h](\tau_{\mathbf{C}_1}, \ldots, \tau_{\mathbf{C}_k})$. Hence, $\vartheta_{\mathbf{N}} \in \tau_{\mathbf{N}} \cap [h](\tau_{\mathbf{C}_1}, \ldots, \tau_{\mathbf{C}_k})$. The proof is thus completed. ■

[1] Note that if the coefficients $a_0, \ldots, a_k$ are real and we working on the floating-point number system, we can replace each a_i in $\mathbf{h}$ with the smallest interval containing it, where $1 \leq i \leq k$.

6.3.2. Backward Propagation on DAG Representations

Consider the DAG representation of a factorable constraint system of the form (6.1). Let $\mathbf{N}$ be a node that is not the ground node and that has k children: $\mathbf{C}_1, \ldots, \mathbf{C}_k$. The operation represented by $\mathbf{N}$ is a function $h : D_h \subseteq \mathbb{R}^k \to \mathbb{R}$. The *backward propagation* attempts to prune each node range $\tau_{\mathbf{C}_i}$ of $\mathbf{C}_i$ based on the node range $\tau_{\mathbf{N}}$ of $\mathbf{N}$ and on the node ranges of the other children, where $1 \leq i \leq k$. In other words, for each child $\mathbf{C}_i$, the backward propagation attempts to encloses the i-th projection of the relation $\vartheta_{\mathbf{N}} = h(\vartheta_{\mathbf{C}_1}, \ldots, \vartheta_{\mathbf{C}_k})$ on the variable $\vartheta_{\mathbf{C}_i}$ in a tight interval. This procedure is called the i-th backward propagation at $\mathbf{N}$ and denoted by $\mathtt{BP}(\mathbf{N}, \mathbf{C}_i)$. We define the following as the backward propagation at $\mathbf{N}$:

$$\mathtt{BP}(\mathbf{N}) = \{\mathtt{BP}(\mathbf{N}, \mathbf{C}_1), \ldots, \mathtt{BP}(\mathbf{N}, \mathbf{C}_k)\}. \tag{6.4}$$

Although the exact projection of a relation is expensive, in general, an enclosure of the exact projection of an elementary operation can often be obtained at low cost. Indeed, suppose that, from the relation $\vartheta_{\mathbf{N}} = h(\vartheta_{\mathbf{C}_1}, \ldots, \vartheta_{\mathbf{C}_k})$, we can infer an equivalent relation

$$\vartheta_{\mathbf{C}_i} = g_i(\vartheta_{\mathbf{N}}, \vartheta_{\mathbf{C}_1}, \ldots, \vartheta_{\mathbf{C}_{i-1}}, \vartheta_{\mathbf{C}_{i+1}}, \ldots, \vartheta_{\mathbf{C}_k})$$

for some $i \in \{1, \ldots, k\}$, where g_i is a function from $D_g \subseteq \mathbb{R}^k$ to $\mathbb{R}$ such that

$$D_g \supseteq \tau_{\mathbf{N}} \times \tau_{\mathbf{C}_1} \times \cdots \times \tau_{\mathbf{C}_{i-1}} \times \tau_{\mathbf{C}_{i+1}} \times \cdots \times \tau_{\mathbf{C}_k}.$$

Let $[g_i]$ be an interval form of the $\{\emptyset\}$-extended function over $\mathbb{R}$ of g_i. The i-th backward propagation, denoted $\mathtt{BP}(\mathbf{N}, \mathbf{C}_i)$, can then be defined as

$$\mathtt{BP}(\mathbf{N}, \mathbf{C}_i) \equiv (\tau_{\mathbf{C}_i} := \tau_{\mathbf{C}_i} \cap [g_i](\tau_{\mathbf{N}}, \tau_{\mathbf{C}_1}, \ldots, \tau_{\mathbf{C}_{i-1}}, \tau_{\mathbf{C}_{i+1}}, \ldots, \tau_{\mathbf{C}_k})). \tag{6.5}$$

In case we cannot infer such a function g_i, more complicated rules have to be constructed in order to obtain the i-th projection of the relation $\mathbf{N} = f(\mathbf{C}_1, \ldots, \mathbf{C}_k)$ if the cost is low, alternatively the relation can be ignored. Fortunately, we can enclose such projections for most elementary operations at low cost, as shown in Definition 6.16.

Remark 6.14. In general, the relation $x = y * z$ and the relation $z = x/y$ are not equivalent because the latter discards the case y = 0 while the former does not.

Example 6.15. Consider the node $\mathbf{N}_{10}$ in Figure 6.2. The relation given at $\mathbf{N}_{10}$ is $\vartheta_{\mathbf{N}_{10}} = h(\vartheta_{\mathbf{N}_5}, \vartheta_{\mathbf{N}_6}, \vartheta_{\mathbf{N}_8})$, where the function h is defined as $h(x_1, x_2, x_3) \equiv -2x_1 + 3x_2 + x_3$. Therefore, we can infer three equivalent relations:

$$\begin{aligned}
\vartheta_{\mathbf{N}_5} &= g_1(\vartheta_{\mathbf{N}_{10}}, \vartheta_{\mathbf{N}_6}, \vartheta_{\mathbf{N}_8}),\\
\vartheta_{\mathbf{N}_6} &= g_2(\vartheta_{\mathbf{N}_{10}}, \vartheta_{\mathbf{N}_5}, \vartheta_{\mathbf{N}_8}),\\
\vartheta_{\mathbf{N}_8} &= g_3(\vartheta_{\mathbf{N}_{10}}, \vartheta_{\mathbf{N}_5}, \vartheta_{\mathbf{N}_6}),
\end{aligned}$$

where three functions g_1, g_2, and g_3 are defined as follows:

$$\begin{aligned}
g_1(x_1, x_2, x_3) &\equiv (-x_1 + 3x_2 + x_3)/2,\\
g_2(x_1, x_2, x_3) &\equiv (x_1 + 2x_2 - x_3)/3,\\
g_3(x_1, x_2, x_3) &\equiv x_1 + 2x_2 - 3x_3.
\end{aligned}$$

♣

Definition 6.16 (Backward Propagation Rule). Let h be the elementary operation represented by $\mathbf{N}$, as discussed above. We use the notation $\oslash$ to mean that either the division $[\div_\star]$ or the division $[\div_\mathbb{R}]$ can be used at the place the notation $\oslash$ appears, but the former is better. The rules for the backward propagation are given as follows:

1. If h is a univariate function such as sqr, sqrt, exp, and log and if $[h]$ is an interval form of the $\{\emptyset\}$-extended function of h, we define

$$\mathtt{BP}(\mathbf{N},\mathbf{C}_1) \equiv \left(\tau_{\mathbf{C}_1} := \tau_{\mathbf{C}_1} \cap [h^{-1}](\tau_\mathbf{N})\right),$$

   where the notation of interval form, $[h^{-1}](\mathbf{x})$, to denote the union of some intervals that contains the inverse image $h^{-1}(\mathbf{x})$;

2. If h is defined as $h(x_1,\ldots,x_k) \equiv a_0 + a_1x_1 + \cdots + a_kx_k$, we define

$$\mathtt{BP}(\mathbf{N},\mathbf{C}_i) \equiv \left(\tau_{\mathbf{C}_i} := \tau_{\mathbf{C}_i} \cap \left((\tau_\mathbf{N} - a_0 \ - \sum_{j=1;\ j\neq i}^{k} a_j * \tau_{\mathbf{C}_j}) \oslash a_i\right)\right) \quad (\text{for } i = 1,\ldots,k);$$

3. If h is defined as $h(x_1,\ldots,x_k) \equiv ax_1\ldots x_k$, we define

$$\mathtt{BP}(\mathbf{N},\mathbf{C}_i) \equiv \left(\tau_{\mathbf{C}_i} := \tau_{\mathbf{C}_i} \cap \left(\tau_\mathbf{N} \oslash (a \ * \prod_{j=1;\ j\neq i}^{k} \tau_{\mathbf{C}_j})\right)\right) \quad (\text{for } i = 1,\ldots,k);$$

4. If h is defined as $h(x,y) \equiv x/y$ (i.e., $k = 2$), we define

$$\mathtt{BP}(\mathbf{N},\mathbf{C}_1) \equiv (\tau_{\mathbf{C}_1} := \tau_{\mathbf{C}_1} \cap (\tau_\mathbf{N} * \tau_{\mathbf{C}_2})),$$
$$\mathtt{BP}(\mathbf{N},\mathbf{C}_2) \equiv (\tau_{\mathbf{C}_2} := \tau_{\mathbf{C}_2} \cap (\tau_{\mathbf{C}_1} \oslash \tau_\mathbf{N})).$$

The following theorem states the correctness of the backward propagation rules given in Definition 6.16.

Theorem 6.17. Consider the DAG representation of a factorable numerical CSP of the form (6.1). The backward propagation defined in Definition 6.16, when applied to any node, never discard any solution of the considered problem.

Proof. By an argument similar to the proof of Theorem 6.13, we have that the first result is due to the definition of h^{-1} in Definition 5.2, and the other results are due to Theorem 5.13 and the inclusion property of the operations $+$, $-$, and $*$ in (standard) interval arithmetic. ■

6.4. Partial DAG Representations for Numerical CSPs

The complete solution to numerical CSPs (NCSPs) is essentially based on the *branch-and-prune* framework (see Section 3.2.2), where the solving process is performed by repeatedly interleaving

a *pruning* step with a *branching* step. The former often uses domain reduction techniques, such as constraint propagation, to reduce variable domains, while the latter splits a problem into subproblems. Each subproblem, which needs to be solved, often consists of (i) a subset, called the set of *running constraints*, of the set of initial constraints; and (ii) subdomains of the initial variable domains. If DAG representation is used in the pruning steps, each instance of it needs to be constructed for a considered subproblem. Therefore, the simplest way consists of explicitly constructing a new DAG to represent each subproblem. However, the total cost of creating such DAGs is potentially high, since there are often a huge number of branching steps during a complete solving process.

Alternatively, we propose to attach a piece of restriction information to the initial DAG, which is the DAG representation of the initial problem, in order to make the initial DAG interpreted as the DAG representation of a subproblem without the necessity of creating a new DAG. Using the piece of information, it is possible to perform forward evaluation and backward propagation on the DAG representation of the initial problem without increasing much the time and space needed for the propagation. A combination of a piece of restriction information and the DAG representation of the initial problem is called a (respectively, the) *partial DAG representation* of the initial problem (respectively, the considered subproblem). For example, partial DAG representations of the problem (6.2) are depicted in Figure 6.3. We use partial DAG representations in our new constraint propagation algorithm (see Section 6.5).

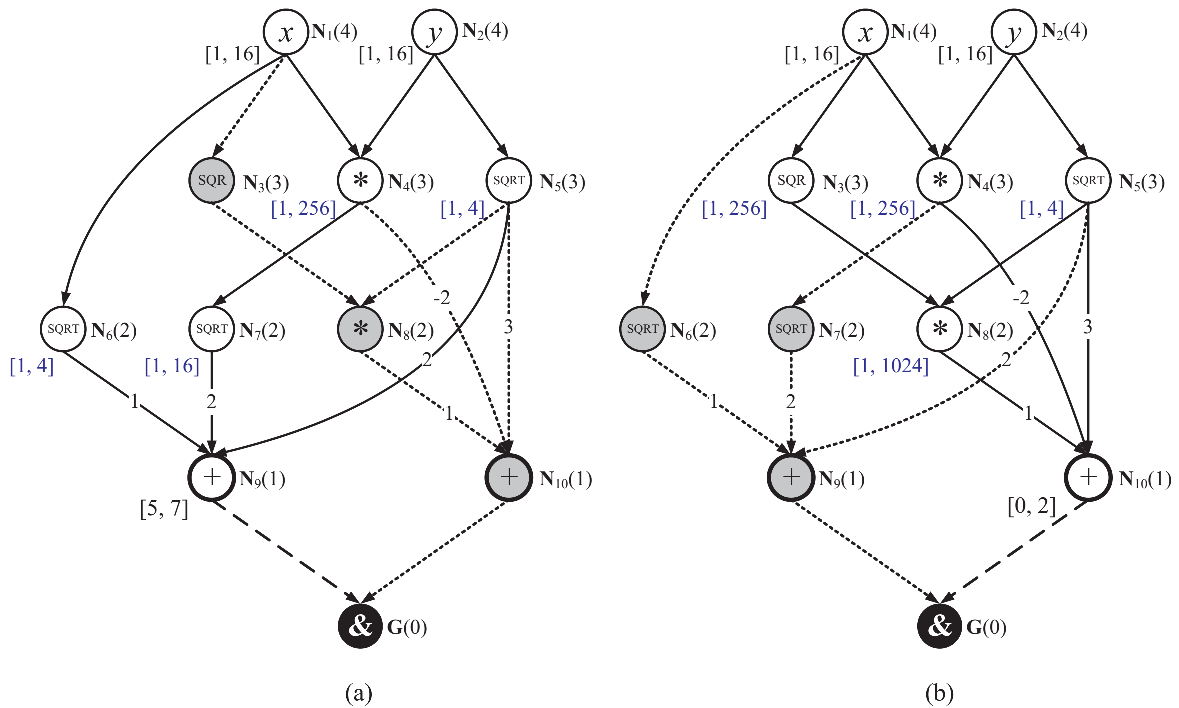

Figure 6.3. Partial DAG representations of the problem (6.2) in the cases: (a) the first constraint is the unique running constraint; and (b) the second constraint is the unique running constraint. The grey nodes are not counted, hence ignored in computations. The dotted edges are redundant. The node levels (in the brackets) are not updated.

In order to represent a subproblem with a set of running constraints without having to create a new DAG, we use a vector V_{oc} of which the size equals to the number of nodes in the

DAG representation, $D(\mathbf{G})$, of the initial problem. For each node $\mathbf{N}$ of $D(\mathbf{G})$, we use the entry $V_{\text{oc}}[\mathbf{N}]$ to count the number of occurrences of $\mathbf{N}$ in the recursive composition of the running constraints. We present a simple recursive procedure, called **NodeOccurrences**, on page 160 to compute such a vector. If we invoke Procedure **NodeOccurrences** at all the nodes representing the running constraints, each entry $V_{\text{oc}}[\mathbf{N}]$ will contains the number of occurrences of $\mathbf{N}$ in the recursive composition of the running constraints. In particular, we have $V_{\text{oc}}[\mathbf{N}] = 0$ holds if and only if $\mathbf{N}$ is not in the representation of the running constraints. Therefore, by combining $D(\mathbf{G})$ with a vector V_{oc}, we have the so-called *partial DAG representation* for a subproblem. In computations, we can use partial DAG representations in a way similar to the way we use DAG representations, except that we ignore every node $\mathbf{N}$ with $V_{\text{oc}}[\mathbf{N}] = 0$. Figure 6.3 shows partial DAG representations of the problem (6.2).

Procedure **NodeOccurrences**(**in:** a node $\mathbf{N}$; **in/out:** a vector V_{oc})

```
foreach C ∈ children(N) do
    V_oc[C] := V_oc[C] + 1;
    NodeOccurrences(C, V_oc);
end
```

6.5. Constraint Propagation on Partial DAG Representations

We recall that each step of the original forward-backward propagation algorithm, called **HC4** and proposed by Benhamou *et al.* [1999], consists of two processes: a recursive forward evaluation and a recursive backward propagation (see Section 3.2.2.2). Each of these processes is recursively performed on the whole tree at a time rather than at individual nodes. In this section, we show how this propagation can be enhanced by a flexible choice of individual nodes for performing the forward evaluation and the backward propagation. The nature of the newly proposed method enables the use of different interval forms at different propagation stages. For efficiency, the new forward-backward propagation is performed on partial DAG representations of the initial problem. In other words, it is a partial forward-backward propagation on the initial DAG. We then devise, in Section 6.6, a detailed search algorithm in combination with the forward-backward propagation on partial DAG representations.

Inspired by the recursive forward evaluation and recursive backward propagation in the **HC4** algorithm, we devise a new algorithm for numerical constraint propagation, which is based on partial DAG representations instead of tree representations. The new algorithm is called the *forward-backward propagation on DAGs*, and is abbreviated to **FBPD**. The main steps of **FBPD** is presented in Algorithm 6.2. Like the **HC4** algorithm, the main body of the **FBPD** algorithm has two principal processes: forward evaluation and backward propagation. Unlike the **HC4** algorithm, the **FBPD** algorithm, however, performs these processes for a single node instead of all nodes at once. Therefore, in the **FBPD** algorithm, the choice of the next node for further processing can be adaptively made based on the results of the previous processes. Moreover, in the **FBPD** algorithm, the choice of the interval form $[h]$ of an operation h for the forward evaluation and the backward propagation is not necessarily fixed. The interval form $[h]$ can be chosen statically or dynamically based on the nature of h at the current context. This issue will be addressed elsewhere as a continued work. In the next paragraphs, we describe in detail the procedures that are not made explicit in Algorithm 6.2.

Algorithm 6.2: The **FBPD** algorithm – a constraint propagation on DAGs

```
    Input: a DAG D(G) with the ground G, variable domains 𝒟, running constraints 𝒞.
    Output: new domains 𝒟.
    Reset all node ranges of D(G) to [−∞, ∞];
    Set the node ranges of variables & constraints to 𝒟 & the constraint ranges of 𝒞, resp.;
    ℒf := ∅; ℒb := ∅; Voc := (0, ..., 0); Vch := (0, ..., 0);
1   Vlvl := (0, ..., 0);                        ◂ This can be made optional together with Line 2.
    foreach node C representing a running constraint in 𝒞 do
        NodeOccurrences(C, Voc);                                         ◂ On page 160.
2       NodeLevel(C, Vlvl);            ◂ This can be made optional together with Line 1. On page 163.
3       ReForwardEvaluation(C, Vch, ℒb);     ◂ A full recursive forward evaluation. On page 162.
        if the infeasible status was detected then return 𝒟 := ∅;
    end
    while ℒb ≠ ∅ ∨ ℒf ≠ ∅ do
4       N := getNextNode(ℒb, ℒf);
        if N was taken from ℒb then
            foreach child C of N do
5               BP(N, C);                                          ◂ See Definition 6.16.
                if τC = ∅ then return 𝒟 := ∅;          ◂ The infeasible status was detected.
6               if τC was changed enough for forward evaluation then
                    foreach P ∈ parents(C) \ {N, G} do
                        if Voc[P] > 0 then Put P into ℒf;   ◂ P occurs in a running constraint.
                    end
                end
7               if τC was changed enough for backward propagation then
                    Put C into ℒb;
                end
            end
        else                                                      ◂ N was taken from ℒf.
8           FE(N, [h]);     ◂ h is the operator at N, [h] is an interval form of h, see Definition 6.10.
            if τN = ∅ then return 𝒟 := ∅;              ◂ The infeasible status was detected.
9           if τN was changed enough for forward evaluation then
                foreach P ∈ parents(N) \ {G} do
                    if Voc[P] > 0 then Put P into ℒf;   ◂ P occurs in some running constraint.
                end
            end
10          if τN was changed enough for backward propagation then
                Put N into ℒb;
            end
        end
    end
    Update 𝒟 with the node ranges of the variables;
```

Recursive Forward Evaluation. Similarly to the **HC4** algorithm, we perform a recursive forward evaluation at the initialization phase (Line 3 in Algorithm 6.2) to evaluate the node ranges of all nodes in a partial DAG representation, that is, the nodes with nonzero entries in V_{oc}. Procedure **ReForwardEvaluation** (on page 162) provides such an algorithm. In order

to avoid evaluating the same subexpressions many times, we use a vector, V_{ch}, to mark the caching status of node ranges. A node $\mathbf{N}$ is marked as "cached", by setting $V_{\text{ch}}[\mathbf{N}] := 1$, if its node range has already been computed. The result of the recursive forward evaluation of the constraint system (6.2) is depicted in Figure 6.4 (for the case both constraints are running) and Figure 6.3 (for the case only one constraint is running).

Procedure ReForwardEvaluation(**in**: a node $\mathbf{N}$; **in/out**: a vector V_{ch}, a list $\mathcal{L}_{\text{b}}$)

if $\mathbf{N}$ is a leaf **or** $V_{\text{ch}}[\mathbf{N}] = 1$ **then** **return** ; ◀ $\mathbf{N}$ is a leaf or has been cached.
foreach $\mathbf{C} \in$ children($\mathbf{N}$) **do**
| **ReForwardEvaluation**($\mathbf{C}$, V_{ch}, $\mathcal{L}_{\text{b}}$);
end
if $\mathbf{N} = \mathbf{G}$ **then** **return** ;
FE($\mathbf{N}$, $[h]$); ◀ This is similar to Line 8 in Algorithm 6.2.
$V_{\text{ch}}[\mathbf{N}] := 1$; ◀ The node range of $\mathbf{N}$ is cached.
if $\tau_{\mathbf{N}} = \emptyset$ **then** **return** `infeasible`;
11 **if** $\tau_{\mathbf{N}}$ was changed enough for backward propagation **then**
| Put $\mathbf{C}$ into $\mathcal{L}_{\text{b}}$;
end

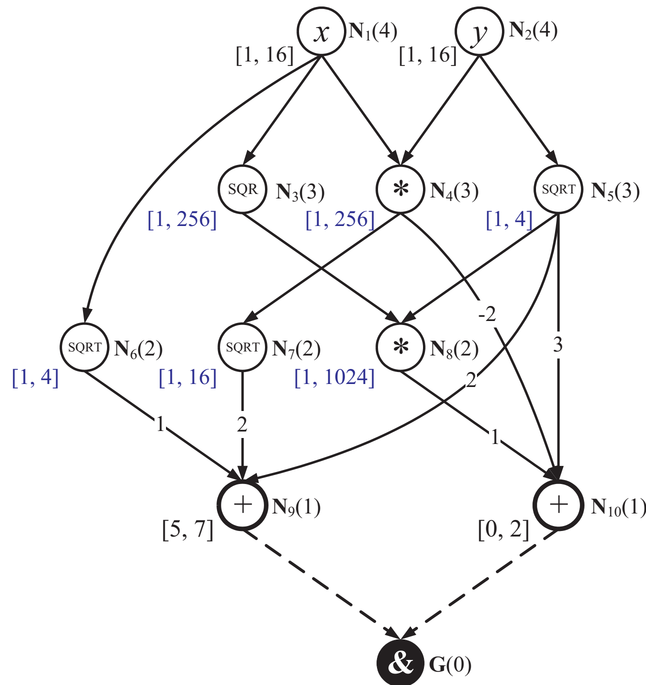

Figure 6.4. The DAG representation of the system (6.2) after a recursive forward evaluation.

Get the Next Node for Further Processing. The **FBPD** algorithm uses two waiting lists, $\mathcal{L}_{\text{f}}$ and $\mathcal{L}_{\text{b}}$, to store the nodes waiting for further processing. The first list, $\mathcal{L}_{\text{f}}$, is a list of nodes that is scheduled for forward evaluation, that is, for evaluating its node range based on the node ranges of its children. The second list, $\mathcal{L}_{\text{b}}$, is a list of nodes that is waiting for backward propagation, that is, for pruning the node ranges of its children based on its

Procedure NodeLevel(in: a node **N**; **in/out:** a vector V_{lvl})

```
foreach child C of node N do
    V_lvl[C] := max{V_lvl[C], V_lvl[N] + 1};
    NodeLevel(C, V_lvl);
end
```

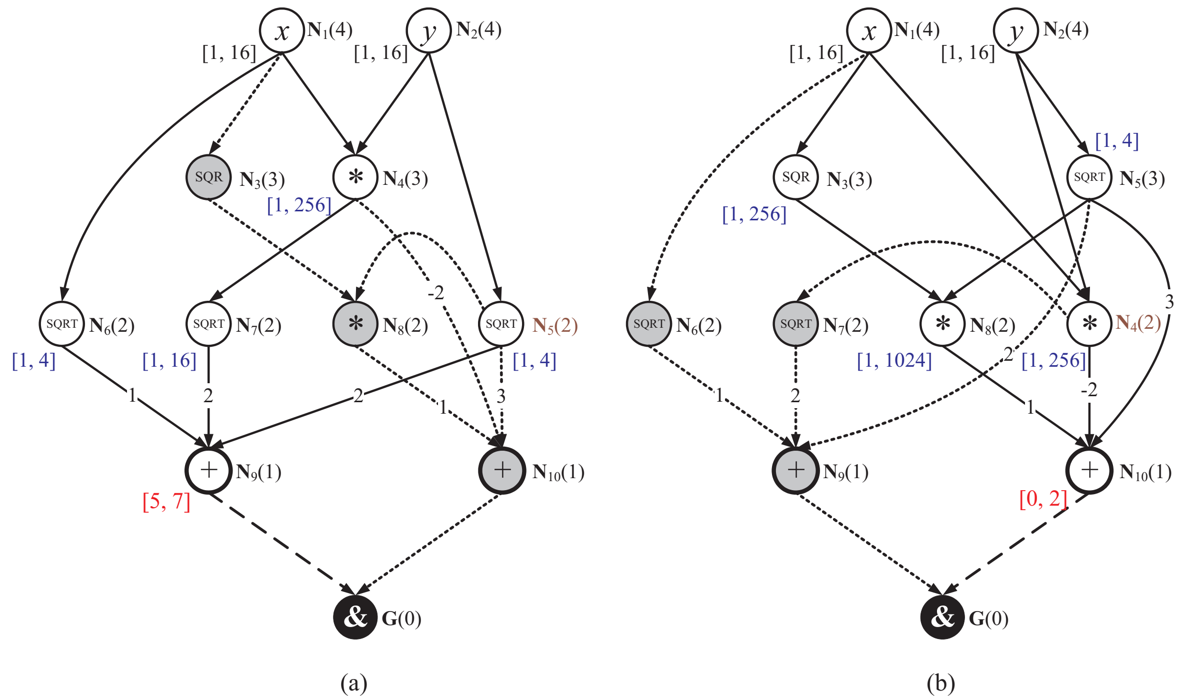

Figure 6.5. The node levels are updated at each call to the **FBPD** algorithm

node range. In general, when $\mathcal{L}_{\text{f}}$ contains many nodes, the nodes should be sorted such that the forward evaluation at a node is performed after the forward evaluations at its children. Analogously, the nodes in $\mathcal{L}_{\text{b}}$ should be sorted such that the backward propagation at a node is performed before the backward propagations at its children. Procedure **NodeLevel** assigns to each node a *node level* such that the node level of an arbitrary node is smaller than the node levels of its descendants (see Theorem 6.6). We then sort the nodes of $\mathcal{L}_{\text{b}}$ and $\mathcal{L}_{\text{f}}$ in ascending order and descending order of node levels, respectively, to meet the above requirements.

The call to Procedure **NodeLevel** at Line 2 in Algorithm 6.2 can be made optional as follows. The first option allows invoking Procedure **NodeLevel** only at the first call to the **FBPD** algorithm. The node levels of the initial DAG still meet the requirements on the ordering of the waiting lists. The numbers in brackets next to the node names in Figure 6.3 are the node levels computed for the initial DAG. Figure 6.5 illustrates the second option that allows invoking Procedure **NodeLevel** at Line 2 in Algorithm 6.2 every time the **FBPD** algorithm is invoked.

The function getNextNode at Line 4 in Algorithm 6.2 chooses one of the two nodes at the beginning of $\mathcal{L}_{\text{b}}$ and $\mathcal{L}_{\text{f}}$. The choosing strategy we use in our implementation is *backward propagation first*, that is, taking the node at the beginning of $\mathcal{L}_{\text{b}}$ whenever $\mathcal{L}_{\text{b}}$ is not empty. Of course, other choosing strategies can also be used.

When Are the Changes of Node Ranges Enough? For simplicity, in Algorithm 6.2 (cf. the lines 6, 7, 9, 10) we only briefly present the procedures to check whether the node ranges have been changed enough for further processing. Hereafter, we detail them. Let $\mathbf{M}$ denote the node $\mathbf{C}$ at Line 5 or the node $\mathbf{N}$ at Line 5 (in Algorithm 6.2). In Algorithm 6.2, the backward propagation at Line 5 and forward evaluation at Line 8 are of the form

$$\tau_{\mathbf{M}} := \tau_{\mathbf{M}} \cap \mathbf{y}, \tag{6.6}$$

where $\mathbf{y}$ is the interval computed by the forward evaluation or backward propagation before intersecting with $\tau_{\mathbf{M}}$.

Let W_{old} and W_{new} be the widths of $\tau_{\mathbf{M}}$ and $\tau_{\mathbf{M}} \cap \mathbf{y}$, respectively. In practice, the change of $\tau_{\mathbf{M}}$ after performing (6.6) is considered enough for doing the forward evaluation at its parents if both the conditions $W_{\text{new}} < r_{\text{f}} * W_{\text{old}}$ and $W_{\text{new}} + d_{\text{f}} \leq W_{\text{old}}$ hold, where r_{f} is a real parameter in $]0, 1[$ and d_{f} is a small positive real parameter. The numbers r_{f} and d_{f} can be predetermined or dynamically computed. Similarly, the change of $\tau_{\mathbf{M}}$ after performing (6.6) is considered enough for doing the backward propagation at $\mathbf{M}$ if both the conditions $W_{\text{new}} < r_{\text{b}} * W_{\text{old}}$ and $W_{\text{new}} + d_{\text{b}} \leq W_{\text{old}}$ hold, where r_{b} is a real parameter in $]0, 1[$ and d_{b} is a small positive real parameter. Moreover, if $\mathbf{y}$ is computed by the forward evaluation (at Line 8), an additional condition that $\mathbf{y} \not\subseteq \tau_{\mathbf{M}}$ must also hold.

The **FBPD** algorithm is *contractive* and *correct* in the following sense.

Theorem 6.18. Let define a function $\Phi : \mathbb{I}^n \to \mathbb{I}^n$ to represent the **FBPD** algorithm. This function takes as input the variable domains in the form of a box $\mathbf{x} \in \mathbb{I}^n$ and returns a box in $\mathbb{I}^n$, denoted by $\Phi(\mathbf{x})$, that represents the variable domains of the output of the **FBPD** algorithm. If the input problem contains only the operations h defined in Definition 6.10 and Definition 6.16, then the **FBPD** algorithm terminates at a finite number of iterations and the following properties hold:

$$\text{(Contractiveness)} \qquad \Phi(\mathbf{x}) \subseteq \mathbf{x}, \tag{6.7}$$

$$\text{(Correctness)} \qquad \Phi(\mathbf{x}) \supseteq \mathbf{x} \cap S, \tag{6.8}$$

where S is the exact solution set of the input problem.

Proof. All the node ranges in the DAG representation of the considered problem are never inflated at each step of the **FBPD** algorithm; hence, the **FBPD** algorithm must terminate at a finite number of iterations because of the finite nature of floating-point numbers. In particular, the ranges of the nodes representing the variables are never inflated; hence, the property (6.7) holds. Moreover, the forward evaluations and backward propagations used in the **FBPD** algorithm are defined in Definition 6.10 and Definition 6.16. It follows from Theorem 6.13 and Theorem 6.17 that they never discard a solution. Therefore, the property (6.8) also holds. ■

6.6. Coordinating Constraint Propagation and Search

We now tackle the issue of coordinating constraint propagation and search for solving NCSPs under the *branch-and-prune* framework, the most common framework for exhaustively solving NCSPs. The most widely used algorithm for search is bisection-based, hence called the *bisection search*. It is suitable for problems with isolated solutions. However, it is often inefficient for

Algorithm 6.5: The **BnPSearch** algorithm – a generic branch-and-prune search

```
Input: a CSP P ≡ (V, D, C).
Output: L_∀, L_ε.                          ◀ Lists of inner and undiscernible boxes, respectively.
Construct the DAG representation D(G) of the initial problem P;
FPBD(D(G), C, D);                          ◀ Prune the domains in D by using Algorithm 6.2.
if D = ∅ then return infeasible;
if the domains in D are small enough then
  | L_ε := L_ε ∪ {(D, C)};
  | return ;
end
WaitingList := {(D, C)};
while WaitingList ≠ ∅ do
  | Get a couple (D_0, C_0) from WaitingList;     ▼/* D_{i=1,k} ⊆ D_0, C_{i=1,k} ⊆ C_0. */
  | Split the CSP (V, D_0, C_0) into sub-CSPs {(V, D_1, C_1), ..., (V, D_k, C_k)};
  | for i := 1, ..., k do
  |   | if C_i = ∅ then
  |   |   | L_∀ := L_∀ ∪ {D_i};                   ◀ All points in D_i are solutions.
  |   |   | continue for;
  |   | end
  |   | FPBD(D(G), C_i, D_i);                     ◀ Prune the domains in D_i by using Algorithm 6.2.
  |   | if D_i = ∅ then continue for;
  |   | if the domains in D_i are small enough then
  |   |   | L_ε := L_ε ∪ {(D_i, C_i)};            ◀ This CSP is not amenable for further splitting.
  |   | end
  |   | WaitingList := WaitingList ∪ {(D_i, C_i)};
  | end
end
```

problems with continuums of solutions, for instance, problems with inequalities. For problems with continuums of solutions, therefore, we need more advanced search techniques like **UCA5**, **UCA6** and **UCA6**$^+$ (see Chapter 4). They all can be viewed as instances of a more generic branch-and-prune search algorithm, called **BnPSearch**, described in Algorithm 6.5.

In general, the **BnPSearch** algorithm produces two lists: $\mathcal{L}_\forall$, $\mathcal{L}_\varepsilon$. The first list, $\mathcal{L}_\forall$, consists of completely feasible subdomains. The second list, $\mathcal{L}_\varepsilon$, consists of tuples, each of which consists of (*i*) a box, which is smaller than the required resolution/precision ε or is canonical, playing the role of domains; and (*ii*) a set of running constraints in the corresponding box.

Because of the finite nature of floating-point numbers, it is easy to prove that the branch-and-prune search presented in Algorithm 6.5 can obtain the predefined positive precision ε (i.e., $\mathcal{L}$ becomes empty) after a finite number of steps. Moreover, this branch-and-prune search is a complete solution technique, w.r.t. the feasibility checker used for classifying ε-bounded boxes (see Chapter 4), because the **FBPD** algorithm is correct and contractive.

6.7. Experiments

We have carried out experiments on the **FBPD** algorithm and two other well-known state-of-the-art interval constraint propagation techniques. The first one is an efficient implementation of box consistency (see Section 2.3.3.3) in a well-known commercial product named ILOG Solver

(v6.0, 11/2003), hereafter denoted by **BOX**. The second one is the **HC4** algorithm [Benhamou *et al.* 1999] (see Section 3.2.2.2). The experiments are carried out on 33 problems, which are *unbiasedly* chosen and divided into five test cases to analyze the empirical results:

- The test case T_1 consists of eight easy problems with isolated solutions. These problems are solvable in short time by the search using the three propagators. (See Section C.2.)
- The test case T_2 consists of four average problems with isolated solutions. These problems are solvable by the search using **FBPD** and **BOX** and cause the search using **HC4** being out of time without reaching 10^6 splits. (See Section C.3.)
- The test case T_3 consists of eight hard problems with isolated solutions. These problems cause the search using **FBPD** being stopped due to running more than 10^6 splits, cause the search using **HC4** being out of time without reaching 10^6 splits, and cause the search using **BOX** either being out of time or being stopped due to running more than 10^6 splits. (See Section C.4.)
- The test case T_4 consists of seven easy problems with a continuum of solutions. These problems are solvable in short time at the predefined precision 10^{-2}. (See Section C.5.)
- The test case T_5 consists of six hard problems with a continuum of solutions. These problems are solvable in short time at the predefined precision 10^{-1}. (See Section C.6.)

The timeout value is set to **10 hours** for all the test cases. ***The timeout values will be used as the running time for the techniques that are out of time in the next result analysis*** (i.e., we are in favor of slow techniques). For the first three test cases, the precision is 10^{-4} and the search to be used is the bisection search. For the last two test cases, the search to be used is a search technique, called **UCA6**, for inequality constraints (see Chapter 4). The comparison of the interval propagation techniques is based on the following measures:

- *The running time:* The relative ratio of the running time of each propagator to that of **FBPD** is called the *relative time ratio.*
- *The number of boxes:* The relative ratio of the number of boxes in the output of each propagator to that of **FBPD** is called the *relative cluster ratio.*
- *The number of splits/iterations:* The number of splits in search needed to solve the problems. The relative ratio of the number of splits used by each propagator to that of **FBPD** is called the *relative iteration ratio.*
- *The volume of boxes (only for T_1, T_2, T_3):* We consider the reduction per dimension $\sqrt[d]{V/D}$; where d is the dimension of the problem, V is the total volume of the output boxes, D is the volume of the initial domains. The relative ratio of the reduction gained by each propagator to that of **FBPD** is called the *relative reduction ratio.*
- *The volume of inner boxes (only for T_4, T_5):* The ratio of the volume of inner boxes to the volume of all output boxes is called the *inner volume ratio.*

The lower the relative ratio is, the better the performance/quality is; and the higher the inner volume ratio is, the better the quality is.

The overviews of results in our experiments are given in Table 6.1 and Table 6.2. In Table 6.3, we give the *overrun ratio* of each propagator for the test case T_1. The *overrun ratio* is defined as $\varepsilon/\sqrt[d]{V/N}$; where ε is the required precision, d is the dimension of the problem, V is the total volume of the output boxes, N is the number of output boxes.

Table 6.1. A comparison of three constraint propagation techniques **FBPD**, **BOX**, and **HC4** in solving NCSPs. In the section (a), the averages of the relative time ratios are taken over all the problems in the test cases T_1, T_2, T_3; and the averages of the other relative ratios are taken over the problems in the test case T_1 (i.e., taken over the problems that are solvable by all the techniques). In the section (b), the averages of the relative ratios are taken over all the problems in the test cases T_4, T_5.

Propagator ▼	(a) Isolated Solutions				(b) Continuum of Solutions			
	Relative time ratio	Relative reduction ratio	Relative cluster ratio	Relative iteration ratio	Relative time ratio	Inner volume ratio	Relative cluster ratio	Relative iteration ratio
FBPD	**1.000**	1.000	1.000	1.000	**1.000**	0.922	1.000	1.000
BOX	20.863	**0.625**	**0.342**	**0.731**	20.919	**0.944**	**0.873**	**0.854**
HC4	203.285	0.906	1.266	0.988	403.915	0.941	0.896	0.879

Table 6.2. The averages of the relative time ratios are taken over the problems in each test case.

Propagator	(a) Isolated Solutions			(b) Continuum of Solutions	
▼	Test case T_1	Test case T_2	Test case T_3	Test case T_4	Test case T_5
FBPD	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
BOX	24.21	28.98	13.45	11.55	31.85
HC4	94.42	691.24	68.17	191.86	651.31

Table 6.3. The overrun ratios for the test case T_1. (An overrun ratio greater than 1 would satisfy the requirements of applications.)

Problem ▶	BIF3	REI3	WIN3	ECO5	ECO6	NEU6	ECO7	ECO8	**Average**
FBPD	1.626	1.360	2.075	1.711	1.676	3.198	1.513	1.455	**1.827**
BOX	2.957	1.974	3.080	1.579	1.660	6.748	1.521	1.485	**2.625**
HC4	2.229	1.914	1.492	1.647	1.679	4.949	1.488	1.449	**2.106**

Clearly, **FBPD** outperforms both **BOX** and **HC4** by 1 to 2 orders of magnitude or more in speed, at least for the unbiasedly chosen benchmarks, while being roughly the same quality w.r.t. enclosure properties in case where the solution set to be enclosed by boxes of macroscopic size (i.e., for continuums of solutions). For isolated solutions, very narrow boxes are produced by any technique in comparison to the required precision. However, the new technique is about 1.1–2.0 times less tight than the other techniques in the measure on reduction per dimension (which hardly matters in applications). In comparison with **HC4** (we recall that **HC4** is a constraint propagation technique that is similar to **FBPD** but works on the tree representation instead of DAGs), **FBPD** is clearly more suitable for applications.

6.8. Conclusion

We propose a new constraint propagation technique, called **FBPD**, which makes the fundamental framework of interval analysis on DAGs [Schichl and Neumaier 2004b] efficient and practical for numerical constraint propagation. We also propose a method to coordinate constraint propagation (**FBPD**) and exhaustive search on partial DAG representations, where only one DAG for each problem is needed for the whole solving process. The experiments carried out on various problems show that the new approach outperforms previously available propagation techniques by 1 to 2 orders of magnitude or more in speed for a set of unbiasedly chosen benchmarks, while being roughly the same quality w.r.t. enclosure properties (they all still satisfy the requirements of applications). The experiments also show that the advance gained for under-constrained problems is better than that for well-constrained problems. Moreover, the design nature of the **FBPD** algorithm is similar to that of the **HC4** algorithm [Benhamou *et al.* 1999]. Therefore, we can use the **FBPD** algorithm in many applications and combination techniques that use the **HC4** algorithm.

In other views, the **FBPD** algorithm can be viewed as a special instance of our generic combination scheme for combining multiple inclusion techniques in numerical constraint propagation, called **CIRD** (see Chapter 7). Our experiments show that the strengths of the **FBPD** algorithm and the **CIRD[ai]** algorithm (an instance of the **CIRD** scheme that uses *affine arithmetic* and *interval arithmetic*) in Chapter 7 are complementary when considering problems with isolated or problems with non-isolated solutions. Therefore, combining and unifying the strengths of **FBPD** and **CIRD[ai]** to solve problems with either isolated or non-isolated solutions is a straightforward direction for further research.

Chapter 7

Combination of Inclusion Techniques in Constraint Propagation

Note: This chapter includes the research conducted jointly with Djamila Sam-Haroud and Boi Faltings in [Vu *et al.* 2004b]. This is the ultimate contribution of the thesis.

7.1. Introduction

Most complete solution techniques follow the *branch-and-prune* framework, which repeatedly interleaves pruning steps with branching steps (see Section 3.2.2). The pruning steps often perform a kind of domain reduction. In early nineties of the twentieth century, an approach, called *interval constraint propagation* [Benhamou and Older 1992, 1997], in constraint programming was proposed for domain reduction. Most domain reduction techniques following this approach are based on, extended from, or similar to several fundamental notations such as kB-consistency [Lhomme 1993], box consistency [Benhamou *et al.* 1994], and hull consistency [Benhamou and Older 1992, 1997]. In particular, these methods aim at computing *box enclosures* of the solution set of a numerical CSP (see Section 2.3.3 and Section 3.2.2).

Recently, *linear relaxations* (or *linear enclosures*) have also been used in constraint programming to *enclose* solutions of numerical CSPs. For example, a domain reduction technique in [Lebbah *et al.* 2003a,b] use *linear relaxations* in combination with a constraint propagation technique. This technique resorts to *linear programming* techniques in order to prune variable domains. Whereas, mathematical solution techniques in [Kolev 2001, 2002] use *generalized interval* and *affine arithmetic* to generate linear enclosures/relaxations, and then use an iterative fixed point reduction rule to narrow variable domains directly instead of resorting to linear programming techniques. The reader can find concise descriptions of several recent linear relaxation based methods, including the above methods, in Section 3.3.

On one hand, different domain reduction techniques have different strengths that are often complementary, in general. Consequently, combining the strengths of different domain reduction techniques is the subject of many intensive research efforts aiming at improving the pruning steps (see Section 3.2.3). On the other hand, the performance of domain reduction techniques significantly depends on strengths of underlying *inclusion techniques*[1]. Therefore, the combination of different inclusion techniques is also considered as an alternative to the combination of domain reduction techniques.

[1] An *inclusion technique* is to include a set of interest into enclosures. It is also called an *enclosure technique*.

In this chapter, we propose a novel generic scheme and several specific strategies to combine multiple inclusion techniques in numerical constraint propagation. Our contributions is described in Section 7.2 and Section 7.3. Namely, we present, in Section 7.2, a novel generic scheme that allows devising new combination strategies for numerical constraint propagation in a flexible way. This scheme enables performing the propagation using different inclusion techniques on (partial) DAG representations of numerical CSPs (see Chapter 6). Moreover, this scheme is applicable to virtually any factorable constraint system. Its goal is to provide a combination scheme that is efficient and flexible but still general enough to bring different inclusion techniques coming from different areas (e.g., constraint programming and mathematical programming) into the framework of constraint propagation. In order to illustrate the flexibility and efficiency of the generic scheme, we base on this scheme and devise, in Section 7.3, several new combination strategies. Theses strategies are based on emerging techniques such as interval constraint propagation, interval arithmetic, affine arithmetic, and linear programming. In Section 7.4, our experiments show that the devised technique is superior than the recent interval propagation methods in performance and quality. It even outperforms some very recent mathematical and constraint programming techniques, which are specially designed to solve special constraint systems. The potential directions for future are presented in Section 7.5. The conclusion is finally given in Section 7.6.

7.2. A Combination Scheme for Constraint Propagation

In Section 5.5, we propose an abstraction of inclusion techniques by using the concept of an *inclusion representation*. Using this abstract inclusion concept, we propose in this section a novel generic scheme to perform constraint propagation on (partial) DAG representations. This scheme allows inclusion techniques to work in cooperation in order to obtain the effect of domain reduction. Choosing a node for constraint propagation is based on representation objects (i.e., the computational data) of this node. This scheme can be viewed as a generalization of the *forward-backward propagation on DAGs* (**FBPD**) algorithm in Chapter 6. Some specific strategies for use in this scheme are described in Section 7.3. They are based on interval arithmetic, affine arithmetic, interval constraint propagation, and linear programming.

Recall that, in Chapter 6, we present the concept of (partial) DAG representations of factorable constraint systems or numerical CSPs (NCSPs). The techniques in this chapter are applicable to both the DAG representation and partial DAG representation notions. For simplicity, we only present these technique for DAG representations, but it is easy to transfer the ideas to partial DAG representations, in the same way we did in Chapter 6. Note that, in this chapter, we use the notations and concepts in Chapter 5, Chapter 6, and Notation 6.8.

7.2.1. Node Range Evaluations

The following constraint system, an NCSP, is used in examples throughout this chapter.

Example 7.1. The DAG representation of the following NCSP is depicted in Figure 7.1:

$$\langle x^2 - 2xy + \sqrt{y} = 0,\ 4x + 3xy + 2\sqrt{y} \leq 9;\ x \in [1,3],\ y \in [1,9]\rangle.$$

The sequence $(\mathbf{N}_1, \mathbf{N}_2, \mathbf{N}_3, \mathbf{N}_4, \mathbf{N}_5, \mathbf{N}_6, \mathbf{N}_7)$ is an ordering of nodes as stated in Theorem 6.6.♣

Consider the DAG representation of a factorable NCSP. Aiming at combining multiple inclusion techniques in constraint propagation, we use multiple inclusion representations at

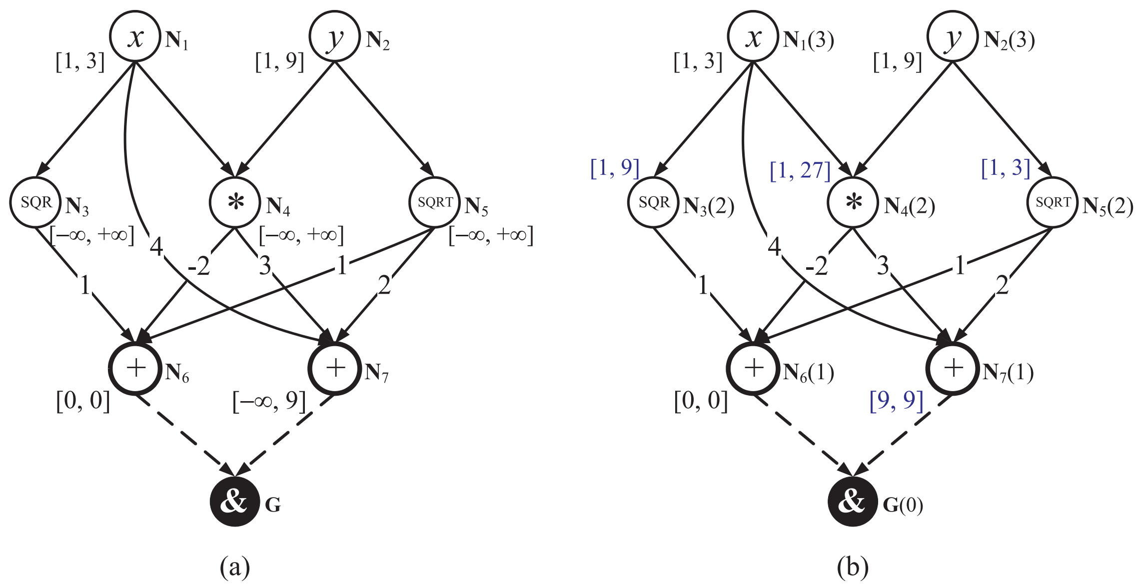

Figure 7.1. The DAG representation: (a) before interval evaluations; and (b) after interval evaluations.

each node $\mathbf{N}$ of the DAG representation. The computational data stored at $\mathbf{N}$ consist of a constraint range $\tau_{\mathbf{N}}$ and multiple representation objects.

Notation 7.2. For each real inclusion representation $\mathcal{I} = (\mathcal{R}, \mu)$ under consideration, one representation object, denoted by $\mathcal{R}(\mathbf{N})$, in $\mathcal{I}$ is stored at $\mathbf{N}$.

Notation 7.3. Let $\mathcal{I} = (\mathcal{R}, \mu)$ be a real inclusion representation, $T \in \mathcal{R}$, $T' \in \mathcal{R}$, and $\mathbf{x} \subseteq \mathbb{R}$. The notation $T \Cap \mathbf{x}$ denotes some object $T'' \in \mathcal{R}$ such that $\mu(T) \cap \mathbf{x} \subseteq \mu(T'')$ holds and either $\mu(T'') \subseteq \mu(T)$ or $\mu(T'') \subseteq \mathbf{x}$ holds. The notation $T \Cap T'$ denotes some object $T'' \in \mathcal{R}$ such that $\mu(T) \cap \mu(T') \subseteq \mu(T'')$ holds and either $\mu(T'') \subseteq \mu(T)$ or $\mu(T'') \subseteq \mu(T')$ holds.

During computations, the inclusion property at each node will be maintained. That is, the exact value of the subexpression at a node $\mathbf{N}$ always lives in the node range $\tau_{\mathbf{N}}$ and in the image $\mu(\mathcal{R}(\mathbf{N}))$ of every representation object $\mathcal{R}(\mathbf{N})$ stored at $\mathbf{N}$. Hereafter, we present a concept that allows reducing the node range of a node basing on the node ranges of its children. This concept is a generalization of the idea of *forward evaluation* [Benhamou *et al.* 1999].

Definition 7.4 (Node Evaluation, `NEV`). Consider the DAG representation of a constraint system and a finite set $\mathcal{S}$ of real inclusion representations. Let $\mathbf{N}$ be a node, $(\mathbf{C}_i)_{i=1}^{k}$ the children of $\mathbf{N}$, and $h : D_h \subseteq \mathbb{R}^k \to \mathbb{R}$ the operation represented by $\mathbf{N}$. Suppose $h_{\mathcal{I}} : \mathcal{R}^k \to \mathcal{R}$ is an inclusion function of the $\{\emptyset\}$-extended function over $\mathbb{R}$ of h. If $\mathbf{N}$ is not the ground, then the following assignment is called the *node evaluation* at $\mathbf{N}$ in $\mathcal{I}$:

$$\mathtt{NEV}(\mathbf{N}, \mathcal{I}) \equiv \left\{ \begin{array}{ll} \mathcal{R}(\mathbf{N}) & := (h_{\mathcal{I}}(\mathcal{R}(\mathbf{C}_1), \ldots, \mathcal{R}(\mathbf{C}_k)) \Cap \tau_{\mathbf{N}}) \Cap \mathcal{R}(\mathbf{N}); \\ \tau_{\mathbf{N}} & := \tau_{\mathbf{N}} \cap \mu(\mathcal{R}(\mathbf{N})); \end{array} \right\}.$$

Example 7.5. Consider the problem in Example 7.1 and the notations in Figure 7.1. At the beginning, we have (see Figure 7.1a):

$$\begin{array}{llll}
\tau_{\mathbf{N}_1} = \mathbb{I}(\mathbf{N}_1) = [1,3]; & \mathbb{A}(\mathbf{N}_1) = \hat{\mathbb{A}}(\mathbf{N}_1) = 2+\epsilon_1; & \\
\tau_{\mathbf{N}_2} = \mathbb{I}(\mathbf{N}_2) = [1,9]; & \mathbb{A}(\mathbf{N}_2) = \hat{\mathbb{A}}(\mathbf{N}_2) = 5+4\epsilon_2; & \\
\tau_{\mathbf{N}_i} = \mathbb{I}(\mathbf{N}_i) = [-\infty,+\infty]; & \mathbb{A}(\mathbf{N}_i) = \hat{\mathbb{A}}(\mathbf{N}_i) = [-\infty,+\infty] & (i=3,4,5); \\
\tau_{\mathbf{N}_6} = \mathbb{I}(\mathbf{N}_6) = [0,0]; & \mathbb{A}(\mathbf{N}_6) = \hat{\mathbb{A}}(\mathbf{N}_6) = 0; & \\
\tau_{\mathbf{N}_7} = \mathbb{I}(\mathbf{N}_7) = [-\infty,9]; & \mathbb{A}(\mathbf{N}_7) = \hat{\mathbb{A}}(\mathbf{N}_7) = [-\infty,9]; &
\end{array}$$

The operation corresponding to $\mathbf{N}_3$ is the square operation; therefore, we have

$$\begin{aligned}
\mathtt{NEV}(\mathbf{N}_3,\mathbb{I}) &\equiv \left\{\begin{array}{ll} \mathbb{I}(\mathbf{N}_3) := \big((\mathbb{I}(\mathbf{N}_1))^2 \Cap \tau_{\mathbf{N}_3}\big) \Cap \mathbb{I}(\mathbf{N}_3); \\ \tau_{\mathbf{N}_3} := \tau_{\mathbf{N}_3} \cap \mathbb{I}(\mathbf{N}_3); \end{array}\right\}, \\
\mathtt{NEV}(\mathbf{N}_3,\hat{\mathbb{A}}) &\equiv \left\{\begin{array}{ll} \hat{\mathbb{A}}(\mathbf{N}_3) := \Big((\hat{\mathbb{A}}(\mathbf{N}_1))^2 \Cap \tau_{\mathbf{N}_3}\Big) \Cap \hat{\mathbb{A}}(\mathbf{N}_3); \\ \tau_{\mathbf{N}_3} := \tau_{\mathbf{N}_3} \cap \mu_{\hat{\mathbb{A}}}(\hat{\mathbb{A}}(\mathbf{N}_3)); \end{array}\right\}, \\
\mathtt{NEV}(\mathbf{N}_3,\mathbb{A}) &\equiv \left\{\begin{array}{ll} \mathbb{A}(\mathbf{N}_3) := \big((\mathbb{A}(\mathbf{N}_1))^2 \Cap \tau_{\mathbf{N}_3}\big) \Cap \mathbb{A}(\mathbf{N}_3); \\ \tau_{\mathbf{N}_3} := \tau_{\mathbf{N}_3} \cap \mu_{\mathbb{A}}(\mathbb{A}(\mathbf{N}_3)); \end{array}\right\}.
\end{aligned}$$

After the node evaluation $\mathtt{NEV}(\mathbf{N}_3,\mathbb{I})$, we have

$$\begin{aligned}
\mathbb{I}(\mathbf{N}_3) &= \big(([1,3])^2 \Cap [-\infty,+\infty]\big) \Cap [-\infty,+\infty] = [1,9], \\
\tau_{\mathbf{N}_3} &= [-\infty,+\infty] \cap [1,9] = [1,9].
\end{aligned}$$

After performing $\mathtt{NEV}(\mathbf{N}_3,\hat{\mathbb{A}})$ and $\mathtt{NEV}(\mathbf{N}_3,\mathbb{A})$ by using affine arithmetic and revised affine arithmetic, we have

$$\begin{aligned}
\hat{\mathbb{A}}(\mathbf{N}_3) &= \big((2+\epsilon_1)^2 \Cap [1,9]\big) \Cap [-\infty,+\infty] = 4.5 + 4\epsilon_1 + 0.5\epsilon_3, \\
\tau_{\mathbf{N}_3} &= [1,9] \cap \mu_{\hat{\mathbb{A}}}(4.5 + 4\epsilon_1 + 0.5\epsilon_3) = [1,9], \\
\mathbb{A}(\mathbf{N}_3) &= \big((2+\epsilon_1)^2 \Cap [1,9]\big) \Cap [-\infty,+\infty] = 4.5 + 4\epsilon_1 + 0.5[-1,1], \\
\tau_{\mathbf{N}_3} &= [1,9] \cap \mu_{\mathbb{A}}(4.5 + 4\epsilon_1 + 0.5[-1,1]) = [1,9].
\end{aligned}$$

Similarly, after performing node evaluations at the other nodes we have

$$\begin{array}{lrcl}
\mathbb{I}(\mathbf{N}_4) = \tau_{\mathbf{N}_4} = [1,27], & \hat{\mathbb{A}}(\mathbf{N}_4) &=& 10 + 5\epsilon_1 + 8\epsilon_2 + 4\epsilon_4, \\
 & \mathbb{A}(\mathbf{N}_4) &=& 10 + 5\epsilon_1 + 8\epsilon_2 + 4[-1,1], \\
\mathbb{I}(\mathbf{N}_5) = \tau_{\mathbf{N}_5} = [1,3], & \hat{\mathbb{A}}(\mathbf{N}_5) &=& 2.125 + \epsilon_2 + 0.125\epsilon_5, \\
 & \mathbb{A}(\mathbf{N}_5) &=& 2.125 + \epsilon_2 + 0.125[-1,1], \\
\mathbb{I}(\mathbf{N}_6) = \tau_{\mathbf{N}_6} = [0,0], & \hat{\mathbb{A}}(\mathbf{N}_6) &=& -13.375 - 6\epsilon_1 - 15\epsilon_2 + 0.5\epsilon_3 - 8\epsilon_4 + 0.125\epsilon_5, \\
 & \mathbb{A}(\mathbf{N}_6) &=& -13.375 - 6\epsilon_1 - 15\epsilon_2 + 8.625[-1,1], \\
\mathbb{I}(\mathbf{N}_7) = \tau_{\mathbf{N}_7} = [9,9], & \hat{\mathbb{A}}(\mathbf{N}_7) &=& 42.25 + 19\epsilon_1 + 26\epsilon_2 + 12\epsilon_4 + 0.25\epsilon_5, \\
 & \mathbb{A}(\mathbf{N}_7) &=& 42.25 + 19\epsilon_1 + 26\epsilon_2 + 12.25[-1,1].
\end{array}$$

In practice, the node range $\tau_{\mathbf{N}}$ and the interval object $\mathbb{I}(\mathbf{N})$ should be merged into one object because they are of the same type. ♣

7.2.2. Induced Constraint Systems for Domain Reduction

In order to present the concept of a *pruning constraint system* concisely (in Definition 7.9), we rely on the following concept.

Definition 7.6 (Inclusion Constraint System, ICS). Let $\mathbf{N}$ be a node of the DAG representation of a constraint system. Suppose $(\mathcal{R}, \mu)$ is a real inclusion representation defined by (5.28), namely, $\mu(T) \equiv \{f_T(V_T) \mid V_T \in D_T[V_T]\}$. The *inclusion constraint system* induced by a representation object $T \equiv \mathcal{R}(\mathbf{N})$ and a *constraint range* $D \subseteq \mathbb{R}$ is defined as

$$\mathtt{ICS}(T, D) \equiv \begin{cases} \{\vartheta_\mathbf{N} \in D_T[\vartheta_\mathbf{N}] \cap D;\} & \text{if } f_T \text{ is identity, where } V_T \equiv (\vartheta_\mathbf{N}); \\ \{f_T(V_T) = \vartheta_\mathbf{N}; V_T \in D_T[V_T]; \vartheta_\mathbf{N} \in D;\} & \text{otherwise;} \end{cases}$$

where the set of variables of $\mathtt{ICS}(T, D)$ consists of the variable $\vartheta_\mathbf{N}$, the variables in V_T, and the auxiliary variables used for describing D_T.

Roughly speaking, the inclusion constraint system induced by a representation object T at a node $\mathbf{N}$ is a set of constraints that can be inferred from the property that the exact value $\vartheta_\mathbf{N}$ of the subexpression represented by $\mathbf{N}$ can be generated by the real evaluation generator f_T at T (because the inclusion $\vartheta_\mathbf{N} \in \mu(T)$ must hold).

Example 7.7. The followings are some examples of inclusion constraint systems for different inclusion representations:

- An inclusion constraint system for the interval form (5.32):

$$\mathtt{ICS}(T, [c, d]) \equiv \{\vartheta_\mathbf{N} \in [c, d] \cap [a, b]\},$$

  where the set of variables is $\{\vartheta_\mathbf{N}\}$. This system is conjunctive and has the form of bound constraint.

- An inclusion constraint system for the interval union form (5.33):

$$\mathtt{ICS}(T, [c, d]) \equiv \left\{\vartheta_\mathbf{N} \in [c, d];\ \vartheta_\mathbf{N} \in \bigcup_{i=1}^{m} [a_i, b_i]\right\},$$

  where the set of variables is $\{\vartheta_\mathbf{N}\}$. This system has the form of the disjunction of bound constraints.

- An inclusion constraint system for the affine form (5.34):

$$\mathtt{ICS}(T, [c, d]) \equiv \left\{x_0 + \sum_{i=1}^{n} x_i \epsilon_i = \vartheta_\mathbf{N};\ (\epsilon_1, \ldots, \epsilon_n) \in [-1, 1]^n;\ \vartheta_\mathbf{N} \in [c, d]\right\},$$

  where the set of variables is $\{\epsilon_1, \ldots, \epsilon_n, \vartheta_\mathbf{N}\}$. This system is conjunctive and linear.

- An inclusion constraint system for the revised affine form (5.36):

$$\mathtt{ICS}(T, [c, d]) \equiv \left\{x_0 + \sum_{i=1}^{n} x_i \epsilon_i + e_x \epsilon_x = \vartheta_\mathbf{N};\ (\epsilon_1, \ldots, \epsilon_n, \epsilon_x) \in [-1, 1]^{n+1};\ \vartheta_\mathbf{N} \in [c, d]\right\},$$

  where the set of variables is $\{\epsilon_1, \ldots, \epsilon_n, \epsilon_x, \vartheta_\mathbf{N}\}$. This system is conjunctive and linear.

- An inclusion constraint system for the Kolev affine form (5.38):

$$\mathtt{ICS}(T,[c,d]) \equiv \left\{ c_x + \sum_{i=1}^{n} x_i\kappa_i + \kappa_x = \vartheta_{\mathbf{N}};\ \kappa_i \in [-v_i, v_i];\ \kappa_x \in [-v_x, v_x];\ \vartheta_{\mathbf{N}} \in [c,d] \right\},$$

  where the set of variables is $\{\kappa_1, \dots, \kappa_n, \kappa_x, \vartheta_{\mathbf{N}}\}$. This system is conjunctive and linear.

- An inclusion constraint system for the Hansen interval form (5.40):

$$\mathtt{ICS}(T,[c,d]) \equiv \left\{ c_x + \sum_{i=1}^{n} x_i\kappa_i = \vartheta_{\mathbf{N}}; \kappa_i \in [-v_i, v_i]; c_x \in [\underline{c}_x, \overline{c}_x]; x_i \in [\underline{x}_i, \overline{x}_i]; \vartheta_{\mathbf{N}} \in [c,d] \right\},$$

  where the set of variables is $\{\kappa_1, \dots, \kappa_n, c_x, x_1, \dots, x_n, \vartheta_{\mathbf{N}}\}$. This system is conjunctive and quadratic.

- An inclusion constraint system for the linear relaxations/polyhedral enclosures (5.41):

$$\mathtt{ICS}(T,[c,d]) \equiv \left\{ a_{i0} + \sum_{j=1}^{n} a_{ij}x_j \le 0\ (i = 1, \dots, m);\ \vartheta_{\mathbf{N}} \equiv x_k \in [c,d];\ (x_1, \dots, x_n) \in \mathbf{B} \right\},$$

  where the set of variables is $\{x_1, \dots, x_n\}$. This system is conjunctive and linear. Note that $\vartheta_{\mathbf{N}}$ is one of the variables $x_1, \dots, x_n$. ♣

Notation 7.8. In the rest of this chapter, we will abuse the notations $\mathbb{I}$ and $\mathbb{A}$ to denote the real inclusion representations, $(\mathbb{I}, \mu_{\mathbb{I}})$ and $(\mathbb{A}, \mu_{\mathbb{A}})$, respectively defined by (5.32) (for interval arithmetic) and by (5.36) (for revised affine arithmetic).

In the next definition, we construct a constraint system, which is used for pruning node ranges, using the representation objects stored at related nodes.

Definition 7.9 (Pruning Constraint System, PCS). Let $\mathbf{N}$ be a node of the DAG representation of a constraint system, $(\mathbf{C}_i)_{i=1}^{k}$ the children of $\mathbf{N}$, $h : \mathbb{R}^k \to \mathbb{R}$ the operation represented by $\mathbf{N}$, and $\mathcal{S}$ a finite set of real inclusion representations. The following constraint system is called the *pruning constraint system* induced by $\mathcal{S}$ at $\mathbf{N}$:

$$\mathtt{PCS}(\mathbf{N},\mathcal{S}) \equiv \begin{cases} \{\bigwedge_{i=1}^{k} \mathtt{ICS}(\mathcal{R}(\mathbf{C}_i), \tau_{\mathbf{C}_i});\} & \text{if } \mathbf{N} \text{ is ground,} \\ \left\{ \begin{array}{l} h(\vartheta_{\mathbf{C}_1}, \dots, \vartheta_{\mathbf{C}_k}) = \vartheta_{\mathbf{N}}; \\ \bigwedge_{(\mathcal{R},\mu)\in\mathcal{S}} \mathtt{PCSub}(\mathbf{N},\mathcal{R},\mu); \end{array} \right\} & \text{otherwise;} \end{cases}$$

where $\mathtt{PCSub}(\mathbf{N},\mathcal{R},\mu)$ is a *pruning constraint subsystem* defined as

$$\mathtt{PCSub}(\mathbf{N},\mathcal{R},\mu) \equiv \left(\mathtt{ICS}(\mathcal{R}(\mathbf{N}), \tau_{\mathbf{N}}) \wedge \bigwedge_{i=1}^{k} \mathtt{ICS}(\mathcal{R}(\mathbf{C}_i), \tau_{\mathbf{C}_i}) \right).$$

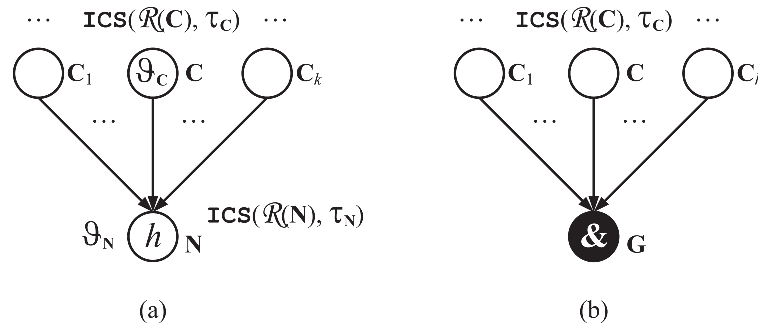

Figure 7.2. A pruning constraint system at $\mathbf{N}$: (a) in case $\mathbf{N}$ is not the ground, all ICS systems and the relation h on the nodes $\mathbf{N}, \mathbf{C}_1, \ldots, \mathbf{C}_k$ are included; and (b) in case $\mathbf{N}$ is the ground, only ICS systems on the child nodes $\mathbf{C}_1, \ldots, \mathbf{C}_k$ are included.

Roughly speaking, the pruning constraint system induced by $\mathcal{S}$ at $\mathbf{N}$ includes all inclusion constraint systems induced by all representation objects (in $\mathcal{S}$) stored at $\mathbf{N}$ and the children of $\mathbf{N}$. The operation h is a constraint itself in the pruning constraint system. The concept of a pruning constraint system is depicted in Figure 7.2.

Example 7.10. Let consider the problem in Example 7.1 and continue the computations in Example 7.5. We have, for instance, the following inclusion constraint systems:

$$
\begin{aligned}
\texttt{ICS}(\mathbb{I}(\mathbf{N}_1), \tau_{\mathbf{N}_1}) &\equiv \{\vartheta_{\mathbf{N}_1} \in [1,3]\};\\
\texttt{ICS}(\mathbb{I}(\mathbf{N}_2), \tau_{\mathbf{N}_2}) &\equiv \{\vartheta_{\mathbf{N}_2} \in [1,9]\};\\
\texttt{ICS}(\mathbb{I}(\mathbf{N}_3), \tau_{\mathbf{N}_3}) &\equiv \{\vartheta_{\mathbf{N}_3} \in [1,9]\};\\
\texttt{ICS}(\mathbb{I}(\mathbf{N}_4), \tau_{\mathbf{N}_4}) &\equiv \{\vartheta_{\mathbf{N}_4} \in [1,27]\};\\
\texttt{ICS}(\mathbb{I}(\mathbf{N}_5), \tau_{\mathbf{N}_5}) &\equiv \{\vartheta_{\mathbf{N}_5} \in [1,3]\};\\
\texttt{ICS}(\mathbb{I}(\mathbf{N}_6), \tau_{\mathbf{N}_6}) &\equiv \{\vartheta_{\mathbf{N}_6} \in [0,0]\};\\
\texttt{ICS}(\mathbb{I}(\mathbf{N}_7), \tau_{\mathbf{N}_7}) &\equiv \{\vartheta_{\mathbf{N}_7} \in [9,9]\};\\
\texttt{ICS}(\hat{\mathbb{A}}(\mathbf{N}_1), \tau_{\mathbf{N}_1}) &\equiv \left\{\begin{array}{l} 2 + \epsilon_1 = \vartheta_{\mathbf{N}_1};\\ \vartheta_{\mathbf{N}_1} \in [1,3];\ \epsilon_1 \in [-1,1]; \end{array}\right\};\\
\texttt{ICS}(\hat{\mathbb{A}}(\mathbf{N}_2), \tau_{\mathbf{N}_2}) &\equiv \left\{\begin{array}{l} 5 + 4\epsilon_2 = \vartheta_{\mathbf{N}_2};\\ \vartheta_{\mathbf{N}_2} \in [1,9];\ \epsilon_2 \in [-1,1]; \end{array}\right\};\\
\texttt{ICS}(\hat{\mathbb{A}}(\mathbf{N}_3), \tau_{\mathbf{N}_3}) &\equiv \left\{\begin{array}{l} 4.5 + 4\epsilon_1 + 0.5\epsilon_3 = \vartheta_{\mathbf{N}_3};\\ \vartheta_{\mathbf{N}_3} \in [1,9];\ \epsilon_i \in [-1,1]\ \ (i = 1,3); \end{array}\right\};\\
\texttt{ICS}(\hat{\mathbb{A}}(\mathbf{N}_4), \tau_{\mathbf{N}_4}) &\equiv \left\{\begin{array}{l} 10 + 5\epsilon_1 + 8\epsilon_2 + 4\epsilon_4 = \vartheta_{\mathbf{N}_4};\\ \vartheta_{\mathbf{N}_4} \in [1,27];\ \epsilon_i \in [-1,1]\ \ (i = 1,2,4); \end{array}\right\};\\
\texttt{ICS}(\hat{\mathbb{A}}(\mathbf{N}_5), \tau_{\mathbf{N}_5}) &\equiv \left\{\begin{array}{l} 2.125 + \epsilon_2 + 0.125\epsilon_5 = \vartheta_{\mathbf{N}_5};\\ \vartheta_{\mathbf{N}_5} \in [1,3];\ \epsilon_i \in [-1,1]\ \ (i = 2,5); \end{array}\right\};\\
\texttt{ICS}(\hat{\mathbb{A}}(\mathbf{N}_6), \tau_{\mathbf{N}_6}) &\equiv \left\{\begin{array}{l} -13.375 - 6\epsilon_1 - 15\epsilon_2 + 0.5\epsilon_3 - 8\epsilon_4 + 0.125\epsilon_5 = \vartheta_{\mathbf{N}_6};\\ \vartheta_{\mathbf{N}_6} \in [0,0];\ \epsilon_i \in [-1,1]\ \ (i = 1,\ldots,5); \end{array}\right\};\\
\texttt{ICS}(\hat{\mathbb{A}}(\mathbf{N}_7), \tau_{\mathbf{N}_7}) &\equiv \left\{\begin{array}{l} 42.25 + 19\epsilon_1 + 26\epsilon_2 + 12\epsilon_4 + 0.25\epsilon_5 = \vartheta_{\mathbf{N}_7};\\ \vartheta_{\mathbf{N}_7} \in [9,9];\ \epsilon_i \in [-1,1]\ (i = 1,2,4,5); \end{array}\right\};\\
\texttt{ICS}(\mathbb{A}(\mathbf{N}_1), \tau_{\mathbf{N}_1}) &\equiv \left\{\begin{array}{l} 2 + \epsilon_1 = \vartheta_{\mathbf{N}_1};\\ \vartheta_{\mathbf{N}_1} \in [1,3];\ \epsilon_1 \in [-1,1]; \end{array}\right\};
\end{aligned}
$$

$$\mathtt{ICS}(\mathbb{A}(\mathbf{N}_2),\tau_{\mathbf{N}_2}) \equiv \left\{\begin{array}{l} 5+4\epsilon_2=\vartheta_{\mathbf{N}_2};\\ \vartheta_{\mathbf{N}_2}\in[1,9];\ \epsilon_2\in[-1,1]; \end{array}\right\};$$

$$\mathtt{ICS}(\mathbb{A}(\mathbf{N}_3),\tau_{\mathbf{N}_3}) \equiv \left\{\begin{array}{l} 4.5+4\epsilon_1+0.5\epsilon_{\mathbf{N}_3}=\vartheta_{\mathbf{N}_3};\\ \vartheta_{\mathbf{N}_3}\in[1,9];\ (\epsilon_1,\epsilon_{\mathbf{N}_3})\in[-1,1]^2; \end{array}\right\};$$

$$\mathtt{ICS}(\mathbb{A}(\mathbf{N}_4),\tau_{\mathbf{N}_4}) \equiv \left\{\begin{array}{l} 10+5\epsilon_1+8\epsilon_2+4\epsilon_{\mathbf{N}_4}=\vartheta_{\mathbf{N}_4};\\ \vartheta_{\mathbf{N}_4}\in[1,27];\ (\epsilon_1,\epsilon_2,\epsilon_{\mathbf{N}_4})\in[-1,1]^3; \end{array}\right\};$$

$$\mathtt{ICS}(\mathbb{A}(\mathbf{N}_5),\tau_{\mathbf{N}_5}) \equiv \left\{\begin{array}{l} 2.125+\epsilon_2+0.125\epsilon_{\mathbf{N}_5}=\vartheta_{\mathbf{N}_5};\\ \vartheta_{\mathbf{N}_5}\in[1,3];\ (\epsilon_2,\epsilon_{\mathbf{N}_5})\in[-1,1]^2; \end{array}\right\};$$

$$\mathtt{ICS}(\mathbb{A}(\mathbf{N}_6),\tau_{\mathbf{N}_6}) \equiv \left\{\begin{array}{l} -13.375-6\epsilon_1-15\epsilon_2+8.625\epsilon_{\mathbf{N}_6}=\vartheta_{\mathbf{N}_6};\\ \vartheta_{\mathbf{N}_6}\in[0,0];\ (\epsilon_1,\epsilon_2,\epsilon_{\mathbf{N}_6})\in[-1,1]^3; \end{array}\right\};$$

$$\mathtt{ICS}(\mathbb{A}(\mathbf{N}_7),\tau_{\mathbf{N}_7}) \equiv \left\{\begin{array}{l} 42.25+19\epsilon_1+26\epsilon_2+12.25\epsilon_{\mathbf{N}_7}=\vartheta_{\mathbf{N}_7};\\ \vartheta_{\mathbf{N}_7}\in[9,9];\ (\epsilon_1,\epsilon_2,\epsilon_{\mathbf{N}_7})\in[-1,1]^3; \end{array}\right\}.$$

Therefore, we can draw the following pruning constraint systems as examples:

$$\mathtt{PCS}(\mathbf{G},\{\hat{\mathbb{A}}\}) \equiv \left\{\begin{array}{l} -13.375-6\epsilon_1-15\epsilon_2+0.5\epsilon_3-8\epsilon_4+0.125\epsilon_5=\vartheta_{\mathbf{N}_6};\\ 42.25+19\epsilon_1+26\epsilon_2+12\epsilon_4+0.25\epsilon_5=\vartheta_{\mathbf{N}_7};\\ \vartheta_{\mathbf{N}_6}\in[0,0];\ \vartheta_{\mathbf{N}_7}\in[9,9];\ \epsilon_i\in[-1,1]\ \ (i=1,\dots,5); \end{array}\right.$$

$$\mathtt{PCS}(\mathbf{G},\{\mathbb{A}\}) \equiv \left\{\begin{array}{l} -13.375-6\epsilon_1-15\epsilon_2+8.625\epsilon_{\mathbf{N}_6}=\vartheta_{\mathbf{N}_6};\\ 42.25+19\epsilon_1+26\epsilon_2+12.25\epsilon_{\mathbf{N}_7}=\vartheta_{\mathbf{N}_7};\\ \vartheta_{\mathbf{N}_6}\in[0,0];\ \vartheta_{\mathbf{N}_7}\in[9,9];\ (\epsilon_1,\epsilon_2,\epsilon_{\mathbf{N}_6},\epsilon_{\mathbf{N}_7})\in[-1,1]^4; \end{array}\right.$$

$$\mathtt{PCS}(\mathbf{N}_6,\{\mathbb{I}\}) \equiv \left\{\begin{array}{l} \vartheta_{\mathbf{N}_3}-2\vartheta_{\mathbf{N}_4}+\vartheta_{\mathbf{N}_5}=\vartheta_{\mathbf{N}_6};\\ \vartheta_{\mathbf{N}_3}\in[1,9];\ \vartheta_{\mathbf{N}_4}\in[1,27];\ \vartheta_{\mathbf{N}_5}\in[1,3];\ \vartheta_{\mathbf{N}_6}\in[0,0]; \end{array}\right.$$

$$\mathtt{PCS}(\mathbf{N}_6,\{\hat{\mathbb{A}}\}) \equiv \left\{\begin{array}{l} \vartheta_{\mathbf{N}_3}-2\vartheta_{\mathbf{N}_4}+\vartheta_{\mathbf{N}_5}=\vartheta_{\mathbf{N}_6};\\ 4.5+4\epsilon_1+0.5\epsilon_3=\vartheta_{\mathbf{N}_3};\\ 10+5\epsilon_1+8\epsilon_2+4\epsilon_4=\vartheta_{\mathbf{N}_4};\\ 2.125+\epsilon_2+0.125\epsilon_5=\vartheta_{\mathbf{N}_5};\\ -13.375-6\epsilon_1-15\epsilon_2+0.5\epsilon_3-8\epsilon_4+0.125\epsilon_5=\vartheta_{\mathbf{N}_6};\\ \vartheta_{\mathbf{N}_3}\in[1,9];\ \vartheta_{\mathbf{N}_4}\in[1,27];\ \vartheta_{\mathbf{N}_5}\in[1,3];\ \vartheta_{\mathbf{N}_6}\in[0,0];\\ (\epsilon_1,\dots,\epsilon_5)\in[-1,1]^5; \end{array}\right.$$

$$\mathtt{PCS}(\mathbf{N}_6,\{\mathbb{A}\}) \equiv \left\{\begin{array}{l} \vartheta_{\mathbf{N}_3}-2\vartheta_{\mathbf{N}_4}+\vartheta_{\mathbf{N}_5}=\vartheta_{\mathbf{N}_6};\\ 4.5+4\epsilon_1+0.5\epsilon_{\mathbf{N}_3}=\vartheta_{\mathbf{N}_3};\\ 10+5\epsilon_1+8\epsilon_2+4\epsilon_{\mathbf{N}_4}=\vartheta_{\mathbf{N}_4};\\ 2.125+\epsilon_2+0.125\epsilon_{\mathbf{N}_5}=\vartheta_{\mathbf{N}_5};\\ -13.375-6\epsilon_1-15\epsilon_2+8.625\epsilon_{\mathbf{N}_6}=\vartheta_{\mathbf{N}_6};\\ \vartheta_{\mathbf{N}_3}\in[1,9];\ \vartheta_{\mathbf{N}_4}\in[1,27];\vartheta_{\mathbf{N}_5}\in[1,3];\ \vartheta_{\mathbf{N}_6}\in[0,0];\\ (\epsilon_1,\epsilon_2,\epsilon_{\mathbf{N}_3},\epsilon_{\mathbf{N}_4},\epsilon_{\mathbf{N}_5},\epsilon_{\mathbf{N}_6})\in[-1,1]^6; \end{array}\right.$$

$$\mathtt{PCS}(\mathbf{N}_7,\{\mathbb{I}\}) \equiv \left\{\begin{array}{l} 4\vartheta_{\mathbf{N}_1}+3\vartheta_{\mathbf{N}_4}+2\vartheta_{\mathbf{N}_5}=\vartheta_{\mathbf{N}_7};\\ \vartheta_{\mathbf{N}_1}\in[1,3];\ \vartheta_{\mathbf{N}_4}\in[1,27];\ \vartheta_{\mathbf{N}_5}\in[1,3];\ \vartheta_{\mathbf{N}_7}\in[9,9]; \end{array}\right.$$

$$\mathtt{PCS}(\mathbf{N}_7,\{\hat{\mathbb{A}}\}) \equiv \left\{\begin{array}{l} 4\vartheta_{\mathbf{N}_1}+3\vartheta_{\mathbf{N}_4}+2\vartheta_{\mathbf{N}_5}=\vartheta_{\mathbf{N}_7};\\ 2+\epsilon_1=\vartheta_{\mathbf{N}_1};\\ 10+5\epsilon_1+8\epsilon_2+4\epsilon_4=\vartheta_{\mathbf{N}_4};\\ 2.125+\epsilon_2+0.125\epsilon_5=\vartheta_{\mathbf{N}_5};\\ 42.25+19\epsilon_1+26\epsilon_2+12\epsilon_4+0.25\epsilon_5=\vartheta_{\mathbf{N}_7};\\ \vartheta_{\mathbf{N}_1}\in[1,3];\ \vartheta_{\mathbf{N}_4}\in[1,27];\ \vartheta_{\mathbf{N}_5}\in[1,3];\ \vartheta_{\mathbf{N}_7}\in[9,9];\\ \epsilon_i\in[-1,1]\ \ (i=1,2,4,5); \end{array}\right.$$

$$\texttt{PCS}(\mathbf{N}_7, \{\mathbb{A}\}) \equiv \begin{cases} 4\vartheta_{\mathbf{N}_1} + 3\vartheta_{\mathbf{N}_4} + 2\vartheta_{\mathbf{N}_5} = \vartheta_{\mathbf{N}_7}; \\ 2 + \epsilon_1 = \vartheta_{\mathbf{N}_1}; \\ 10 + 5\epsilon_1 + 8\epsilon_2 + 4\epsilon_{\mathbf{N}_4} = \vartheta_{\mathbf{N}_4}; \\ 2.125 + \epsilon_2 + 0.125\epsilon_{\mathbf{N}_5} = \vartheta_{\mathbf{N}_5}; \\ 42.25 + 19\epsilon_1 + 26\epsilon_2 + 12.25\epsilon_{\mathbf{N}_7} = \vartheta_{\mathbf{N}_7}; \\ \vartheta_{\mathbf{N}_1} \in [1,3];\ \vartheta_{\mathbf{N}_4} \in [1,27];\ \vartheta_{\mathbf{N}_5} \in [1,3];\ \vartheta_{\mathbf{N}_7} \in [9,9]; \\ (\epsilon_1, \epsilon_2, \epsilon_{\mathbf{N}_4}, \epsilon_{\mathbf{N}_5}, \epsilon_{\mathbf{N}_7}) \in [-1,1]^5. \end{cases}$$

We will see later that, from either $\texttt{PCS}(\mathbf{N}_7, \{\mathbb{I}\})$ or $\texttt{PCS}(\mathbf{N}_7, \{\mathbb{A}\})$, we have $\vartheta_{\mathbf{N}_1} = 1$, $\vartheta_{\mathbf{N}_4} = 1$, and $\vartheta_{\mathbf{N}_5} = 1$. Therefore, the system has a unique solution $(x, y) = (1, 1)$. ♣

7.2.3. CIRD – A Generic Combination Scheme

In this section, we describe a generic combination scheme that combines different real inclusion representations (and related techniques) in constraint propagation. In this scheme, each input constraint system, an NCSP, is represented by a DAG as described in Section 6.2 or Section 6.4. The computational data stored at each node are representation objects as described in Section 7.2.2. In principle, the scheme uses node evaluations and pruning constraint systems, which are defined in Section 7.2.2, and uses relevant domain reduction techniques to reduce node ranges and, in particular, variable domains.

Let $\mathcal{G}$ be the DAG representation of the input constraint system. The proposed scheme, called **CIRD**, uses two waiting lists. The first waiting list, $\mathcal{L}_\mathrm{e}$, stores the nodes waiting for node evaluation. The second waiting list, $\mathcal{L}_\mathrm{p}$, stores the nodes waiting for node range pruning. Note that each node can only appear once at a time in one waiting list, but may appear in both the waiting lists at a time. The set of real inclusion representations for use in the scheme is denoted by $\mathcal{E}$. Suppose each real inclusion representation in $\mathcal{E}$ provides elementary operations that are inclusion functions of their real-valued counterparts. In Algorithm 7.1, we present the main steps of the **CIRD** scheme with inline detailed descriptions.

Example 7.11. Figure 7.3 illustrates the distribution of auxiliary variables (ϵ_1 and ϵ_2) when using affine arithmetic and interval arithmetic to perform node evaluations on the DAG representation of the problem in Example 7.1. The sets $\mathcal{S}$ and $\mathcal{T}$ in Algorithm 7.1 can be chosen as follows: $\mathcal{S} = \mathcal{T} = \{\mathbb{I}, \mathbb{A}\}$. The set $\mathcal{T}$ can be partitioned into two subsets: $\mathcal{U}_1 = \{\mathbb{I}\}$, $\mathcal{U}_2 = \{\mathbb{A}\}$. Example 7.10 gives the pruning constraint systems (PCS) for each node.

Consider the node $\mathbf{N}_7$ (Figure 7.3a). Applying a linear programming technique to the system $\texttt{PCS}(\mathbf{N}_7, \{\mathbb{A}\})$, we get $\epsilon_1 = 1$ and $\epsilon_2 = 1$. Any node involving an auxiliary variable of which the domain has just been reduced, ϵ_1 or ϵ_2, will be in the set $\mathcal{K}$ in Algorithm 7.1, namely, $\mathcal{K} = \{\mathbf{N}_1, \ldots, \mathbf{N}_7\}$. Of course, only a subset $\mathcal{H}$ of $\mathcal{K}$ should be considered for node range updates. More details are described in Section 7.3.5.2. This shows that, in some cases, ***every node in a DAG representation may be chosen for a node range update, not only the descendants of the current node***. Therefore, combination strategies based on the **CIRD** scheme can freely choose the set $\mathcal{H}$ of nodes for node range updates, depending on the nature of the underlying inclusion techniques.

Consider the node $\mathbf{N}_6$ (Figure 7.3b). Suppose we do not applying linear programming techniques to $\texttt{PCS}(\mathbf{N}_6, \{\mathbb{A}\})$, but use some symbolic reasoning. For example, from the first equation of $\texttt{PCS}(\mathbf{N}_6, \{\mathbb{A}\})$, we have an equivalent equation: $\vartheta_{\mathbf{N}_3} = 2\vartheta_{\mathbf{N}_4} - \vartheta_{\mathbf{N}_5} + \vartheta_{\mathbf{N}_6}$. Substituting $\vartheta_{\mathbf{N}_4}$, $\vartheta_{\mathbf{N}_5}$, and $\vartheta_{\mathbf{N}_6}$ from the last three equations of $\texttt{PCS}(\mathbf{N}_6, \{\mathbb{A}\})$ into this equation, we will see that $\vartheta_{\mathbf{N}_3}$ contains not only the auxiliary variable ϵ_1 but also the auxiliary variable ϵ_2, in general.

Algorithm 7.1: **CIRD** – a scheme for **c**ombining **i**nclusion **r**epresentations on **D**AGs

1. **Initialization Phase.**

   (a) **Initial Node Evaluation.** Traverse the DAG representation $\mathcal{G}$ in an order described in Theorem 6.6. When visiting a node $\mathbf{N} \in \mathcal{G}$, perform the node evaluation $\mathtt{NEV}(\mathbf{N}, \mathcal{I})$ for each $\mathcal{I} \in \mathcal{E}$. It is encouraged to merge the assignments of multiple $\mathtt{NEV}(\mathbf{N}, \mathcal{I})$, where $\mathcal{I} \in \mathcal{E}$, into a single process to avoid repeating the same assignments or computations.

   (b) **Initialization of Waiting Lists.** Set $\mathcal{L}_\mathrm{e} := \emptyset$ and $\mathcal{L}_\mathrm{p} :=$ {the list of all nodes representing the running constraints together with all the real inclusion representations of $\mathcal{E}$}.

2. **Propagation Phase.** Repeat this step until both $\mathcal{L}_\mathrm{e}$ and $\mathcal{L}_\mathrm{p}$ become empty or the limit, if any, on the number of iterations is reached.

   (a) **Getting the Next Node.** Get a node $\mathbf{N}$ (and the set $\mathcal{S}$ of real inclusion representations associated with $\mathbf{N}$ in the corresponding list) from the two waiting lists $\mathcal{L}_\mathrm{e}$ and $\mathcal{L}_\mathrm{p}$, according to some *choosing strategy*.

   (b) **Node Evaluation.** Do this step only if $\mathbf{N}$ was taken from $\mathcal{L}_\mathrm{e}$ at Step 2a.
   For each $\mathcal{I} = (\mathcal{R}, \mu) \in \mathcal{S}$, do the following steps (with noticing that combining several inclusion representations, by using inclusion converters, for obtaining better node evaluations is also a good option to consider):

      i. Perform the node evaluation $\mathtt{NEV}(\mathbf{N}, \mathcal{I})$. If this returns an empty set, the algorithm terminates with an infeasible status.

      ii. If the changes of $\mathcal{R}(\mathbf{N})$ and $\tau_\mathbf{N}$ at Step 2(b)i are considered enough to re-evaluate the parents of $\mathbf{N}$, then put each node in $\mathsf{parents}(\mathbf{N})$ (associated with $\mathcal{I}$) into $\mathcal{L}_\mathrm{e}$, if $\mathbf{N}$ is not the ground node, or into $\mathcal{L}_\mathrm{p}$, otherwise.

      iii. If the changes of $\mathcal{R}(\mathbf{N})$ and $\tau_\mathbf{N}$ at Step 2(b)i are considered enough to do a node pruning at $\mathbf{N}$ again, then put $(\mathbf{N}, \mathcal{I})$ into $\mathcal{L}_\mathrm{p}$.

   (c) **Node Pruning.** Do this step only if $\mathbf{N}$ was taken from $\mathcal{L}_\mathrm{p}$ at Step 2a.

      i. Choose a subset $\mathcal{T} \subseteq \mathcal{S}$ such that, for each $\mathcal{I} \in \mathcal{T}$, there are efficient domain reduction techniques for the constraint system $\mathtt{PCS}(\mathbf{N}, \mathcal{I})$.

      ii. Partition $\mathcal{T}$ into subsets such that, for each subset $\mathcal{U}$ of the partition, choose a domain reduction technique which may efficiently reduce the domains of the variables of the system (or a subsystem of) $\mathtt{PCS}(\mathbf{N}, \mathcal{U})$. Afterwards, apply the chosen domain reduction technique to each system (or a subsystem of) $\mathtt{PCS}(\mathbf{N}, \mathcal{U})$ in a certain order. If this process returns an empty set, the algorithm terminates with an infeasible status.

      iii. Let $\mathcal{K}$ be the set of all nodes of which the evaluating functions in the form (5.28) contain some variables of which the domains were reduced at Step 2(c)ii. Choose a subset $\mathcal{H}$ of $\mathcal{K}$ for node range updates, *for example*, such that each node $\mathbf{M}$ in the set $\mathcal{H}$ is a descendant of $\mathbf{N}$.
      Node Range Update: For each node $\mathbf{M}$ in $\mathcal{H}$ and each real inclusion representation $\mathcal{I} = (\mathcal{R}, \mu) \in \mathcal{E}$ such that the representation of $\mu(\mathcal{R}(\mathbf{M}))$ in the form (5.28) contains some variables (in V_T) of which the domains have just been reduced at Step 2(c)ii, update $\mathcal{R}(\mathbf{M})$ by using these newly reduced domains, then update $\tau_\mathbf{M} := \tau_\mathbf{M} \cap \mu(\mathcal{R}(\mathbf{M}))$. If an empty is obtained, the algorithm terminates with an infeasible status.

         A. If the changes of $\mathcal{R}(\mathbf{M})$ and $\tau_\mathbf{M}$ are considered enough to re-evaluate the parents of $\mathbf{M}$, put each node in $\mathsf{parents}(\mathbf{M})$ associated with $\mathcal{I}$ into $\mathcal{L}_\mathrm{e}$.

         B. If the changes of $\mathcal{R}(\mathbf{M})$ and $\tau_\mathbf{M}$ are considered enough to do a node pruning at $\mathbf{M}$, put $(\mathbf{M}, \mathcal{I})$ into $\mathcal{L}_\mathrm{p}$.

This shows that ***the distribution of auxiliary variables in a DAG representation may be changed during computations***. ♣

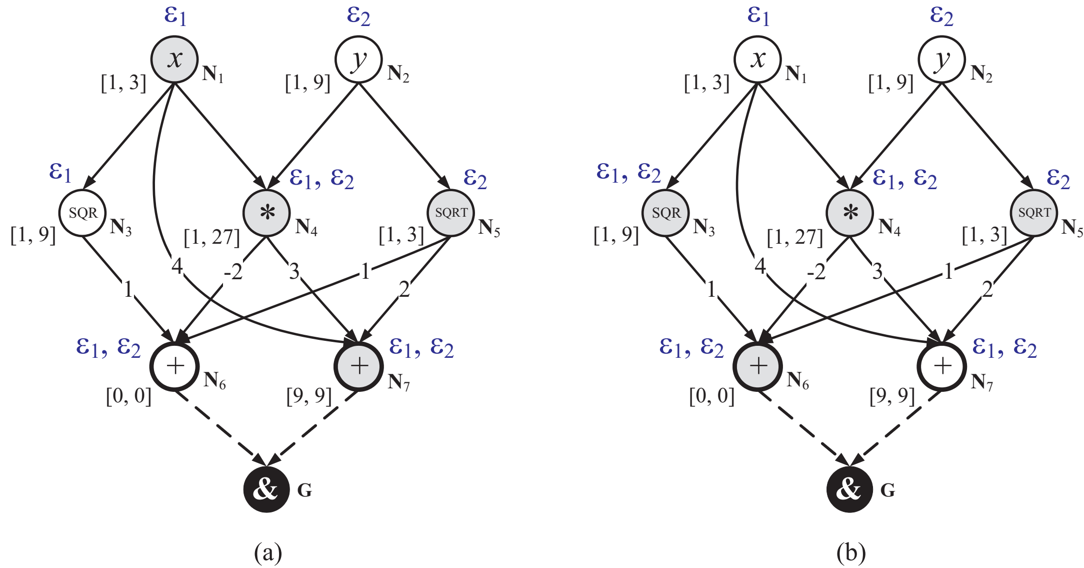

Figure 7.3. The distribution of auxiliary variables (ϵ_1, ϵ_2) in the DAG representation in Example 7.1: the grey nodes are the nodes involving a pruning constraint system (PCS): (a) at $\mathbf{N}_7$; and (b) at $\mathbf{N}_6$.

7.3. Specific Combination Strategies as Instances of **CIRD**

In general, the performance of a particular constraint propagation technique following the **CIRD** scheme depends on the design of each step in this scheme. In this section, we propose some simple strategies for each step in the **CIRD** scheme. These strategies are based on two inclusion representations, $\mathbb{I}$ and $\mathbb{A}$. Note that by combining different strategies at all the steps we get different combination strategies in constraint propagation.

7.3.1. Step 1a: Initial Node Evaluation

A post-order visiting or a recursive evaluation starting from the nodes that represent the running constraints can be used for the traverse at Step 1a. In our implementation, we use a recursive evaluation procedure given in Procedure **RecursiveNodeEvaluation** (page 179) for this purpose. If this procedure exits with an infeasible status, the main algorithm invoking it will terminate with an infeasible status.

Procedure RecursiveNodeEvaluation(in: a node **N)**

if N is a leaf **or N** has been visited **then return** ;
foreach C $\in$ children(**N**) **do RecursiveNodeEvaluation(C)**;
foreach $\mathcal{I} \in \mathcal{E}$ **do** NEV($\mathbf{N}, \mathcal{I}$); ◄ $\mathcal{E}$ is the set of real inclusion representations.
Mark **N** as visited;
if infeasible status is detected **then exit**(infeasible);

Example 7.12. We continue to consider the problem in Example 7.1. Similarly to Example 7.5, after performing the initial node evaluation by using interval arithmetic and revised affine arithmetic, we have:

$$
\begin{aligned}
\tau_{\mathbf{N}_3} &= \mathbb{I}(\mathbf{N}_3) = [1,9]; & \mathbb{A}(\mathbf{N}_3) &= 4.5 + 4\epsilon_1 + 0.5[-1,1];\\
\tau_{\mathbf{N}_4} &= \mathbb{I}(\mathbf{N}_4) = [1,27]; & \mathbb{A}(\mathbf{N}_4) &= 10 + 5\epsilon_1 + 8\epsilon_2 + 4[-1,1];\\
\tau_{\mathbf{N}_5} &= \mathbb{I}(\mathbf{N}_5) = [1,3]; & \mathbb{A}(\mathbf{N}_5) &= 2.125 + \epsilon_2 + 0.125[-1,1];\\
\tau_{\mathbf{N}_6} &= \mathbb{I}(\mathbf{N}_6) = [0,0]; & \mathbb{A}(\mathbf{N}_6) &= -13.375 - 6\epsilon_1 - 15\epsilon_2 + 8.625[-1,1];\\
\tau_{\mathbf{N}_7} &= \mathbb{I}(\mathbf{N}_7) = [9,9]; & \mathbb{A}(\mathbf{N}_7) &= 42.25 + 19\epsilon_1 + 26\epsilon_2 + 12.25[-1,1];
\end{aligned}
$$

See the distribution of auxiliary variables in Figure 7.3. ♣

7.3.2. Step 1b: Initialization of Waiting Lists

We use the process described in the **CIRD** scheme in Algorithm 7.1.

Example 7.13. Consider Example 7.1. After performing this step with $\mathcal{S} = \{\mathbb{I}, \mathbb{A}\}$, we have:

$$
\begin{aligned}
&\mathcal{L}_\mathrm{e} = \emptyset;\\
&\mathcal{L}_\mathrm{p} = \{(\mathbf{N}_6; \mathbb{I}, \mathbb{A}),\ (\mathbf{N}_7; \mathbb{I}, \mathbb{A})\}.
\end{aligned}
$$

♣

7.3.3. Step 2a: Getting the Next Node

At first, we assign a *node level* to each node in the DAG representation of the considered constraint system such that each node has a level smaller than the levels of their descendants. Hence, an ordering in Theorem 6.6 can be obtained easily by sorting the levels of nodes.

Procedure **NodeLevel** (on page 163) is a simple algorithm for computing a vector V_{lvl} of node levels if this procedure is invoked at all nodes representing the running constraints. Figure 7.1 illustrates the node levels for the constraint system given in Example 7.1. The node levels are given in brackets next to the node names. Figure 7.3 illustrates the distribution of auxiliary variables. The list $\mathcal{L}_\mathrm{p}$ is sorted in the ascending order of node levels. This ensures that each node will be taken into pruning processes before its descendants, thus reducing redundant computations. Similarly, the list $\mathcal{L}_\mathrm{e}$ is sorted in the descending order of node levels to ensure that each node will be evaluated before its ancestors.

There are three simple choosing strategies to get the next node from the union of two waiting lists, $\mathcal{L}_\mathrm{e} \cup \mathcal{L}_\mathrm{p}$:

- Get the next node from $\mathcal{L}_\mathrm{p}$ whenever $\mathcal{L}_\mathrm{p}$ is not empty. This strategy is called the *pruning-first strategy*, which gives the priority to the pruning phase.
- Get the next node from one of the two waiting lists until it becomes empty, then switch to the other list. This strategy is called the *loyal choosing strategy.*
- Get the next node from one of the two waiting lists in a rotational manner. This strategy is called the *rotational choosing strategy.*

In our implementation, we use the pruning-first strategy. More complicated strategies for choosing the next node can be used as alternatives, for example, based on the pruning efficiency of nodes. A choosing strategy based on learning tools may also be useful.

7.3.4. Step 2b: Node Evaluation

For the node evaluation at each node $\mathbf{N}$, we can perform $\texttt{NEV}(\mathbf{N}, \mathbb{A})$ and $\texttt{NEV}(\mathbf{N}, \mathbb{I})$ in any order, if $\mathbf{N}$ is not the ground node. At Step 2(b)ii, Step 2(b)iii and Step 2(c)iii, we only count on the change of $\tau_{\mathbf{N}}$ in our current implementation. The change of $\tau_{\mathbf{N}}$ is often considered enough if the ratio of the new width to the old width is less than a number predefined $r_{\mathrm{w}} \in (0, 1)$ and the difference between the old width and the new width is greater than a predefined number $d_{\mathrm{w}} > 0$ (see Section 6.5 for an argument on a similar case). More complicated criteria that have been used in constraint programming can be used as alternatives.

7.3.5. Step 2c: Node Pruning

The subset $\mathcal{T}$ (in the **CIRD**) at this step can be chosen as $\{\mathbb{I}, \mathbb{A}\}$. For node pruning, we use $\texttt{PCS}(\mathbf{N}, \{\mathbb{I}\})$ and a linear subsystem of $\texttt{PCS}(\mathbf{N}, \{\mathbb{A}\})$ defined as follows:

$$\texttt{PCS}(\mathbf{N}, \{\mathbb{I}\}) \equiv \left\{ \begin{array}{l} f(\vartheta_{\mathbf{C}_1}, \ldots, \vartheta_{\mathbf{C}_k}) = \vartheta_{\mathbf{N}}; \\ \vartheta_{\mathbf{N}} \in \tau_{\mathbf{N}}; \\ \bigwedge_{i=1}^{k} (\vartheta_{\mathbf{C}_i} \in \tau_{\mathbf{C}_i}); \end{array} \right\} \quad \text{if } \mathbf{N} \text{ is not ground;}$$

$$\texttt{PCS}_L(\mathbf{N}, \{\mathbb{A}\}) \equiv \left\{ \begin{array}{ll} \left\{ \bigwedge_{i=1}^{k} \texttt{ICS}(\mathbb{A}(\mathbf{C}_i), \tau_{\mathbf{C}_i}) \right\} & \text{if } \mathbf{N} \text{ is ground,} \\ \left\{ \texttt{ICS}(\mathbb{A}(\mathbf{N}), \tau_{\mathbf{N}}) \wedge \bigwedge_{i=1}^{k} \texttt{ICS}(\mathbb{A}(\mathbf{C}_i), \tau_{\mathbf{C}_i}) \right\} & \text{otherwise;} \end{array} \right.$$

where the inclusion constraint systems are defined as

$$\texttt{ICS}(\mathbb{A}(\mathbf{M}), D) \equiv \left\{ \begin{array}{l} x_{\mathbf{M},0} + \sum_{i=1}^{k} x_{\mathbf{M},i}\epsilon_i + e_{\mathbf{M}}\epsilon_{\mathbf{M}} = \vartheta_{\mathbf{M}}; \\ \epsilon_i \in [-1, 1] \ \ (i = 1, \ldots, n); \\ \epsilon_{\mathbf{M}} \in [-1, 1]; \vartheta_{\mathbf{M}} \in D; \end{array} \right\}.$$

In general, we have

$$\texttt{PCS}(\mathbf{N}, \{\mathbb{A}\}) \equiv \texttt{PCS}(\mathbf{N}, \{\mathbb{I}\}) \wedge \texttt{PCS}_L(\mathbf{N}, \{\mathbb{A}\}).$$

The system $\texttt{PCS}_L(\mathbf{N}, \{\mathbb{A}\})$ contains the linear constraints of $\texttt{PCS}(\mathbf{N}, \{\mathbb{A}\})$.

Example 7.14. Consider the problem in Example 7.1, it is depicted in Figure 7.1. Some examples of pruning constraint systems are given in Example 7.10. We then have:

$$\texttt{PCS}_L(\mathbf{G}, \{\mathbb{A}\}) \equiv \left\{ \begin{array}{l} -13.375 - 6\epsilon_1 - 15\epsilon_2 + 8.625\epsilon_{\mathbf{N}_6} = \vartheta_{\mathbf{N}_6}; \\ 42.25 + 19\epsilon_1 + 26\epsilon_2 + 12.25\epsilon_{\mathbf{N}_7} = \vartheta_{\mathbf{N}_7}; \\ \vartheta_{\mathbf{N}_6} \in [0, 0]; \vartheta_{\mathbf{N}_7} \in [9, 9]; \\ (\epsilon_1, \epsilon_2, \epsilon_{\mathbf{N}_6}, \epsilon_{\mathbf{N}_7}) \in [-1, 1]^4; \end{array} \right.$$

$$\texttt{PCS}_L(\mathbf{N}_6, \{\mathbb{A}\}) \equiv \left\{ \begin{array}{l} 4.5 + 4\epsilon_1 + 0.5\epsilon_{\mathbf{N}_3} = \vartheta_{\mathbf{N}_3}; \\ 10 + 5\epsilon_1 + 8\epsilon_2 + 4\epsilon_{\mathbf{N}_4} = \vartheta_{\mathbf{N}_4}; \\ 2.125 + \epsilon_2 + 0.125\epsilon_{\mathbf{N}_5} = \vartheta_{\mathbf{N}_5}; \\ -13.375 - 6\epsilon_1 - 15\epsilon_2 + 8.625\epsilon_{\mathbf{N}_6} = \vartheta_{\mathbf{N}_6}; \\ \vartheta_{\mathbf{N}_3} \in [1, 9]; \ \vartheta_{\mathbf{N}_4} \in [1, 27]; \ \vartheta_{\mathbf{N}_5} \in [1, 3]; \ \vartheta_{\mathbf{N}_6} \in [0, 0]; \\ (\epsilon_1, \epsilon_2, \epsilon_{\mathbf{N}_3}, \epsilon_{\mathbf{N}_4}, \epsilon_{\mathbf{N}_5}, \epsilon_{\mathbf{N}_6}) \in [-1, 1]^6; \end{array} \right.$$

$$\texttt{PCS}_L(\mathbf{N}_7, \{\mathbb{A}\}) \equiv \left\{ \begin{array}{l} 2 + \epsilon_1 = \vartheta_{\mathbf{N}_1}; \\ 10 + 5\epsilon_1 + 8\epsilon_2 + 4\epsilon_{\mathbf{N}_4} = \vartheta_{\mathbf{N}_4}; \\ 2.125 + \epsilon_2 + 0.125\epsilon_{\mathbf{N}_5} = \vartheta_{\mathbf{N}_5}; \\ 42.25 + 19\epsilon_1 + 26\epsilon_2 + 12.25\epsilon_{\mathbf{N}_7} = \vartheta_{\mathbf{N}_7}; \\ \vartheta_{\mathbf{N}_1} \in [1, 3]; \ \vartheta_{\mathbf{N}_4} \in [1, 27]; \ \vartheta_{\mathbf{N}_5} \in [1, 3]; \ \vartheta_{\mathbf{N}_7} \in [9, 9]; \\ (\epsilon_1, \epsilon_2, \epsilon_{\mathbf{N}_4}, \epsilon_{\mathbf{N}_5}, \epsilon_{\mathbf{N}_7}) \in [-1, 1]^5; \end{array} \right.$$

All these systems are linear; hence, we can use linear programming techniques to reduce variable domains, as described in Section 3.3. Tight safe bounds can often be obtained by using the technique proposed by Neumaier and Shcherbina [2004]. ♣

The combination of the following backward propagation and affine pruning techniques results in different strategies for the node pruning phase in the **CIRD** scheme.

7.3.5.1. Backward Propagation

If $\mathbf{N}$ is not the ground, the domains of the variables of the constraint system $\texttt{PCS}(\mathbf{N}, \{\mathbb{I}\})$ can be pruned by a domain reduction technique that is called *backward propagation*, as described in Section 6.3.2. In brief, let h be the elementary operation represented by a node $\mathbf{N}$, we then have the relation $\vartheta_{\mathbf{N}} = f(\vartheta_{\mathbf{C}_1}, \ldots, \vartheta_{\mathbf{C}_k})$. The purpose of backward propagation is to compute, at a low cost, an enclosure of the i-th projection of the relation $\vartheta_{\mathbf{N}} = f(\vartheta_{\mathbf{C}_1}, \ldots, \vartheta_{\mathbf{C}_k})$ onto $\vartheta_{\mathbf{C}_i}$ by using interval arithmetic. Only k nodes $\mathbf{C}_1, \ldots, \mathbf{C}_k$ possibly contain the variables of which the domains are pruned in the above backward propagation. Hence, after performing this backward propagation, at Step 2(c)iii we only need to consider k nodes $\mathcal{H} = \{\mathbf{C}_1, \ldots, \mathbf{C}_k\}$ for node range updates and for putting into the waiting lists.

7.3.5.2. Affine Pruning

In revise affine arithmetic, $\mathbb{A}$, each variable of the input constraint system is associated with one noise symbol ϵ_i (for $i = 1, \ldots, n$). The system $\texttt{PCS}_L(\mathbf{N}, \{\mathbb{A}\})$ is a linear constraint system; therefore, the domains of the variables of $\texttt{PCS}_L(\mathbf{N}, \{\mathbb{A}\})$ can be pruned by using a linear programming technique supplemented with the technique proposed by Neumaier and Shcherbina [2004] (see Section 3.3). If the operation represented by $\mathbf{N}$ is linear, we can apply a linear programming technique to $\texttt{PCS}(\mathbf{N}, \{\mathbb{A}\})$, instead of $\texttt{PCS}_L(\mathbf{N}, \{\mathbb{A}\})$, to get tighter bounds on the variables. For efficiency, only the domains of the variables $\vartheta_{\mathbf{C}_1}, \ldots, \vartheta_{\mathbf{C}_k}$ and/or $\epsilon_1, \ldots, \epsilon_n$ are needed to be pruned. We can devise three possible pruning strategies for Step 2(c)iii:

1. The first strategy only requires to prune the domains of $\vartheta_{\mathbf{C}_1}, \ldots, \vartheta_{\mathbf{C}_k}$;
2. The second strategy only requires to prune the domains of auxiliary variables $\epsilon_1, \ldots, \epsilon_n$;
3. The third strategy is to prune the domains of both $\{\vartheta_{\mathbf{C}_1}, \ldots, \vartheta_{\mathbf{C}_k}\}$ and $\{\epsilon_1, \ldots, \epsilon_n\}$.

For the first strategy, we only need to consider $\mathcal{H} = \{\mathbf{C}_1, \ldots, \mathbf{C}_k\}$ for node range updates. For the last two strategies, the set $\mathcal{H}$ can be chosen as any subset of the set of $\mathsf{descendants}(\mathbf{N})$ of which the noise variables in $\mu_{\mathbb{A}}$ have just been pruned. In our implementation, we use the second pruning strategy with two options for $\mathcal{H}$: (*i*) the set $\mathsf{descendants}(\mathbf{N})$; and (*ii*) the set of initial variables associated with $\epsilon_1, \ldots, \epsilon_n$.

If, for each $i \in \{1, \ldots, n\}$, the new domain of the noise variable ϵ_i is $[a_i, b_i] \subseteq [-1, 1]$, then the node range update at $\mathbf{M} \in \mathcal{H}$ will be

$$\tau_{\mathbf{M}} := \tau_{\mathbf{M}} \cap \left(x_0^{\mathbf{M}} + \sum_{i=1}^{n} x_i^{\mathbf{M}} [a_i, b_i] + e_{\mathbf{M}}[-1, 1] \right), \tag{7.1}$$

where $\mathbb{A}(\mathbf{M}) = x_0^{\mathbf{M}} + \sum_{i=1}^{n} x_i^{\mathbf{M}} \epsilon_i + e_{\mathbf{M}}[-1, 1]$, and $x_i^{\mathbf{M}}$ (for $i = 1, \ldots, n$) are real numbers. It seems that Kolev affine forms (Section 2.2.3.4) are more suitable for the node range update (7.1) than revised affine forms are because the radius of the domain of a noise variable is allowed to be less than 1. Hence, Kolev affine forms do not need to use the domain $[-1, 1]$ when restarting the computations at $\mathbf{M}$. In this case, we only need to replace the previous node range $[a_i, b_i]$ with the smallest symmetric interval containing $[a_i, b_i]$.

Remark 7.15. The cost of linear programming is high; therefore, we should use the affine pruning technique only if the pruning ratio is predicted to be high. For this, we propose to use the affine pruning technique only if the absolute accumulative error $e_{\mathbf{M}}$ of each node $\mathbf{M}$ involving the above linear systems is small enough. That is, the value of the operation represented by $\mathbf{M}$ lies in a thin slot between two hyperplanes, $x_0^{\mathbf{M}} + \sum_{i=1}^{n} x_i^{\mathbf{M}}\epsilon_i - e_{\mathbf{M}}$ and $x_0^{\mathbf{M}} + \sum_{i=1}^{n} x_i^{\mathbf{M}}\epsilon_i + e_{\mathbf{M}}$, in the space of the noise variables $(\epsilon_1, \ldots, \epsilon_n)$. Moreover, the affine pruning should only be used for nodes at low levels (i.e., the nodes near roots).

7.4. Experiments

7.4.1. Comparisons with Interval Constraint Propagation Techniques

We have carried out experiments on an implementation of the **CIRD[ai]** algorithm (an instance of the **CIRD** scheme using affine arithmetic, interval arithmetic, interval constraint propagation, and linear programming) and two other well-known state-of-the-art interval constraint propagation techniques. The first one is a variant of box consistency [Benhamou *et al.* 1994] in a well-known commercial product named ILOG Solver (v6.0, 11/2003), hereafter denoted by **BOX**. The second one is the **HC4** algorithm [Benhamou *et al.* 1999] (see Section 3.2.2.2). The experiments are carried out on 33 problems that are *unbiasedly* chosen and divided into five test cases to analyze the empirical results:

- The test case T_1 consists of eight easy problems with isolated solutions. These problems are solvable in short time by the search using the three propagators. (See Section C.2.)
- The test case T_2 consists of four average problems with isolated solutions. These problems are solvable by the search using **CIRD[ai]** and **BOX** and cause the search using **HC4** being out of time without reaching 10^6 splits. (See Section C.3.)
- The test case T_3 consists of eight hard problems with isolated solutions. These problems cause the search using **HC4** being out of time without reaching 10^6 splits and cause the search using **BOX** either being out of time or being stopped due to running more than 10^6 splits. The search using **CIRD[ai]** accomplishes the solving for six of eight problems in this test case and runs more than 10^6 splits for the other two problems. (See Section C.4.)
- The test case T_4 consists of seven easy problems with a continuum of solutions. These problems are solvable in short time at the predefined precision 10^{-2}. (See Section C.5.)
- The test case T_5 consists of six hard problems with a continuum of solutions. These problems are solvable in short time at the predefined precision 10^{-1}. (See Section C.6.)

The timeout value is set to **10 hours** for all the test cases. ***The timeout values will be used as the running time for the techniques that are out of time in the next result analysis*** (i.e., we are in favor of slow techniques). For the first three test cases, the precision is 10^{-4} and the search to be used is the bisection search. For the last two test cases, the search technique to be used is the **UCA6** algorithm for NCSPs with inequality constraints (see Chapter 4). The comparison of interval constraint propagation techniques is based on the following measures:

- *The running time:* The relative ratio of the running time of each propagator to that of **CIRD[ai]** is called the *relative time ratio*.
- *The number of boxes:* The relative ratio of the number of boxes in the output of each propagator to that of **CIRD[ai]** is called the *relative cluster ratio*.
- *The number of splits:* The number of splits in search needed to solve the problems. The relative ratio of the number of splits used by each propagator to that of **CIRD[ai]** is called the *relative iteration ratio*.
- *The volume of boxes (only for T_1, T_2, T_3):* We consider the reduction per dimension $\sqrt[d]{V/D}$; where d is the dimension, V is the total volume of the output boxes, D is the volume of the initial domains. The relative ratio of the reduction gained by each propagator to that of **CIRD[ai]** is called the *relative reduction ratio*.
- *The volume of inner boxes (only for T_4, T_5):* The ratio of the volume of inner boxes to the volume of all output boxes is called the *inner volume ratio*.

The lower the relative ratio is, the better the performance/quality is; and the higher the inner volume ratio is, the better the quality is.

The overviews of results in our experiments are given in Table 7.1 and Table 7.2. Clearly, **CIRD[ai]** is superior than **BOX** and **HC4** in performance and quality measures for the problems with isolated solutions in the unbiasedly chosen benchmarks. **CIRD[ai]** still outperforms the others for the problems with continuums of solutions in the benchmarks, while being a little better than the others in quality measures. Note that the ratios for the test case T_3 are in fact much higher than shown because the solving processes using **BOX** and **HC4** does not terminates after **10 hours** while the one using **CIRD[ai]** terminates in seconds or minutes.

Table 7.1. A comparison of three constraint propagation techniques **CIRD[ai]**, **BOX**, and **HC4** in solving NCSPs. In the section (a), the averages of the relative time ratios are taken over all the problems in the test cases T_1, T_2, T_3; and the averages of the other relative ratios are taken over the problems in the test case T_1 (i.e., taken over the problems that are solvable by all the techniques). In the section (b), the averages of the relative ratios are taken over all the problems in the test cases T_4, T_5.

Prop. ▼	(a) Isolated Solutions				(b) Continuum of Solutions			
	Relative time ratio	Relative reduction ratio	Relative cluster ratio	Relative iteration ratio	Relative time ratio	Inner volume ratio	Relative cluster ratio	Relative iteration ratio
CIRD[ai]	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**0.945**	**1.000**	**1.000**
BOX	1429.660	5.323	30.206	4.263	3.414	0.944	1.102	1.056
HC4	17283.614	7.722	105.825	5.515	60.101	0.941	1.168	1.118

Table 7.2. The averages of the relative time ratios, which are taken over problems in each test case.

Propagator	(a) Isolated Solutions			(b) Continuum of Solutions	
▼	Test case T_1	Test case T_2	Test case T_3	Test case T_4	Test case T_5
CIRD[ai]	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
BOX	8.33	6097.45	517.10	2.33	4.68
HC4	54.47	83009.81	1649.66	31.42	93.56

7.4.2. Comparisons with Linear Relaxation Based Techniques

Encouraged by the comparison in the previous section, we compare the proposed technique with a very recent mathematical solution technique, called **A2**, in [Kolev 2002], which was specially designed to solve a nonlinear equation system $f(x) = 0$. The **A2** algorithm converts this system into *separable form* $g(x) = 0$, and then uses Kolev affine arithmetic to evaluate $g(x)$ and get a linear form $\mathcal{L}(x, y) = -Ax + By + b,\ x \in \mathbf{x},\ y \in \mathbf{y}$; where A and B are real matrices, b is a real vector, and $\mathbf{x}$ and $\mathbf{y}$ are interval vectors. This technique has to assume a posterior-condition that the matrix A is *invertible* in order to use the domain reduction rule (3.115) of the form $\mathbf{x} := \mathbf{x} \cap (A^{-1}B\mathbf{y} + A^{-1}b)$. The reader can find more details in Section 3.3.3.2. No *rigorous rounding control* is found in [Kolev 2002]. We take the first problem in [Kolev 2002], which was used for illustrating the power of the **A2** algorithm in [Kolev 2002], for our comparison:

$$\begin{cases} ((4x_3 + 3x_6)x_3 + 2x_5)x_3 + x_4 = 0, \\ ((4x_2 + 3x_6)x_2 + 2x_5)x_2 + x_4 = 0, \\ ((4x_1 + 3x_6)x_1 + 2x_5)x_1 + x_4 = 0, \\ x_4 + x_5 + x_6 + 1 = 0, \\ (((x_2 + x_6)x_2 + x_5)x_2 + x_4)x_2 + (((x_3 + x_6)x_3 + x_5)x_3 + x_4)x_3 = 0, \\ (((x_1 + x_6)x_1 + x_5)x_1 + x_4)x_1 + (((x_2 + x_6)x_2 + x_5)x_2 + x_4)x_3 = 0, \\ x_1 \in [0.0333, 0.2173],\ x_2 \in [0.4000, 0.6000], \\ x_3 \in [0.7826, 0.9666],\ x_4 \in [-0.3071, -0.1071], \\ x_5 \in [1.1071, 1.3071],\ x_6 \in [-2.1000, -1.9000]. \end{cases} \tag{7.2}$$

This system has a unique solution. It is known to be very hard for interval constraint propagation techniques. To solve it on a 1.7 GHz Pentium PC at the precision 10^{-5} using a bisection search; **A2** has to perform 917 iterations in 3.46 seconds to reduce the problem to five boxes (see [Kolev 2002]); while an instance of the **CIRD** scheme, called **CIRD[ai]**, performs 54 iterations in only 0.118 seconds to reduce the problem to three boxes. Hence, **CIRD[ai]** is about 29.3 times faster than **A2** for the system (7.2), while it is more rigorous and accurate than **A2**.

Another technique to compare with is a very recent technique, called **Quad**, in [Lebbah *et al.* 2003b], which was specifically designed to process quadratic constraints, and then extended to address power terms in [Lebbah *et al.* 2003a]. A short description of **Quad** is presented in Section 3.3.1.1. Note that this technique only work on power terms, thus only applying to the systems with (many) power terms. Again, we take as example two problems, called *Gough-Steward* and *Yama196*, which were used to illustrate the power of **Quad** in [Lebbah *et al.* 2003b] and [Lebbah *et al.* 2003a], respectively. *Gough-Steward* is a non-sparse quadratic equation system of nine variables in robotics, which has four solutions [Lebbah *et al.* 2003b]:

$$\begin{cases} x_1^2 + y_1^2 + z_1^2 = 31, \\ x_2^2 + y_2^2 + z_2^2 = 39, \\ x_3^2 + y_3^2 + z_3^2 = 29, \\ x_1x_2 + y_1y_2 + z_1z_2 + 6x_1 - 6x_2 = 51, \\ x_1x_3 + y_1y_3 + z_1z_3 + 7x_1 - 2y_1 - 7x_3 + 2y_3 = 50, \\ x_2x_3 + y_2y_3 + z_2z_3 + x_2 - 2y_2 - x_3 + 2y_3 = 34, \\ -12x_1 + 15y_1 - 10x_2 - 25y_2 + 18x_3 + 18y_3 = -32, \\ -14x_1 + 35y_1 - 36x_2 - 45y_2 + 30x_3 + 18y_3 = 8, \\ 2x_1 + 2y_1 - 14x_2 - 2y_2 + 8x_3 - y_3 = 20, \\ x_1 \in [-2.00; 5.57],\ y_1 \in [-5.57, 2.70],\ z_1 \in [0.00, 5.57], \\ x_2 \in [-6.25, 1.30],\ y_2 \in [-6.25, 2.70],\ z_2 \in [-2.00, 6.25], \\ x_3 \in [-5.39, 0.70],\ y_3 \in [-5.39, 3.11],\ z_3 \in [-3.61, 5.39]. \end{cases} \tag{7.3}$$

Yama196 is a series of high-dimensional sparse problems of n variables and n equations:

$$(n+1)^2 x_{i-1} - 2(n+1)^2 x_i + (n+1)^2 x_{i+1} + e^{x_i} = 0,\ x_i \in [-10, 10] \quad (\text{for } i = 1, \ldots, n),$$

where $x_0 = x_{n+1} = 0$. Similarly to [Lebbah *et al.* 2003a], we use the precision 10^{-8} for these problems. Table 7.3 presents a preliminary comparison between **CIRD[ai]** and **Quad**.

Table 7.3. A preliminary comparison of **Quad** and **CIRD[ai]**, n is the number of variables.

Propagator ▶	**Quad**				**CIRD[ai]**				Time ratio
Problem ▼	$\#S$	$\#B$	Time (sec.)	CPU speed (GHz)	$\#S$	$\#B$	Time (sec.)	CPU speed (GHz)	$\frac{\text{Quad}}{\text{CIRD[ai]}}$
Gough-Steward ($n = 9$)	24	4	183.0	1.0	912	4	2.7	1.7	39.9
Yama196 ($n = 30$)	108	16	31.4	2.66	25	2	3.8	1.7	12.9
Yama196 ($n = 60$)	n/a	n/a	n/a	n/a	18	2	21.0	1.7	n/a
Yama196 ($n = 100$)	n/a	n/a	n/a	n/a	20	2	85.8	1.7	n/a
Yama196 ($n = 200$)	n/a	n/a	n/a	n/a	19	2	560.2	1.7	n/a
Yama196 ($n = 300$)	n/a	n/a	n/a	n/a	20	2	1878.1	1.7	n/a

The results of **Quad** in Table 7.3 are copied from [Lebbah *et al.* 2003a,b], except that the ones in the cells filled with "n/a" are not yet available due to our limited access to the code of **Quad**. In Table 7.3, $\#S$ denotes the number of splits and $\#B$ denotes the number of boxes in the output.

Remark 7.16. We have also carried out experiments on the naive use of variants of affine arithmetic as a replacement of interval arithmetic in interval constraint propagation. That is, affine arithmetic is used only to get bounds on subexpression like in interval arithmetic. However, the performance of obtained techniques (using affine arithmetic) is even (at least two times) worse than their counterparts (using interval arithmetic).

7.5. Potential Directions for CIRD

The **CIRD** scheme opens several potential directions for future research:

- Replace the linear programming technique in the affine pruning (in **CIRD[ai]**) with any domain reduction techniques for linear systems because linear programming is very expensive for this purpose. For example, we can use the Krawczyk iteration for linear equations (Section 3.1.1.1), the interval Gauss-Seidel iteration (Section 3.1.1.2), the interval Gauss elimination (Section 3.1.2.1), the hull method (Section 3.1.2.2), and all the methods described in Section 3.3.
- Implement the new divisions for affine forms proposed by [Kolev 2002] and [Miyajima *et al.* 2003] in place of the division $x/y := x * (1/y)$ implemented in **CIRD[ai]**.
- Integrate Kolev generalized affine arithmetic (with the improvements of the multiplication presented in Chapter 5) into the **CIRD** scheme. (See also Section 2.2.3.4.)
- Integrate linear relaxation techniques (e.g., the ones in [Hongthong and Kearfott 2004] and [Borradaile and Van Hentenryck 2004]) into the **CIRD** scheme.
- Integrate the *quadratic form* proposed by [Messine 2002] and **Quad** into the **CIRD** scheme.

- Investigate intelligent choosing strategies of the **CIRD** scheme: choose the next node in waiting lists based on the pruning efficiency of nodes. The idea is to follow the way of predicting the convergence in [Lebbah and Lhomme 2002]. Learning the behavior of node range reductions is also an interesting direction.

- Investigate the ability to integrate high-order inclusion techniques, such as the convexification techniques proposed by [Jansson 2000] and [Tawarmalani and Sahinidis 2002], into the **CIRD** scheme. Rigorous bounds on polynomials can be obtained by using the technique proposed in [Garloff *et al.* 2003].

7.6. Conclusion

In summary, our contribution in this chapter is twofold:

1. We propose **a novel generic scheme, called CIRD, to combine multiple inclusion techniques in numerical constraint propagation**. The scheme potentially allows bringing into the constraint propagation framework the strengths of different techniques coming from different fields. It uses real inclusion representations (proposed in Chapter 5) on either DAG representations (proposed by Schichl and Neumaier [2004b], see Chapter 6) or partial DAG representations (Chapter 6). This enables devising fine-grained and flexible combination strategies for performing constraint propagation on virtually any factorable constraint system, including numerical constraint satisfaction problems.

2. We devise from the generic scheme **several specific combination strategies for numerical constraint propagation**. These strategies are designed to combine interval constraint propagation, interval arithmetic, affine arithmetic, and linear programming into the framework of constraint propagation. Our experiments on a particular strategy, called **CIRD[ai]**, show that the new approach outperforms previously available constraint propagation techniques by 1 to 4 orders of magnitude or more in speed,[2] while still being better in quality measures. It even outperforms some very recent techniques that are specially designed to solve special constraint systems. The largest acceleration is for well-constrained problems. This shows that the fine-grained structure made available on (partial) DAG representations through the abstract inclusion concept, called the *real inclusion representation*, is a very important and original contribution.

Additionally, our experiments show that the strengths of the **FBPD** algorithm (proposed in Chapter 6) and the **CIRD[ai]** algorithm are complementary. Namely, the **FBPD** algorithm outperforms the **CIRD[ai]** algorithm by an order of magnitude or more in speed when solving NCSPs with continuums of solutions. Conversely, the **CIRD[ai]** algorithm is superior than the **FBPD** algorithm by 1 to 3 orders of magnitude or more in speed when solving numerical CSPs with isolated solutions. Theoretically, it is easy to combine these two algorithms to solve numerical CSPs, in most cases, at the highest speed of both because the resulting combination algorithm only need to *heuristically* predict if the constraints are equality then resort to the **CIRD[ai]** algorithm, otherwise resort to the **FBPD** algorithm. A failure of this prediction does not impact on the correctness of the combination algorithm, but only on the performance of the algorithm. Further predictions can be based on the nature of real world applications.

[2] Our observations show that the gain in speed quickly increases when the hardness of problems increases.

Chapter 8

Clustering Techniques for Disconnected Solution Sets

Note: This chapter includes the research conducted jointly with Djamila Sam-Haroud and Boi Faltings in [Vu *et al.* 2004a]. This is a contribution of the thesis to promote interval constraint solvers.

8.1. Introduction

In practice, constraints can be equations or inequalities of arbitrary type, usually expressed by using arithmetic and logical expressions. Numerical constraint satisfaction problems (NCSPs) with *continuums of solutions* are often encountered in real-world engineering applications. In such applications, a set of non-isolated solutions often expresses relevant alternatives that need to be identified as precisely and as completely as possible. *Interval constraint solvers* take as input an NCSP and generate a collection of boxes of which the union *rigorously* encloses the solution set. They have shown their ability to solve some complex instances of NCSPs with non-isolated solutions, especially in low dimensional space. However, they provide enclosures that are still prohibitively verbose for further exploitations rather than just a simple query[1]. More complex queries, on connectedness or intersection for example, are however central to many applications. Although the running time of interval constraint solvers is getting gradually improved, the space complexity (the number of boxes) is at least proportional to the number of boxes needed to cover the boundary of the solution set, therefore still very verbose.

One of the applications of interval constraint solvers we are investigating is a cooperation scheme of optimization and constraint satisfaction, where the verbose output data of the solvers is usually simplified by imposing a limit on the number of boxes to be produced or by restricting the solving process to a low predefined precision. However, in case the solution set is complex and disconnected, the above simplifications affect significantly the quality of the output and make it unsuitable for practical use. We propose to use fast *clustering* techniques to regroup a large collection of output boxes into a reduced collection of boxes each of which tightly covers one or several connected subsets.

The needs for progress on this direction are also ubiquitous in real-world applications such as collision detection, self-collision detection in molecular dynamics and dynamic systems, where the systems have multiple components in a low dimensional space and the reduced

[1] In [Agarwal *et al.* 2001a], worst-case query time of a *box-tree* in d-dimensions is $\Theta(N^{1-1/d} + k)$, where N is the number of boxes and k is the number of boxes intersecting the *query range*.

collection of boxes is useful for soon guaranteeing that systems have no collisions; hence, further expensive computations can be avoided in many cases. To name a few, we cite some recent works in collision detection [Fahn and Wang 1999, 2003; Ganovelli *et al.* 2000; Haverkort *et al.* 2002; Larsson and Akenine-Moller 2001; van den Bergen 1997; Zhou and Suri 1999] that have showed that using *bounding-volume techniques* could gain good results. These techniques aim at computing bounding boxes that are very similar to the output from interval constraint solvers for NCSPs. In such applications, interval constraint solvers can be used to produce *conservative* approximations of the solution sets within acceptable time and clustering techniques can be used to bridge the gap between the verboseness in the output of interval constraint solvers and the compactness required by the applications.

It is known that general computation of an optimal set of k clusters is NP-complete [Garey and Johnson 1979]. Therefore, fast clusterings usually can be achieved only by using heuristic algorithms or by restricting to approximations. Very recently, a fast clustering algorithm named **AntClust** [Labroche *et al.* 2002] was proposed. This algorithm is close to, but still not suitable for addressing our requirements. Though successful, the general clustering techniques are not suitable for directly applying to our case because they are not guaranteed to be convergent and the homogeneity information in our problems is not available as required. Therefore, specific clustering techniques for the data produced by interval constraint solvers are needed.

This chapter considers the issue of *post-processing* the output of interval constraint solvers for further exploitations when solving numerical CSPs with continuums of solutions. In particular, we first present in Section 8.3 a basic clustering algorithm, called **Colonization**, which takes $\mathcal{O}(dN^2)$ time to cluster N boxes in d-dimensions. The basic method computes *homogeneity* (i.e., the connectedness in this case) for each pair of boxes. Moreover, most search techniques implemented in interval constraint solvers follow the *branch-and-prune* framework; hence, they essentially produce boxes in the tree structure.[2] Taking advantages of the tree structure, we propose two approximate algorithms to cluster a large collection of boxes organized in the form of a tree. The first algorithm, called **MCC**, very quickly performs clusterings such that no connected subsets are partitioned. The second algorithm, called **SDC**, generates more adaptive clusterings in very short time. It can be seen as a further process of **MCC**. In Section 8.3.5, we also discuss about combining the proposed algorithms in order to reduce the running time of the basic algorithm. The conclusion is finally given in Section 8.5.

8.2. Goals of Clustering

8.2.1. Basic Concepts

When solving NCSPs with non-isolated solutions, interval-based search techniques following the branch-and-prune framework produce a large collection of boxes that can be maintained in a *bounding-box tree*, where child nodes represent for branchings in search. The bounding boxes stored at search nodes are results of the pruning phase. In literature, there are some variants of bounding-box tree like *bounding-volume tree*, *interval tree*, *box-tree* and *AABB tree*.

We call the boxes at the tree leaves that is produced by the solvers the *primitive boxes*. In this paper, we use the concept of a *hull of boxes* to refer to the smallest box that contains all the given boxes and, for convenience, also to refer to the collection of the involved boxes if not confusing. A collection of primitive boxes is called *connected* if the union of the boxes is

[2] Even if the tree structure is not available, we still can construct a good bounding-box tree from a list of boxes in $\mathcal{O}(N \log N)$ time [Agarwal *et al.* 2001a].

a connected set. If this is the case, we say that the primitive boxes connect to each other. A collection of primitive boxes is called *max-connected* (w.r.t to the set of all primitive boxes) if it is connected and no other primitive boxes connect to the given boxes. A clustering is called *max-connected* if each connected collection in a cluster is max-connected. A bounding-box tree is called *orthogonal-separable* if every decomposition of each bounding-box into pairwise disconnected collections (of primitive boxes) can be performed by sequentially by using separating hyper-planes that are orthogonal to axes (see Figure 8.1).

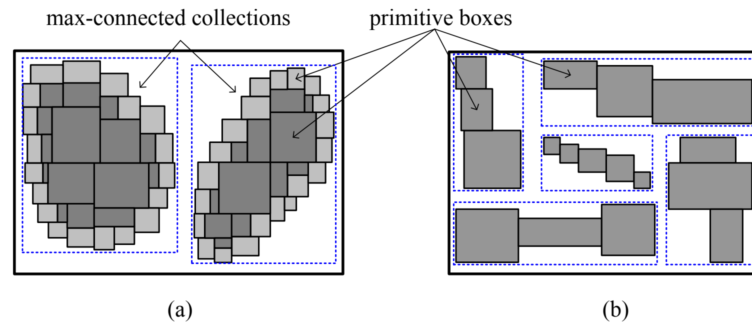

Figure 8.1. (a) This tree is orthogonal-separable: grey boxes are primitive boxes, they form two max-connected collections; (b) This tree is not orthogonal-separable.

8.2.2. Goal Setting

We recall that a collection of sets representing points is called *disjoint* if every two objects have no common points. Two sets are called *hull-disjoint* if the two hulls of them are disjoint. The solution set of an NCSP with non-isolated solutions usually consists of one or more connected subsets each of which is a continuum. In many applications, there is a need for enclosing the solution set by a collection of disjoint bounding boxes such that each of the connected subsets is contained in only one bounding box. To describe the connected subsets as well as possible, the number of bounding boxes should be as big as possible.

Interval-based search techniques usually produce approximations that are either very poor (if being stopped at low precision), or prohibitively verbose (if being stopped at medium/high precision) when solving problems with non-isolated and disconnected solution sets. If a certain quality is required, we obviously need to take the latter setting and then do a clustering on the large collection of boxes. However, the exact solution set is unknown yet, only a collection of primitive boxes is known. The above need is then translated into the need for a max-connected clustering. If a max-connected clustering has the maximum number of clusters, we call it an *optimal max-connected clustering*[3]. The optimal max-connected clustering of an orthogonal-separable bounding-box tree provides pairwise hull-disjoint clusters each of which is a hull of exact one max-connected collection of primitive boxes (see Figure 8.7a). In case the bounding-box tree is not orthogonal-separable, the clusters in the optimal max-connected clustering may be not hull-disjoint (see Figure 8.7b). In this case, we may need a further decomposition in order to obtain hull-disjoint, if this property is required by applications (see Figure 8.7b).

In the next subsections, we propose three new algorithms and their combinations to address different goals. The first subsection focuses on a basic algorithm that computes the optimal max-connected clustering. It is however not very efficient in practice, we then incrementally

[3] It is easy to prove that the optimal max-connected clustering exists uniquely.

propose two alternative algorithms. Each algorithm has two phases, but they have the same first phase. The second subsection describes this common phase. The third subsection describes an algorithm for computing max-connected clusters that are hull-disjoint. The fourth subsection gives an algorithm for computing more adaptive clusterings. In the last subsection, we discuss about combining the proposed algorithms in order to reduce the running time of computing the optimal max-connected clustering.

8.3. Algorithms

8.3.1. Optimal Max-Connected Clustering

As far as we know, there does not exist any algorithm that are suitable to find the optimal max-connected clustering. Fortunately, there exists a recent clustering algorithm named **AntClust** [Labroche *et al.* 2002] that exploits the phenomenon known as *colonial closure* of ants to create homogeneous groups of individuals. The **AntClust** algorithm addresses the general clustering problem in the way that we can borrow a part for addressing our goals. Inspired by Seidel's *invasion* algorithm [Tsang 1993] and some ideas in the **AntClust** algorithm, we propose a simple deterministic algorithm, called **Colonization**, which is given in Figure 8.1 to compute the optimal max-connected clustering. This basic algorithm check if each unprocessed box connects to existing collections. The check is taken on each member box of each collection to see if the unprocessed box connect to the collection. The other processes are described in detailed in Figure 8.1.

It is easy to see that in the worst-case the first phase takes $d(1+2+\cdots+(N-1)) = dN(N-1)/2$ checks for connectedness of two intervals, where N is the number of primitive boxes in d-dimensions. As a result, the time complexity of the first phase is $\mathcal{O}(dN^2)$. At the end of the first phase, all produced collections are max-connected and they have no common

Algorithm 8.1: The **Colonization** algorithm – a naive clustering

```
Phase 1 begin                         ◄ The first phase is to obtain the optimal max-connectedness.
          ℒ := ∅;
          foreach box B not in any collection in ℒ do
              C := {B};
              foreach C_i ∈ ℒ do
                  if {B} ∪ C_i is connected then
                      C := C ∪ C_i;
                      ℒ := ℒ \ {C_i};
                  end
              end
              ℒ := ℒ ∪ {C};
          end
        end
Phase 2 begin                         ◄ the optional phase to obtain the hull-disjointness.
          Replace each collection by the hull of its primitive boxes;
          while There exist two hulls that have nonempty intersection do
              Combine the two hulls into a single hull;
          end
        end
```

points, but their hulls are not guaranteed to be disjoint. The number of collections produced by the first phase is therefore equal to the maximum number, p, of max-connected collections. The second phase is optional. It is only for the applications that requires hull-disjoint. The second phase has the time complexity $\mathcal{O}(dp^2)$. We obviously have $p \leq N$; hence, the total time complexity of the **Colonization** algorithm is $\mathcal{O}(dN^2)$, or $\mathcal{O}(N^2)$ if d is fixed. In practice, $p \ll N$ and p is bounded for fixed problems.

8.3.2. Separator Computation

In order to cluster the primitive boxes into a number of disjoint hulls, we can use hyper-planes that are orthogonal to axes to separate the primitive boxes. We then define new notations for computing such separating hyper-planes as follows.

Definition 8.1 (Separator, SPT). Given a hull, $\mathbf{H}$, of primitive boxes, an axis $x \in \mathbb{N}$, and an interval, $I \in \mathbb{I}_\circ$. The couple (x, I) is called a *separator* of $\mathbf{H}$ if I is a maximum interval (w.r.t. the set inclusion) that satisfies the following condition for each primitive box, $\mathbf{B}$, in $\mathbf{H}$: $I \subseteq \mathbf{H}|_x \wedge I \cap \mathbf{B}|_x = \emptyset$. When this holds, I is called a *separating interval* and x is called a *separating axis*. The set of all separators of $\mathbf{H}$ on axis x is denoted by $\texttt{SPT}(\mathbf{H}, x)$. $\texttt{SPT}(\mathbf{H}) = \cup_x \texttt{SPT}(\mathbf{H}, x)$.

Definition 8.2 (Extension, EXT). Given a hull, $\mathbf{H}$, of primitive boxes, an axis $x \in \mathbb{N}$, and a box $\mathbf{B} \subseteq \mathbf{H}$. A couple (x, I) is called an *extension* of $\mathbf{B}$ w.r.t. $\mathbf{H}$ (on axis x) if I is an interval that is maximum w.r.t. the set inclusion in $\mathbf{H}|_x \setminus \mathbf{B}|_x$. When this holds, I is called an *extending interval* and x is called an *extending axis*. We denote by $\texttt{EXT}(\mathbf{B}, x)$ the set of extensions of $\mathbf{B}$ (w.r.t. $\mathbf{H}$) on axis x.

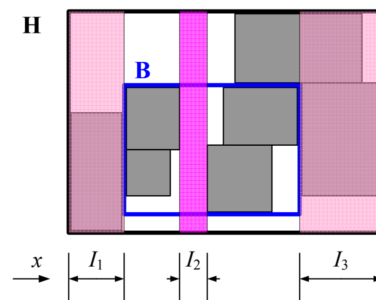

Figure 8.2. The grey boxes are primitive; (x, I_2) is a separator of $\mathbf{B}$ and $\mathbf{H}$; (x, I_1) and (x, I_3) are extensions of $\mathbf{B}$ to $\mathbf{H}$ on axis x.

Figure 8.2 gives an illustration of the two above concepts. It is easy to see that, on each axis, a hull has at most two extensions w.r.t. its parent hull and that all separators of the hull lie between the two extensions. In a bounding-box tree of primitive boxes, we do a process in a bottom-up manner (e.g., in post-order) to make each bounding box to be the hull of its child bounding boxes. Hence, the box at each node becomes the hull of primitive boxes contained in it. A bounding-box tree of which the bounding boxes are the hulls of primitive boxes contained in the bounding boxes is called a *fitted tree*.

Definition 8.3 (Separating Set, SE). In a fitted tree, given an axis $x \in \mathbb{N}$ and a hull, $\mathbf{H}$, at a node. The union of the intervals of all separators and extensions of $\mathbf{H}$ on axis x is called the *separating set* of $\mathbf{H}$ on axis x and denoted by $\mathtt{SE}(\mathbf{H}, x)$. Note that, if not specified, the extensions of a hull are taken w.r.t. the parent hull in the tree.

For simplicity, we use the same notations SPT, EXT and SE for the tree node corresponding to the hull $\mathbf{H}$. The computation of extensions of a hull w.r.t. its parent hull is trivial and takes $\mathcal{O}(d)$ time. We can see that, in a fitted tree, a couple (x, I) is a separator of a hull if and only if I is a maximum interval (w.r.t. the set inclusion) that can be contained in all the separating sets on axis x of all children of the hull. It means that all the separators of the hull can be computed from separators and extensions of its children by intersection. Moreover, this is true even during the above-mentioned bottom-up process. The ordering relation on the set of separators and extensions is given in the following proposition.

Proposition 8.4. In a fitted tree, given a hull, $\mathbf{H}$, of primitive boxes. We have that any two different separating/extending intervals of $\mathbf{H}$ on the same axis do not connect to each other. Moreover, for any two separators/extensions, s_1 and s_2, of $\mathbf{H}$, either $s_1 = s_2$, $s_1 < s_2$ or $s_2 < s_1$ holds; where the ordering relation on the separators and extensions is defined as follows:

$$(i) \qquad (x_1, I_1) = (x_2, I_2) \qquad \Leftrightarrow \qquad x_1 = x_2 \wedge I_1 = I_2,$$

$$(ii) \qquad (x_1, I_1) < (x_2, I_2) \qquad \Leftrightarrow \qquad x_1 < x_2 \vee (x_1 = x_2 \wedge I_1 < I_2).$$

Proof. By Definition 8.1 and Definition 8.2, one can easily see that every two different separators/extensions of $\mathbf{H}$ on the same axis do not connect to each other, otherwise either the separators/extensions are not maximum w.r.t to the set inclusion or $\mathbf{H}$ is not the hull of its children. As a result, the set of all separating and extending intervals on an axis is totally ordered. This results in what we have to prove. ∎

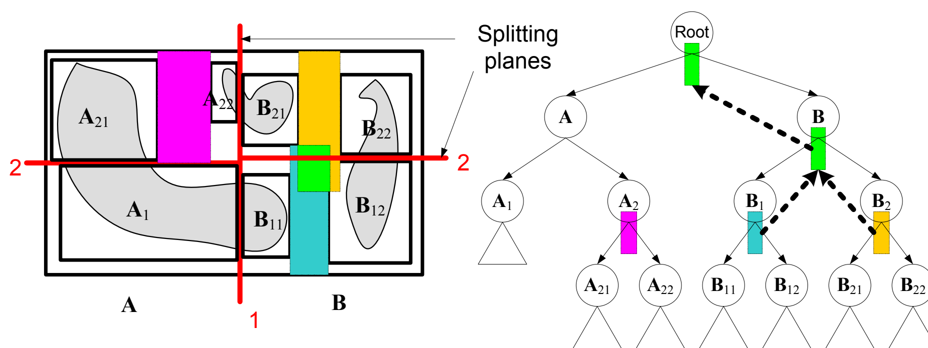

Figure 8.3. In the fitting process on a bounding-box tree: the computation of separators is processed in bottom-up manner.

To compute separators of all nodes in a bounding-box tree, we only need to do a single bottom-up process, called the *fitting process*. In summary, in the fitting process, operations at each node consist of (i) making the box at the current node to be the hull of its children; and (ii) computing the ordered set of separators by taking the ordered intersection of ordered separating

sets of its children. The details of the fitting process is given in Procedure **BoxTreeFitting** on page 196. We denote by m the maximum number of children of a tree node. The operation (i) is trivial and takes $\mathcal{O}(md)$ time. The leaf nodes of the tree have no separators. Each node in the tree has at most two extensions that straddle the box stored at the node (hence, they straddle the set of separators). Moreover, the bottom-up process starts at leaf nodes; therefore, the separating sets can be computed by the intersection in the operation (ii) and maintained sorted by the total order in Proposition 8.4.

At the current node, we denote by $q_{i,j}$ the number of elements in the separating set on axis $i \in \mathbb{N}$ of the j-th child hull, where $1 \leq i \leq d, 1 \leq j \leq m$. We denote $\bar{q}_j = \sum_i q_{i,j}$ and $\bar{q} = \max_j\{\bar{q}_j\}$, then $\bar{q}$ is the maximum number of separators/extensions that a child of the current node can have. Computing the intersection of a sorted collection of k_1 pairwise disconnected intervals and another sorted collection of k_2 pairwise disconnected intervals takes $\mathcal{O}(k_1 + k_2)$ time (see Algorithm 8.2). By Proposition 8.4 and the result of Procedure **SeparatorPulling** in Procedure **BoxTreeFitting**, $q_{i,j}$ separating/extending intervals in the separating set on axis i of the j-th child are pairwise disconnected and totally ordered. Therefore, the time complexity of computing the intersection of $\sum_j q_{i,j}$ separating/extending intervals on axis i of all children is linear in the number of intervals, that is, $\mathcal{O}(\sum_j q_{i,j})$. As a result, the time complexity of the operation (ii) is $\mathcal{O}(\sum_i \sum_j q_{i,j}) = \mathcal{O}(\sum_j \sum_i q_{i,j}) = \mathcal{O}(\sum_j \bar{q}_j)$, that is, not greater than $\mathcal{O}(m\bar{q})$. Because the number of nodes in the tree is $\mathcal{O}(N)$, the total time complexity of the fitting process is $\mathcal{O}(mdN + Q)$, where Q is the total number of separators/extensions in the tree except in the root. This can not exceed $\mathcal{O}((md + q)N)$, where q is the maximum number of separators/extensions that each node in the tree has. In most existing solvers, m is not greater than $2d + 1$ and is usually small in comparison with $2d + 1$ (e.g., $m = 2$ if bisection is used). We conjecture that q is bounded for fixed problems; in particular, it is bounded by $\mathcal{O}(p)$, where p is given in Section 8.3.1. If this conjecture is true, as our experiments show, the time complexity of the fitting process is $\mathcal{O}(N)$ for fixed problems.

Algorithm 8.2: The $\mathbb{I}_\circ$-**intersect** algorithm – intersection of two sets of intervals

---

Input: two ordered sets of disjoint open/closed intervals $\{\langle\alpha_i^{\langle 1\rangle}, \beta_i^{\langle 1\rangle}\rangle \in \mathbb{I}_\circ \mid 1 \leq i \leq k_1\}$ and $\{\langle\alpha_j^{\langle 2\rangle}, \beta_j^{\langle 2\rangle}\rangle \in \mathbb{I}_\circ \mid 1 \leq j \leq k_2\}$.

Output: an ordered set $\mathcal{I}$ of disjoint open/closed intervals as the interval intersection.

$\mathcal{I} := \emptyset$; $i := 1$; $j := 1$;

while $i \leq k_1 \wedge j \leq k_2$ **do**

- **if** $\neg\beta_i^{\langle 1\rangle} \leq \alpha_j^{\langle 2\rangle}$ **then**
  - $i := i + 1$; **continue while**;
- **end**
- **if** $\neg\beta_j^{\langle 2\rangle} \leq \alpha_i^{\langle 1\rangle}$ **then**
  - $j := j + 1$; **continue while**;
- **end**
- $\alpha := \max\{\alpha_i^{\langle 1\rangle}, \alpha_j^{\langle 2\rangle}\}$; $\beta := \min\{\beta_i^{\langle 1\rangle}, \beta_j^{\langle 2\rangle}\}$;
- $\mathcal{I} := \mathcal{I} + \{\langle\alpha, \beta\rangle\}$;
- **if** $\beta = \beta_i^{\langle 1\rangle}$ **then** $i := i + 1$;
- **if** $\beta = \beta_j^{\langle 2\rangle}$ **then** $j := j + 1$;

end

return $\mathcal{I}$;

The $\mathbb{I}_\circ$-**intersect** algorithm (in Algorithm 8.2) computes the intersection of two ordered collections of pairwise disconnected intervals. The intersection of multiple collections is computed by calling this function multiple times.

Procedure BoxTreeFitting(in/out: a bounding-box tree $\mathcal{T}_0$)

```
foreach node P of T_0 visited a post-order do
  | if P is a leaf then
  |   | foreach axis x do SPT(P, x) := ∅;
  | else
  |   | SeparatorPulling(P);                                  ◄ On page 196.
  | end
end
```

Procedure SeparatorPulling(in/out: a node **P**)

```
if P is a leaf then return ;     ◄ Needed only for Procedure SubtreeSeparation on page 198.
C := children(P);
Set the box at P to the hull □{B ∈ I_∘^d | B is a box at a node in C};   ◄ d-dimensional.
foreach C ∈ C and each axis x do
  | Compute EXT(C, x) and denote it by {E_l, E_u};            ◄ This may be empty.
  | SE(C, x) := {E_l, SPT(C, x), E_u};                        ◄ The result is an ordered set.
end
foreach axis x do    ▼/* I_∘-intersect is called multiple times, axis x is ignored to get intervals. */
  | SPT(P, x) := I_∘-intersect{SE(C, x) | C ∈ C};             ◄ See Algorithm 8.2.
end
SPT(P) := {SPT(P, 1), ..., SPT(P, d)};                        ◄ Construct an ordered set.
```

8.3.3. Max-Connected Clustering

The second phase is called the *separating process*. We now compute a max-connected clustering based on next propositions. During the separating process, we maintain max-connected collections of primitive boxes in the form of fitted trees.

Proposition 8.5. Any separator stored at the root of a fitted tree representing a max-connected collection can be used to partition this tree into two fitted subtrees each of which represents a max-connected collection.

Proposition 8.6. If a set of primitive boxes represented by a fitted tree can be partitioned by a hyper-plane that is orthogonal to axis x into two collections of which the projections on x do not connect to each other, then the root of the fitted tree has some separating interval that contains the projection of the hyper-plane on x.

Proof of these propositions is trivial due to Definition 8.1 and the definition of fitted tree. By these propositions, we can compute a max-connected clustering from the separators computed

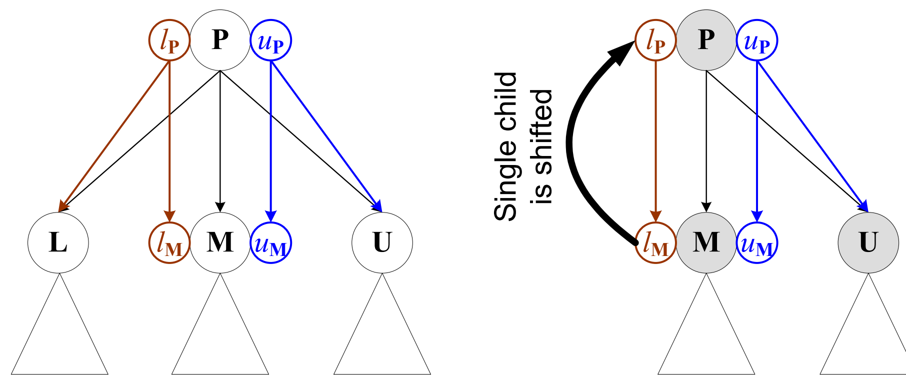

Figure 8.4. The separating process: recursively construct lower-bound and upper-bound trees from the subtrees on the lower and upper sides respectively of a separator, and from lower-bound and upper-bound trees of the children that are intersected with the separator.

in Section 8.3.2. The process can be done in bottom-up manner, or simply by a recursive call at the root. For simplicity, we describe the process in recursive mode. A recursive call starts from a root node of a fitted tree created during the separating process. Recursively, for each node $\mathbf{P}$ and the current separator $S \in \texttt{SPT}(\mathbf{P})$, we construct two fitted trees from the subtree rooted at $\mathbf{P}$. One tree is called *lower-bound tree* (denoted by $l_{\mathbf{P}}$), the other is *upper-bound tree* (denoted by $u_{\mathbf{P}}$) with single roots being copied from $\mathbf{P}$ at first. If the box at a child node lies on lower-bound (respectively upper-bound) side of S, the subtree rooted at the child node is moved to $l_{\mathbf{P}}$ ($u_{\mathbf{P}}$, respectively). Otherwise the child node, called $\mathbf{M}$, is processed similarly to construct its lower-bound and upper-bound trees, $l_{\mathbf{M}}$ and $u_{\mathbf{M}}$ respectively. The trees $l_{\mathbf{M}}$ and $u_{\mathbf{M}}$ are then attached to $l_{\mathbf{P}}$ and $u_{\mathbf{P}}$ respectively. If any tree root in this process has only one child, this child is shifted to the root. Additionally, we make roots of lower-bound and upper-bound trees to be hulls of their child nodes. The separators of the root nodes of new lower- and upper-bound trees are updated from its children.

In Figure 8.5, we describe the *max-connected clustering* (**MCC**) algorithm that uses the separating process to get a hull-disjoint max-connected clustering. For simplicity, in the **MCC** algorithm we use only one list, $\mathcal{L}$, however in implementation it should be maintained as two lists: one list for the trees that have no separators at the root and the other list for the rest. In Function **SubtreeSeparation**, we describe the details of the separating process for a subtree rooted at a node in a fitted tree. Figure 8.4 and Figure 8.5 give some illustrations of the separating process. Figure 8.7b gives another example on which **MCC** provides the optimal max-connected clustering. Without difficult, we can prove, basing on Proposition 8.6, that the obtained clustering is optimal max-connected if the bounding-box tree is orthogonal-separable.

Algorithm 8.5: The max-connected clustering (**MCC**) algorithm

Input: a bounding-box tree $\mathcal{T}_0$.
Output: a list $\mathcal{L}$ of fitted trees decomposed from $\mathcal{T}_0$.
BoxTreeFitting($\mathcal{T}_0$); ◀ On page 196.
$\mathcal{L} := \{\mathcal{T}_0\}$;
while $\exists \mathcal{T} \in \mathcal{L}$: $\mathcal{T}$ has at least one separator, S, at its root **do**
| $\mathcal{L} := (\mathcal{L} \setminus \mathcal{T}) \cup$ **SubtreeSeparation**(root($\mathcal{T}$), S); ◀ On page 198.
end

Function SubtreeSeparation(in: a node $\mathbf{P}$, a separator $S \in \mathtt{SPT}(\mathbf{P})$; **out:** $l_{\mathbf{P}}$, $u_{\mathbf{P}}$)

```
Remove the separator S from SPT(P);
Create two trees, l_P and u_P, each has a single node copied from P;
foreach C ∈ children(P) do
    if C lies on lower side of S then
        Move the subtree rooted at C to a new subtree of the root of l_P;
    else if C lies on upper side of S then
        Move the subtree rooted at C to a new subtree of the root of u_P;
    else
        Find the separator S' ∈ SPT(C) : S ⊆ S';
        (l_C, u_C) := SubtreeSeparation(C, S');      ◀ Repeat this process for child nodes.
        Attach l_C as a new subtree of the root of l_P;
        Attach u_C as a new subtree of the root of u_P;
    end
end
if any root in the trees l_P or u_P has only one child then Shift this child to the root;
SeparatorPulling(root(l_P));                          ◀ On page 196.
SeparatorPulling(root(u_P));                          ◀ On page 196.
```

As proved in Section 8.3.2, the time complexity of the fitting process is $\mathcal{O}(mdN + Q)$. There are at most Q separators in the trees, each separator is processed once, each process takes $\mathcal{O}(m)$ time. Therefore, the time complexity of the separating process is $\mathcal{O}(mQ)$. The total time complexity of **MCC** is hence $\mathcal{O}(mdN + mQ)$. This is practically shown to be linear in N for fixed problems.

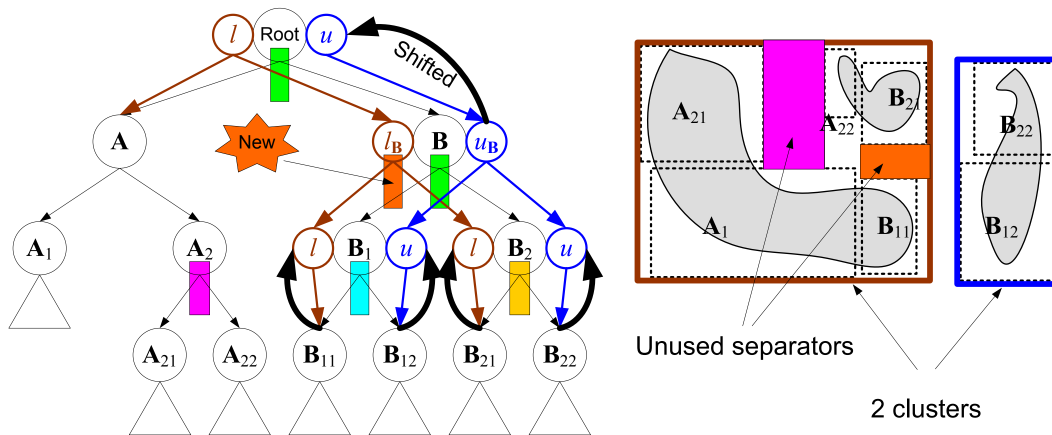

Figure 8.5. **MCC**: The separators exist in the root of the fitted tree are used for recursive separating the tree into lower-bound and upper-bound subtrees. The figure on the right gives the result of this process. Two obtained clusters are hull-disjoint and max-connected.

8.3.4. Separator-Driven Clustering

In some applications, the clustering obtained by the **MCC** algorithm characterizes the solution set not well enough (see the example in Figure 8.5). A remedy for this problem is given as an additional process for **MCC** so that applications can choose to whether to run it. We observe

that the separators that still exist in output fitted trees of the **MCC** algorithm (e.g., the two separators on the right side in Figure 8.5) can be used as hints for further separations.

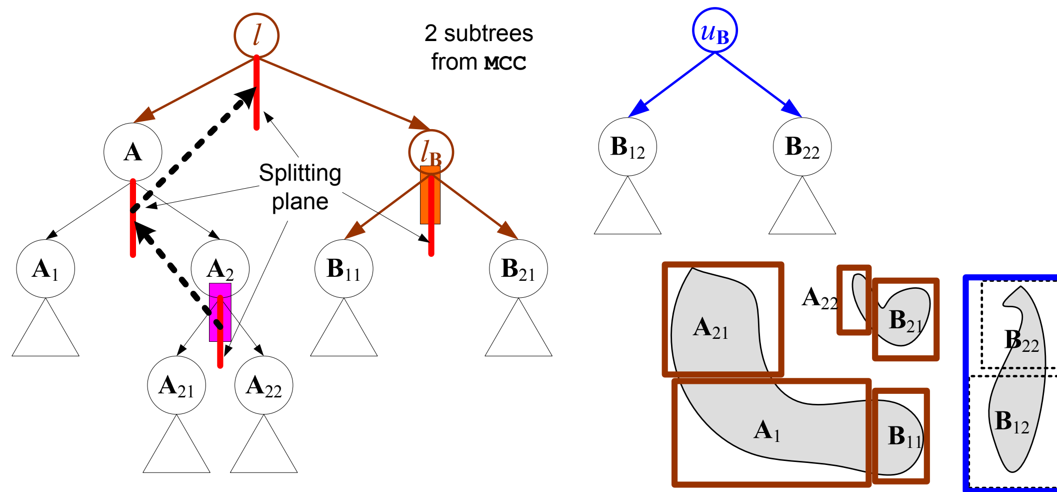

Figure 8.6. **SDC**: The separators that still exist in subtrees after running **MCC** are considered as hints for decomposition. The process produces six clusters (boxes) that are pairwise hull-disjoint, but not all are max-connected.

Algorithm 8.7: The separator-driven clustering (**SDC**) algorithm

---

Input: a bounding-box tree $\mathcal{T}_0$.
Output: a list $\mathcal{L}$ of fitted trees decomposed from $\mathcal{T}_0$.
$\mathcal{L} := \emptyset$;
$\mathcal{L}_0 :=$ **MCC**$(\mathcal{T}_0)$; ◀ See Algorithm 8.5.
foreach fitted tree $\mathcal{T} \in \mathcal{L}_0$ **do**
| $\mathcal{L} := \mathcal{L} \cup$ **MaxSeparation**$(\mathcal{T})$; ◀ On page 200.
end

If we consider the remaining separators in the fitted trees of the output of **MCC** as hints for further decomposition, we will need to use all the splitting hyper-planes (i.e., the branchings in the trees) in the paths from the tree nodes of these separators upward to the roots of the trees for the separation. That is, all siblings of the nodes from the current node upward to the root are to be separated into different groups/collections in the clustering. The use of splitting hyper-planes for the separation does not guarantee that the clusters are pairwise disconnected. In Figure 8.7, we describe the main steps of an algorithm, called the *separator-driven clustering* (**SDC**) algorithm, which performs the above idea as an additional process for the **MCC** algorithm. It is obvious that this algorithm produces a hull-disjoint clustering. Figure 8.6 gives the result of the further process for the subtrees in Figure 8.5. Six clusters (boxes) obtained in Figure 8.6 describe the solution set better than two boxes in Figure 8.5. In Figure 8.7b, we give another example on the **SDC** algorithm where the solution set consisting of two connected subsets is well covered by 13 boxes each of which is a hull of primitive boxes. The **SDC** algorithm quickly provides an adaptive clustering as shown in our experiments. By an argument similar to the one in Section 8.3.3, we have the time complexity of the **SDC** algorithm is $\mathcal{O}(mdN + mQ)$. This is also practically shown to be linear in N for fixed problems.

Function MaxSeparation(**in:** a tree $\mathcal{T}$; **out:** a list $\mathcal{L}$ of trees decomposed from $\mathcal{T}$)

```
Search in post-order, from left to right, for a node N0 such that SPT(N0) ≠ ∅;
if not found then return L := {T};
L := ∅; N := N0; P := parent(N);                                  ◂ P may be null.
while P ≠ ∅ do
    foreach C ∈ children(P) do
        if C on the left of N then  Move the subtree rooted at C to L;
        if C on the right of N then
            Detach the subtree rooted at C and make a new tree T_C;
            L := L ∪ MaxSeparation(T_C);          ◂ Repeat this process for a child node.
        end
        if C = N ≠ N0 then  Erase the node C;
        if C = N = N0 then
            Detach the subtree rooted at C and make a new tree T_C;
            Find a separator S ∈ SPT(C);
            L := L ∪ SubtreeSeparation(root(T_C), S);              ◂ On page 198.
        end
    end
    N := P;                                                        ◂ Go up a level.
    P := parent(N);                                                ◂ P may be null.
end
```

Proposition 8.7. Each cluster produced by **SDC** is a connected set.

Proof. If there are two primitive boxes in the subtree of a cluster that form a disconnected set, then there must be a separator at their common ancestor. This contradicts the property of the **SDC** algorithm: no separator exists in output subtrees. ∎

8.3.5. Combinations of Algorithms

Three proposed algorithms produce the same results (see Figure 8.7a) for orthogonal-separable bounding-box trees. However, they produce different results if bounding-box trees are not orthogonal-separable (see Figure 8.5, Figure 8.6, Figure 8.7b and Figure 8.8). The **Colonization** algorithm allows getting the best max-connected clustering, but its time complexity is quadratic. The **MCC** algorithm only allows getting a hull-disjoint max-connected clustering; however, it terminates in very short time in our experiments. Both the **Colonization** algorithm and the **MCC** algorithm guarantee the max-connectedness of produced clusters. Conversely, the **SDC** algorithm does not guarantee the max-connectedness, although it seems to be the most adaptive clustering technique among the three and terminates in very short time. Therefore, we need to investigate their combinations.

8.3.5.1. Combination of MCC and Colonization

As mentioned in Section 8.3.3, the **MCC** algorithm provides a max-connected clustering such that the clusters are pairwise hull-disjoint. Hence, if we apply the **Colonization** algorithm to every fitted tree produced by the **MCC** algorithm, we will get the optimal max-connected

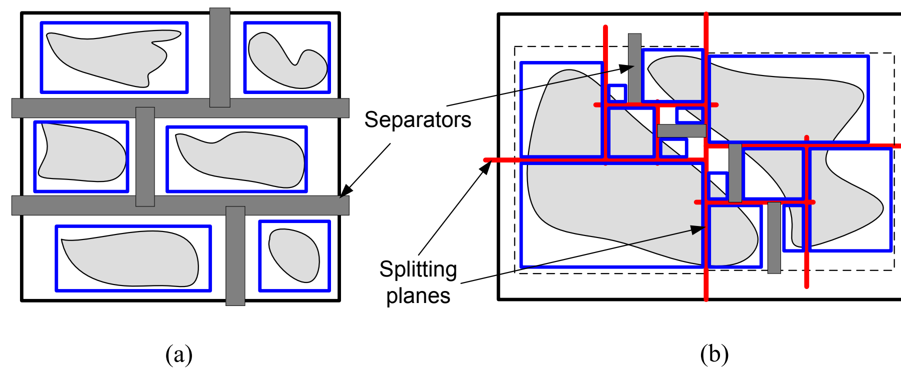

Figure 8.7. The bounding-box tree is (a) orthogonal-separable: all algorithms produce the same result; (b) not orthogonal-separable: **Colonization** produces 2 clusters (or 1 box if perform the optional phase), **MCC** produces 1 box, **SDC** produces 13 boxes.

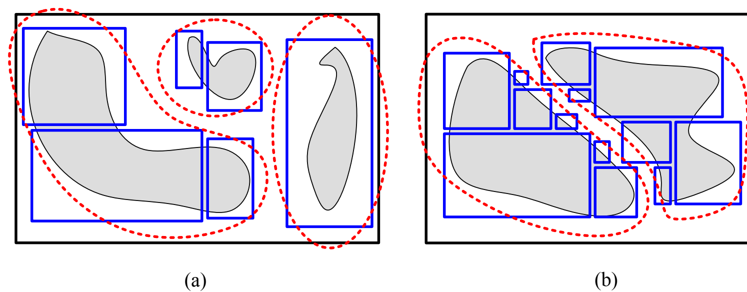

Figure 8.8. (a) Applying **OSDC** to the problem in Figure 8.6 to get 3 clusters; (b) Applying **OSDC** to the problem in Figure 8.7b to get 2 clusters.

clustering. The obtained algorithm is therefore called the *optimal max-connected clustering* (**OMCC**) algorithm.

Let N_i ($1 \le i \le p'$) be the number of primitive boxes in the i-th fitted tree produced by the **MCC** algorithm, where p' ($\le p$) is the number of subtrees produced by **MCC**. The running time of the **Colonization** algorithm on each tree is $\mathcal{O}(dN_i^2)$, where $1 \le i \le p'$. Hence, the total running time of **OMCC** is

$$\mathsf{time}(\mathbf{OMCC}) = \mathsf{time}(\mathbf{MCC}) + \mathcal{O}(d\sum_i N_i^2) = \mathcal{O}(mdN + mQ + d\sum_i N_i^2). \tag{8.1}$$

Noting that $\sum_i N_i = N$, we have $\sum_i N_i^2 \le N^2$; thus, $\mathcal{O}(mdN+mQ+d\sum_i N_i^2)$ does not exceed $\mathcal{O}(dN^2)$. In practice, we often see that Q is much smaller than $\mathcal{O}(dN^2)$ and $\sum_i N_i^2 \ll N^2$. Therefore, the actual running time of the **OMCC** algorithm is often better than that of the **Colonization** algorithm for problems with highly disconnected solution sets; that is, p' is big.

8.3.5.2. Combination of SDC and Colonization

When p' is small, the **OMCC** algorithm will not be efficient. Instead of running the **Colonization** algorithm after the **MCC** algorithm, we continue the **MCC** algorithm until the end of the **SDC** algorithm to get n pairwise hull-disjoint clusters. Each of these clusters is a connected set (by Proposition 8.7). Therefore, if we apply the **Colonization** algorithm to the hull boxes of these clusters, we will get a *nearly* optimal max-connected clustering. The obtained algorithm is

then called the *optimized separator-driven clustering* (**OSDC**) algorithm. The **OSDC** algorithm combines the adaptiveness of the **SDC** algorithm and the optimal max-connectedness of the **Colonization** algorithm.

The complexity of the second phase of the **OSDC** algorithm (i.e., the call to the **Colonization** algorithm) is $\mathcal{O}(dn^2)$. The total running time of the **OSDC** algorithm is

$$\mathsf{time}(\mathbf{OSDC}) = \mathsf{time}(\mathbf{SDC}) + \mathcal{O}(dn^2) = \mathcal{O}(mdN + mQ + dn^2). \tag{8.2}$$

Note that $p' \leq p \leq n$. In practice, n is bounded for fixed numerical CSPs, and if p' is very small then n is small. Therefore, the running time of the **OSDC** algorithm is close to the running time of the **SDC** algorithm.

8.4. Experiments

We now present an evaluation on 16 nonlinear problems that have at most four dimensions and that are selected to reflect different topologies of solution set. The selected problems are categorized into three groups: (*i*) problems with only one connected subset; (*ii*) problems with several connected subsets; and (*iii*) problems with tens to hundreds connected subsets. Our experiments show the similarity in the running time of each algorithm in each group. Therefore, we only need to give the average running time and the average number of computed clusters in individual groups on the left and right of the cells, respectively, in Table 8.1. Figure 8.9 shows the graph of the average running times.

Table 8.1. The average running times in milliseconds (on the left of cells) and the average number of clusters (on the right of cells) in three groups.

$N \rightarrow$	100	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
(*i*) Solver	41	205	409	619	823	1044	1285	1519	1701	1952	2245
(*i*) **Colon.**	2|1.0	48|1.0	195|1.0	430|1.0	779|1.0	1251|1.0	1937|1.0	2762|1.0	3985|1.0	5328|1.0	6734|1.0
(*i*) **MCC**	1|1.0	2|1.0	5|1.0	8|1.0	10|1.0	14|1.0	16|1.0	19|1.0	21|1.0	26|1.0	29|1.0
(*i*) **SDC**	1|3.3	2|3.3	6|3.3	9|3.3	12|3.3	15|3.3	17|3.3	20|3.3	23|3.3	28|3.3	32|3.3
(*ii*) Solver	69	285	525	774	1024	1263	1538	1788	2002	2288	2601
(*ii*) **Colon.**	2|3.1	25|3.6	78|3.7	172|3.9	290|4.0	447|4.1	636|4.1	863|4.4	1120|4.4	1453|4.4	1850|4.4
(*ii*) **MCC**	0|3.1	2|3.6	4|3.7	6|3.9	8|4.0	10|4.1	12|4.1	15|4.4	16|4.4	19|4.4	21|4.4
(*ii*) **SDC**	0|5.3	2|11.8	4|11.3	7|11.2	9|10.4	11|10.3	13|11.0	15|9.7	17|9.0	19|9.0	22|9.0
(*iii*) Solver	66	415	860	1253	1683	2112	2557	2935	3371	3883	4271
(*iii*) **Colon.**	1|9.0	19|24.5	46|31.5	83|47.3	148|69.8	229|97.3	310|116.8	420|116.8	579|116.8	740|116.8	949|116.8
(*iii*) **MCC**	1|9.0	6|24.5	12|31.5	18|47.3	24|69.8	31|97.3	38|116.8	41|116.8	42|116.8	46|116.8	49|116.8
(*iii*) **SDC**	1|26.8	7|79.5	12|109.5	18|118.5	24|131.3	32|136.5	39|116.8	41|116.8	43|116.8	46|116.8	50|116.8

The results show that the running time of the **Colonization** algorithm is quadratic in N (the number of boxes) while the running times of the **MCC** and **SDC** algorithms are very short (the average time of clustering 5000 boxes is less than 50 ms, and the running time is always less than 120 ms). The running time of the solver that uses the search algorithm in [Vu *et al.* 2003] is linear in N. For all tested problems, the running times of **MCC** and **SDC** are much less than that of the **Colonization** algorithm and, in our experiments, are close to zero.

For all tested problems, the **Colonization** and **MCC** algorithms provide the same clusterings. For most tested problems, the three algorithms produce the same clusterings, though the **SDC** algorithm is better than the others for a few problems. Namely, they only show significant

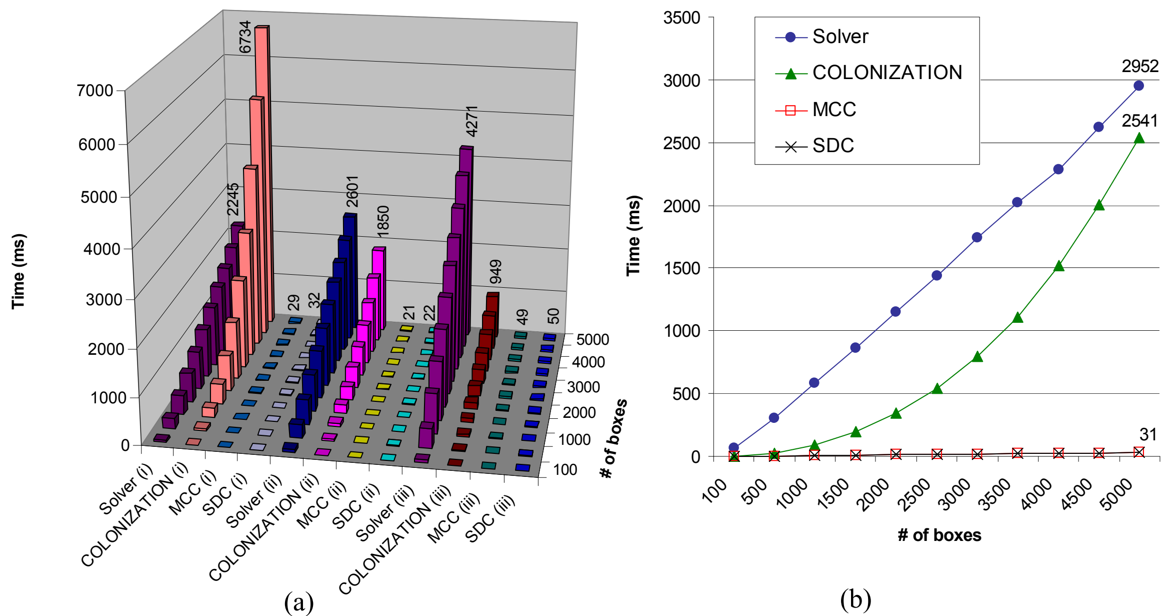

Figure 8.9. The average running times in milliseconds (a) in three groups; (b) for all problems

differences for the following problems:

$$\mathbf{F2.4} \equiv \langle \sin(x \sin y) \geq \cos(y \cos x);\ -4 \leq x, y \leq 4\rangle;$$

$$\mathbf{G1.2} \equiv \langle x_1^2 + 0.5x_2 + 2(x_3 - 3) \geq 0,\ x_1^2 + x_2^2 + x_3^2 \leq 25,\ \forall i : -8 \leq x_i \leq 8\rangle;$$

$$\mathbf{H1.1} \equiv \begin{cases} x_1^2 + x_2^2 + x_3^2 \leq 9, \\ (x_1 - 0.5)^2 + (x_2 - 1)^2 + x_3^2 \geq 4, \\ x_1^2 + (x_2 - 0.2)^2 \geq x_3, \\ -4 \leq x_i \leq 4 \quad (i = 1, \dots, 4). \end{cases}$$

For example, when $N = 1000$, the **Colonization** and **MCC** algorithms provide six boxes for the problem **F2.4**, the ratio of the volume of these six boxes to the volume of the hull of primitive boxes is 0.761 while the ratio obtained by **MCC** is 0.385 with 30 boxes. When $N = 1000$ for problem **G1.2** (and **H1.1** respectively), the **Colonization** and **MCC** algorithm result in no reduction while the **SDC** algorithm produces four (respectively, five) boxes with the volume ratio is 0.225 (respectively, 0.191). Our experiments show that, even in case the **Colonization** and **MCC** algorithms cannot characterize the solution set well (e.g., the problems **G1.2** and **H1.1**), the **SDC** algorithm is still be able to provide an adaptive clustering that are more suitable for applications.

We have only carried out experiments with three fundamental algorithms (**Colonization**, **MCC** and **SDC**); however, the performance of the **OMCC** algorithm (respectively, the **OSDC** algorithm) could be estimated from the performance of the **MCC** algorithm (respectively, the **SDC** algorithm) and the performance of **Colonization** for a small number of boxes (less than 120 boxes in our experiments, then takes less than 2 to 3 ms) by using the formula (8.1) (respectively, the formula (8.2)).

8.5. Conclusion

We propose three post-processing algorithms (namely, **Colonization**, **SDC** and **SDC**) and their combinations, which address different goals, to regroup the verbose output of an interval constraint solver into a much smaller number of boxes. In our experiments, the **SDC** algorithm shows to be the most adaptive among the above-listed three techniques. Moreover, the **MCC** algorithm and the **SDC** algorithm are very cheap (in running time) *post-processing* techniques to get useful grouping information on the set of boxes provided by interval constraint solvers. We also can deduce that, for applications that require the optimal (respectively, a nearly optimal) max-connected clustering, the **OMCC** algorithm (respectively, the **OSDC** algorithm) could be an alternative to the **Colonization** algorithm, in case the number of clusters is big (respectively, small). Potentially, applying these post-processing techniques to the output of interval constraint solvers makes it possible to use the solvers in various applications (as mentioned in Section 8.1) that require concise representations of the solution set.

Chapter 9

Conclusions

In Section 9.1, we summarize our contributions proposed in the previous chapters. In Section 9.2, we give some limitations and challenges of constraint programming methods, including our methods. In Section 9.3, we summarize the potential directions that are open for further research. In Section 9.4, we give the final conclusion of the thesis.

9.1. Contributions

The detailed contributions of the thesis are presented at the end of each previous chapter (cf. the chapters 4, 5, 6, 7, and 8). They are summarized as follows:

C1 In Chapter 4, we propose **a new complete search technique, called UCA6$^+$, to solve numerical constraint satisfaction problems**. The proposed search technique is general and applicable to most branch-and-prune based solution methods. In general, it improves search performance of the most recent generic search technique in case the solution set contains continuums of solutions, often by an order of magnitude, while maintaining the same performance as the latest search technique in case solutions are isolated. Moreover, it provides a concise representation of solutions.

C2 In Chapter 5, we propose **several improvements to, and an abstraction of, inclusion techniques**. These proposals are used in our new constraint propagation techniques in Chapter 6 and Chapter 7. Among our proposals are the followings:

(a) We revise the concept of an interval form for extended functions. The revision of the division in interval arithmetic allows obtaining tighter enclosures at each stage in constraint propagation.

(b) We revise affine forms such that the number of noise variables will not increase during computations; hence, it is potentially useful for long-running computations.

(c) We point out that Kolev generalized affine arithmetic on revised affine forms is two times faster than that on Kolev affine forms (namely, it reduces the number of real operations needed for the multiplication, the most common but expensive operation, from $4n^2 + 8n + 10$ to $2n^2 + 6n + 10$; where n is the number of noise variables).

(d) We propose a new multiplication for variants of affine arithmetic. It reduces the complexity of the tight multiplication from $\mathcal{O}(n^2)$ to $7n + 12$ and provides tighter enclosures than the latest multiplication with the same number of real operations.

(e) We propose a generic procedure to compute Chebyshev affine approximations for continuously differentiable functions in a rigorous manner.

(f) We propose an abstract inclusion concept, called the *real inclusion representation*. This facilitates presenting our novel generic scheme for combining multiple inclusion techniques in constraint propagation (in Chapter 7).

C3 In Chapter 6, we propose **a new numerical constraint propagation technique, called FBPD, and a method for coordinating FBPD and search on directed acyclic graphs**. This makes the fundamental framework of interval analysis on *directed acyclic graphs* (DAGs) (proposed by Schichl and Neumaier [2004b]) efficient and practical for numerical constraint satisfaction. Our experiments show that the new propagation technique outperforms previously available constraint propagation techniques by 1 to 2 orders of magnitude or more in speed, while being roughly the same quality w.r.t. enclosure properties. The experiments also show that the advance gained for under-constrained problems is better than that for well-constrained problems.

C4 In Chapter 7, our contribution is twofold:

(a) We propose **a novel generic scheme, called CIRD, to combine multiple inclusion techniques in numerical constraint propagation**. The scheme potentially allows bringing into the constraint propagation framework the strengths of different techniques coming from different fields. It uses real inclusion representations (proposed in Chapter 5) on (partial) DAG representations (proposed by Schichl and Neumaier [2004b]). This enables devising fine-grained and flexible combination strategies for performing constraint propagation on virtually any factorable constraint system, including numerical constraint satisfaction problems. We also draw out plenty of potential directions to integrate inclusion tools into the **CIRD** scheme.

(b) We devise from the generic scheme **several specific combination strategies for numerical constraint propagation**. These strategies are designed to combine interval constraint propagation, interval arithmetic, affine arithmetic, and linear programming into the framework of constraint propagation. Our experiments on a specific combination strategy, **CIRD[ai]**, show that the new approach outperforms previously available constraint propagation techniques by 1 to 4 orders of magnitude or more in speed, while still being better in quality. It even outperforms some very recent techniques that are specially designed to solve special constraint systems. The largest acceleration is gained for well-constrained problems. It shows that the fine-grained structure made available on (partial) DAG representations through the abstract inclusion concept is a very important and original contribution.

C5 In Chapter 8, we propose **several post-processing techniques for the representation of continuums of solutions**: **MCC**, **SDC**, **OMCC**, **OSDC**. Based on connectedness, they allow grouping each cluster of connected solution subsets into a larger subset. Hence, these techniques are potentially useful in applications such as collision detection and interactive graphics because the grouping information is useful for soon guaranteeing that systems have no collisions. In such applications, the proposed techniques bridge the gap between the verboseness of the output of interval constraint solvers and the compactness required by the applications. Hence, they enable interval constraint solvers to be alternatives to bounding-volume techniques used in such applications.

9.2. Limitations and Challenges

There are several limitations of, and challenges to, constraint programming methods, especially the complete methods such as our proposed methods:

- The *explicit representation* of (non-isolated) solution sets are highly intractable, although it is certainly desired in many applications. They can only obtained for low dimensional problems (say, a few dimensions) even for simple and convex sets such as hyper-spheres. So far, there are only very few research results on this issue. Therefore, there is a need for progress on the semi-explicit representation of solution sets, such as 2^k-tree for *ternarized NCSPs*. However, the *semi-explicit representation* usually requires much efforts to be spent on implementation.

- The performance of numerical constraint programming methods is almost measured by experiments. There is almost no algorithm coming out with a good formula to estimate the overall performance. There is also no algorithm for predicting the speed.

- The complete methods of today can solve only problems of medium sizes, in general. It is a challenge to boost, in the near future, the complete methods for solving a *vast number* of practical problem instances of 20 to 50 variables or more.

- The constraint programming methods of today are often more complicated than mathematical methods in implementation. Moreover, many research results are published without a free demonstration code. This would result in constraint programming methods being less and less used in industry.

9.3. Further Research

Basing on our results, we draw several potential directions for further research:

1. **Search Techniques.** The results in Chapter 4 are encouraging enough to investigate further in the following directions:

   (a) Other combinations of the control parameters and higher values of D_{stop} in combination with solvers with good alignments such as 2^k-tree solvers [Lottaz 2000; Sam-Haroud 1995; Sam-Haroud and Faltings 1996], the **Exclusion** algorithm (Algorithm 3.1, page 69), and the **Inclusion** algorithm (Algorithm 3.2, page 77) in the place of the secondary solution technique (**DimStopSolver**, page 123).

   (b) The idea of complementary boxing (Chapter 4) can be combined with the idea of extreme points in [Faltings 1994; Faltings and Gelle 1997] to make a new algorithm that works on ternarized NCSPs. Hence, the *semi-explicit representation* of solution sets can be obtained at low time and space complexity for large class of problems. The idea of stopping reducing tiny domains for better alignments allows integrating domain reduction operators into the 2^k-tree solvers. Namely, a reduced box can be enlarged to the smallest box with bounds of the binary expansion $\sum_{i=1}^{n} b_i 2^{-i}$. There is probably a close relation between these techniques and the wavelet theory. Therefore, these two fields should be useful to each other.

2. **Forward-Backward Propagation.** The nature of the **FBPD** algorithm (Chapter 6) is similar to that of the **HC4** algorithm [Benhamou *et al.* 1999]. Hence, we can use the **FBPD** algorithm in many applications and combination techniques that use the **HC4** algorithm. Moreover, this technique might be useful for solving optimization problems because it appears to be more suitable for under-constrained problems than well-constrained ones. It seems to be easy to integrate the idea of **FBPD** into other interval analytic methods in the framework proposed by Schichl and Neumaier [2004b].

3. **Combination of Inclusion Techniques.** The **CIRD** scheme (Chapter 7) opens several potential directions for further research:

   (a) The strengths of the **FBPD** algorithm and the **CIRD[ai]** algorithm are complementary. Therefore, unifying the strengths of **FBPD** and **CIRD[ai]** to solve problems with either isolated or non-isolated solutions is a straightforward direction.

   (b) Replace the *linear programming* (LP) technique in the affine pruning (in **CIRD[ai]**) with domain reduction techniques for linear systems, such as the techniques listed in Section 7.5, because LP is very expensive for this purpose.

   (c) Implement the new divisions for affine forms proposed by [Kolev 2002] and [Miyajima *et al.* 2003] in place of the division $x/y := x * (1/y)$ implemented in **CIRD[ai]**.

   (d) Integrate Kolev generalized affine arithmetic (with the improvements of the multiplication presented in Chapter 5) into the **CIRD** scheme. (See also Section 2.2.3.4.)

   (e) Integrate linear relaxation techniques (e.g., the ones in [Hongthong and Kearfott 2004] and [Borradaile and Van Hentenryck 2004]) into the **CIRD** scheme.

   (f) Integrate the *quadratic form* proposed by [Messine 2002] and **Quad** proposed by [Lebbah *et al.* 2003a,b] (see Section 3.3.1.1) into the **CIRD** scheme.

   (g) Investigate intelligent choosing strategies of the **CIRD** scheme, which choose the next node in waiting lists based on the pruning efficiency of nodes. Behavior learning of node range reductions is also an interesting direction.

   (h) Investigate the ability to integrate high-order inclusion techniques, such as the convexification techniques as mentioned in Section 7.5, into the **CIRD** scheme.

4. **Clustering Techniques.** Because interval constraint solvers are very similar to successful techniques, called the *bounding-volume techniques* (see Section 8.1), arising in collision detection, we can think about using interval constraint solvers in place of the bounding-volume techniques. The clustering techniques proposed in Chapter 8 are potentially useful in this context.

9.4. The Final Conclusion

Because our observations on the limitations of existing methods are reasonable, the goal set for the thesis (see Section 1.2) has been achieved successfully. Namely, we achieve the results summarized in Section 9.1, which address four major issues in solving NCSPs: search, constraint propagation, combination of techniques, post-processing. Moreover, the results also open various potential directions for further research, as described in Section 9.3.

Appendix A

Extended Concepts of Interval Arithmetic

A.1. A Short History of Interval Arithmetic

The first idea of using intervals for computations can be traced back to Archimedes (Syracuse, Greece, 287–212 B.C.),[1] the famous physicist and mathematician who found two-sided bounds for π: $3\frac{10}{71} < \pi < 3\frac{10}{70}$, and a method for improving them successively (see [Archimedes 1953]). More than 2000 years later, W. H. Young introduced the concept of a function having values that are bounded within limits (see [Young 1908]). Tens years later, the American mathematician and physicist, N. Wiener, brought the computation of two fundamental physical quantities, namely the position and the time, into the concept of an interval (see [Wiener 1914, 1921]). Afterwards, J. C. Burkill introduced the concept of functions of intervals in [Burkill 1924]. Next seven years, R. C. Young introduced the concept of operations with a set of multi-valued numbers (see [Young 1931]). Twenty years later, P. S. Dwyer further developed a special case of closed intervals (see [Dwyer 1951]). Few years later, a basic calculus on intervals with care for rounding errors was developed by M. Warmus in [Warmus 1956] (see also [Warmus 1961]). It is believed that modern interval arithmetic was developed independently in late 1950s by several researchers, including M. Warmus [1956], T. Sunaga [1958] and R. E. Moore [1959]. Later on, R. E. Moore enriched the research in this direction with his PhD thesis [Moore 1962]. While almost nobody was willing to make any progress in this direction, R. E. Moore kept his end up and wrote the first foundational book on interval analysis [Moore 1966], and then many further publications. Until one discovered that the traditional mathematical methods sometimes produce drastically erroneous results because of the presence of rounding errors in computers, they started considering interval arithmetic as an alternative. Owning to his excellent accomplishments and relentless efforts, R. E. Moore is regarded as a founding father of interval arithmetic and interval analysis. The interest in interval arithmetic has been growing. Interval arithmetic has been successfully used in numerous computing methods, not only interval analytic ones, to solve real world applications.

[1] According to the report at `http://www.cs.utep.edu/interval-comp/early.html`.

A.2. Typical Interval Functions

A.2.1. Natural Interval Form

Definition A.1 (Natural Interval Form). Let $f : \mathbb{R}^n \to \mathbb{R}$ be a factorable function using the operators $+$, $-$, $*$, $\div$ and elementary functions such as sin, cos, exp, and sqr. The interval form $\mathbf{f} : \mathbb{I}^n \to \mathbb{I}$ for f that is obtained by replacing each real variable x_i by an interval variable $\mathbf{x}_i$ and each operator or elementary function by its interval counterpart is called the *natural interval form*.

It has been proved that $f(x) \in \mathbf{f}(\mathbf{x})$ for all $x \in \mathbf{x}$. That is, the interval function defined in Definition A.1 is an interval form of its counterpart.

A.2.2. Centered Interval Form

The centered interval form is derived from the mean value theorem, which is defined as follows (see [Jaulin *et al.* 2001, p. 33]).

Definition A.2 (Centered Interval Form). Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function, $\mathbf{x} \in \mathbb{I}^n$, $c = \mathrm{mid}(\mathbf{x})$, g the *gradient* (or the derivative) of f, $[g]$ an interval form of g. The following function is called the *centered interval form* of f in $\mathbf{x}$:

$$[f]_{\mathrm{C}}(\mathbf{x}) \equiv f(c) + (\mathbf{x} - c)^{\mathrm{T}}[g](\mathbf{x}). \tag{A.1}$$

It follows from the mean value theorem that $f(x) \in [f]_{\mathrm{C}}(\mathbf{x})$ for all $x \in \mathbf{x}$. That is, the interval function defined by (A.1) is an interval form of its counterpart.

A.2.3. Mixed Centered Interval Form

The following interval function is due to Hansen [1968]. Hansen [1968] proved that it is an interval form (see also [Jaulin *et al.* 2001, p. 34]).

Definition A.3 (Mixed Centered Interval Form). Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function, $\mathbf{x} \in \mathbb{I}^n$, $c = \mathrm{mid}(\mathbf{x})$, g the gradient (or the derivative) of f, $[g]$ an interval form of g. Denote $c = (c_1, \ldots, c_n)^{\mathrm{T}}$ and $[g] = ([g_1], \ldots, [g_n])^{\mathrm{T}}$. The *mixed centered interval form* is defined as

$$f(c) + \sum_{i=1}^{n} (\mathbf{x}_i - c_i)^{\mathrm{T}}[g_i](\mathbf{x}_1, \ldots, \mathbf{x}_i, c_{i+1}, \ldots, c_n). \tag{A.2}$$

Recall that the partial derivative $g_i = \partial f / \partial x_i$. The mixed centered interval form can be viewed as the applied the rule (A.1) for constructing the centered interval form n times sequentially, each for one variable. In particular, the proof that the mixed centered interval is

indeed an interval form of its real-valued counterpart is based on the sequence of applications of the mean valued theorem:

$$f(x_1,\ldots,x_i,c_{i+1},\ldots,c_n) \in f(x_1,\ldots,x_{i-1},c_i,\ldots,c_n)+(\mathbf{x}_i-c_i)^{\mathrm{T}} g_i(x_1,\ldots,x_{n-1},\mathbf{x}_i,c_{i+1},\ldots,c_n)$$

for all $i = 1,\ldots,n$. Composing all these inclusions, we have the result as required.

A.2.4. Taylor Interval Form

From the Taylor expansion of real-valued functions, we can construct an interval form as follows (see [Jaulin *et al.* 2001, p. 35]).

Definition A.4 (Taylor Interval Form). Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function, $\mathbf{x} \in \mathbb{I}^n$, $c = \mathrm{mid}(\mathbf{x})$, g the gradient (or the derivative) of f, $[g]$ an interval form of g, $[H]$ an interval form of the *Hessian matrix*. The following is called the *Taylor interval form*:

$$[f]_{\mathrm{T}}(\mathbf{x}) \equiv f(c) + (\mathbf{x}-c)^{\mathrm{T}} g(c) + \frac{1}{2}(\mathbf{x}-c)^{\mathrm{T}}[H](\mathbf{x})(\mathbf{x}-c). \tag{A.3}$$

Recall that the component $[H_{ij}]$ of the matrix $[H]$ is an interval form of

$$h_{ij} \equiv \begin{cases} \partial^2 f/\partial x_i^2 & \text{if } i=j \quad (\text{for } i=1,\ldots,n); \\ 2\partial^2 f/\partial x_i x_j & \text{if } i>j \quad (\text{for } i=1,\ldots,n); \\ 0 & \text{otherwise.} \end{cases} \tag{A.4}$$

A interval form of the symmetric *Hessian matrix* ($h_{ij} = \partial^2 f/\partial x_i x_j$ for all i and j) could also be used, but the resulting would increase in the number of interval components of $[H](\mathbf{x})$, hence this would lead to an overestimation of $[f]_{\mathrm{T}}$.

A.2.5. Linear Interval Mapping

A linear/sublinear interval mapping is defined as follows [Neumaier 1990].

Definition A.5 (Linear, Sublinear Mapping). A mapping $\phi : \mathbb{I}^n \to \mathbb{I}^m$ is said to be *sublinear* if the axioms

$$\text{(inclusion isotonicity)} \qquad \mathbf{x} \subseteq \mathbf{y} \Rightarrow \phi(\mathbf{x}) \subseteq \phi(\mathbf{y}), \tag{A.5}$$

$$\text{(homogeneity)} \qquad \alpha \in \mathbb{R} \Rightarrow \phi(\alpha\mathbf{x}) = \alpha\phi(\mathbf{x}), \tag{A.6}$$

$$\text{(subadditivity)} \qquad \phi(\mathbf{x}+\mathbf{y}) \subseteq \phi(\mathbf{x}) + \phi(\mathbf{y}) \tag{A.7}$$

hold for all $\mathbf{x},\mathbf{y} \in \mathbb{I}^n$; and called *linear* if the axioms (A.5), (A.6) and

$$\text{(additivity)} \qquad \phi(\mathbf{x}+\mathbf{y}) = \phi(\mathbf{x}) + \phi(\mathbf{y}) \tag{A.8}$$

hold for all $\mathbf{x},\mathbf{y} \in \mathbb{I}^n$. A sublinear mapping ϕ is said to be *regular* if

$$x \in \mathbb{R}^n,\ 0 \in \phi(x) \Rightarrow x = 0.$$

A.3. Advanced Concepts on Intervals

A.3.1. Interval Matrix

We recall the concept and notation of the spectral radius of a matrix.

Definition A.6 (Spectral Radius). The maximum absolute value of the eigenvalues of a matrix $A \in \mathbb{R}^{n\times n}$, denoted by

$$\rho(A) \equiv \max\{|\lambda| \mid \lambda \in \mathbb{C}, \exists x \in \mathbb{C} \setminus \{0\} : Ax = \lambda x\},$$

is called the *spectral radius* of A.

The spectral radius $\rho(A)$ is a continuous function of A because the eigenvalues of A continuously depend on A. See some properties of the spectral radius in [Neumaier 1990, p. 86–88]. Hereafter, we extend the concept of a matrix inverse to include the non-square matrices.

Definition A.7 (Extended Matrix Inverse). Let $A \in \mathbb{R}^{m\times n}$ be a *real matrix* of rank n, then the equation $Ax = b$ has at most one solution x^*, and $A^{\mathrm{T}}A$ is a *nonsingular* real matrix of size $n \times n$. The *(extended) inverse* of A is defined as

$$A^{-1} \equiv (A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}} \in \mathbb{R}^{n\times m}. \tag{A.9}$$

In Definition A.7, we also have $A^{-1}A = (A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}A = I \in \mathbb{R}^{n\times n}$, and the solution x^* can be written as $x^* = A^{-1}b$. Hence, this definition is compatible with the definition of the inverse of square matrices, with restriction to these two properties.

Definition A.8 (Interval Matrix). An *interval matrix* is a matrix of intervals.

The next definition of comparison matrix is from [Neumaier 1990, p. 110].

Definition A.9 (Comparison Matrix). The *comparison matrix* of an interval matrix $\mathbf{A} \equiv (\mathbf{A}_{ij})_{n\times n}$ is a matrix $\langle\mathbf{A}\rangle \equiv (\langle\mathbf{A}\rangle_{ij})_{n\times n}$, where

$$\langle\mathbf{A}\rangle_{ij} \equiv \begin{cases} \inf\{|x| \mid x \in \mathbf{A}_{ij}\} & \text{if } i = j; \\ -|\mathbf{A}_{ij}| & \text{if } i \neq j. \end{cases} \tag{A.10}$$

The concepts of an M-matrix and an H-matrix are defined in the next definitions (from [Neumaier 1990, p. 105, 111]). Their new properties can be found in [Spiteri 2003].

Definition A.10 (M-Matrix). An *M-matrix* is a square matrix $\mathbf{A} \in \mathbb{I}^{n\times n}$ such that

$$\exists u \in \mathbb{R}^n : \mathbf{A}u > 0. \tag{A.11}$$

Definition A.11 (H-Matrix). An *H-matrix* is a square matrix $\mathbf{A} \in \mathbb{I}^{n\times n}$ such that

$$\exists u \in \mathbb{R}^n : \langle \mathbf{A} \rangle u > 0. \tag{A.12}$$

A.3.2. Interval Matrix Inverse

Definition A.12 (Linear Interval Equation). Let $\mathbf{A} \in \mathbb{I}^{m\times n}$, $\mathbf{b} \in \mathbb{I}^m$, and $\mathbf{x} \in \mathbb{I}^n$. The family of linear equations

$$Ax = b, A \in \mathbf{A}, b \in \mathbf{b}, x \in \mathbf{x} \tag{A.13}$$

is called a *linear interval equation*, where x is a vector of the real variables.

Definition A.13 (Linear Interval Inequalities). Let $\mathbf{A} \in \mathbb{I}^{m\times n}$, $\mathbf{b} \in \mathbb{I}^m$, and $\mathbf{x} \in \mathbb{I}^n$. The family of linear inequalities

$$Ax \le b \;\; (A \in \mathbf{A}, b \in \mathbf{b}) \tag{A.14}$$

is called a *linear interval inequality*, where x is a vector of the real variables.

Definition A.14 (Hull Inverse). Let $\mathbf{A}$ be a set of real matrices of size $m \times n$ and $\mathbf{b} \in 2^{\mathbb{R}^m}$ a vector of size m. The *solution set* of the family of linear equations

$$Ax = b \;\; (A \in \mathbf{A}, b \in \mathbf{b}) \tag{A.15}$$

is the set $\Sigma(\mathbf{A}, \mathbf{b}) \equiv \{x \in \mathbb{R}^n \mid Ax = b, A \in \mathbf{A}, b \in \mathbf{b}\}$. If $\mathbf{A}$ is an interval matrix and $\mathbf{b}$ is an interval vector, the hull $\square\Sigma(\mathbf{A}, \mathbf{b})$ is called the *hull inverse* of $\mathbf{A}$ w.r.t. $\mathbf{b}$.

The concept of a fixed point inverse of an interval matrix is based on fixed point iterations. Namely, it is defined as follows [Neumaier 1990, p. 145].

Definition A.15 (Fixed Point Inverse). Let $\mathbf{A} = (\mathbf{A}_{ij})_{n\times n} \in \mathbb{I}^{n\times n}$ be an H-matrix. The *fixed point inverse* of $\mathbf{A}$ is the unique mapping, denoted by $\mathbf{A}^{\mathrm{F}} : \mathbb{I}^n \to \mathbb{I}^n$, which maps each $\mathbf{b} = (\mathbf{b}_1, \dots, \mathbf{b}_n)^{\mathrm{T}} \in \mathbb{I}^n$ to the unique solution, denoted by $\mathbf{A}^{\mathrm{F}}\mathbf{b}$, $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_n)^{\mathrm{T}} \in \mathbb{I}^n$ of the system of equations

$$\mathbf{x}_i = (\mathbf{b}_i - \sum_{j\ne i} \mathbf{A}_{ij}\mathbf{x}_j)/\mathbf{A}_{ii} \quad (\text{for } i = 1, \dots, n). \tag{A.16}$$

It has been proved that $\mathbf{A}^{\mathrm{F}}$ is sublinear, and $\square\Sigma(\mathbf{A}, \mathbf{b}) \subseteq \mathbf{A}^{\mathrm{F}}\mathbf{b}$ (see Theorem 4.4.4 in [Neumaier 1990, p. 145]).

Next, we give three slight extensions of the concepts of a regular interval set and related concepts (see [Neumaier 1990, p. 107]).

Definition A.16 (Regular Matrix Set). A set $\mathbf{A}$ of real matrices of size $m \times n$ is said to be *regular* it is closed, convex and bounded, and every real matrix $A \in \mathbf{A}$ has rank n.

Definition A.17 (Strongly Regular Interval Matrix). An interval matrix $\mathbf{A}$ of size $m \times n$ is said to be *strongly regular* if it is regular and the matrix product $\mathrm{mid}(\mathbf{A})^{-1}\mathbf{A}$ is regular.

Definition A.18 (Interval Matrix Inverse). The *matrix inverse* of a regular interval matrix, $\mathbf{A}$, is defined as

$$\mathbf{A}^{-1} \equiv \square\{A^{-1} \mid A \in \mathbf{A}\}. \tag{A.17}$$

Basing on the above extended concepts of a regular matrix set and of the interval matrix inverse, we have the following theorem that extends the result in [Neumaier 1990, p. 174] for the case $m \neq n$.

Theorem A.19. Consider the extended matrix inverse defined in Definition A.7. Let $\mathbf{A}$ be a regular interval matrix of size $m \times n$, $\mathbf{b} \in \mathbb{I}^m$. Then

$$\Sigma(\mathbf{A}, \mathbf{b}) \subseteq \square\Sigma(\mathbf{A}, \mathbf{b}) \subseteq \mathbf{A}^{-1}\mathbf{b}. \tag{A.18}$$

Proof. The first part of (A.18) is obvious. Here we prove the second part. Let $x \in \Sigma(\mathbf{A}, \mathbf{b})$, then there exist $A \in \mathbf{A}$ and $b \in \mathbf{b}$ such that $Ax = b$. Since $\mathbf{A}$ is regular, then the extended inverse A^{-1} is defined and contained in $\mathbf{A}^{-1}$. Therefore, we have

$$A^{-1}b = A^{-1}Ax = (A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}Ax = Ix = x.$$

Hence $x = A^{-1}b \in \mathbf{A}^{-1}\mathbf{b}$ (see [Neumaier 1990, Proposition 3.1.4]). Then (A.18) follows this because the terms are all interval matrix. ∎

For completeness, we recall here the definitions from [Neumaier 1990, p. 174].

Definition A.20 (Lipschitz Set, Lipschitz Matrix). A closed, convex and bounded set $\mathbf{A}$ of real matrices of size $m \times n$ is called a *Lipschitz set* for a function $f : D_0 \subseteq \mathbb{R}^n \to \mathbb{R}^m$ on $D \subseteq D_0$ if for every $x, y \in D$ we have

$$\exists A \in \mathbf{A} : f(x) - f(y) = A(x - y).$$

In addition, if $\mathbf{A} \in \mathbb{I}^{m \times n}$, then we call $\mathbf{A}$ a *Lipschitz matrix* for f on D.

A.3.3. Interval Slope

We recall here the definitions of slopes, which were introduced in [Hansen 1978; Krawczyk and Neumaier 1985] (see also [Neumaier 1990, p. 56]).

Definition A.21 (Slope, Slope Matrix). Let $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ be a function. If for some $x, z \in D$ we have a matrix $f[z, x] \in \mathbb{R}^{m\times n}$ such that

$$f(x) - f(z) = f[z, x](x - z), \tag{A.19}$$

then $f[z, x]$ is called a *(first order) slope matrix*, or *(first order) slope* for short, of f at $[z, x]$. If (A.19) holds for all $x, z \in D$, then it defines a function $f[., .] : D^2 \to \mathbb{R}^{m\times n}$ that is called a *(first order) slope function* of f on D.

Similarly to interval concepts, the slope form and slope matrix are defined as follows.

Definition A.22 (Slope Form). If (A.19) holds for every $x \in \mathbf{x} \in \mathbb{I}^n$ and $f[u, v]$ is an arithmetic expression in u and v, then the following is called the *slope form* with center $z \in \mathbf{x}$

$$s(\mathbf{x}) \equiv f(z) + f[z, \mathbf{x}](\mathbf{x} - z). \tag{A.20}$$

Definition A.23 (Second Order Slope Matrix). Let $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ be a function that has a slope function $f[., .] : D^2 \to \mathbb{R}^{m\times n}$ on D. If for some $x, y, z \in D$ we have a three-dimensional matrix $f[z, y, x] \in \mathbb{R}^{m\times n\times n}$, interpreted as a row vector of $m \times n$ matrices, such that

$$f[z, x] - f[z, y] = f[z, y, x](x - y), \tag{A.21}$$

then $f[z, y, x]$ is called a *second order slope matrix*, or *second order slope* for short, of f at $[z, y, x]$. If (A.21) holds for all $x, y, z \in D$, then it defines a function $f[., ., .] : D^3 \to \mathbb{R}^{m\times n\times n}$ that is called a *second order slope function* of f on D.

Krawczyk and Neumaier [1986] found a bound on the difference of slopes and the ranges of functions (see [Neumaier 1990, Theorem 2.3.3]). Later on, Neumaier [2002, Theorem 8.1] generalized the result as follows.

Theorem A.24 (Neumaier). Let $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ be a function, $\mathbf{x} \in \mathbb{I}^n$ an interval vector contained in D, and $z \in \mathbf{x}$. Let $\mathbf{a} \in \mathbb{I}^m$ and $\mathbf{A} \in \mathbb{I}^{m\times n}$ such that

$$\forall x \in \mathbf{x}, f(x) \in \mathbf{s} \equiv \mathbf{a} + \mathbf{A}(x - z).$$

Then the range $f(\mathbf{x})$ is contained in $\mathbf{s}$ and the *Hausdorff distance* between them satisfies

$$\delta_{\mathrm{H}}(f(\mathbf{x}), \mathbf{s}) \le 2\,\mathrm{rad}(\mathbf{a}) + \mathrm{zl}(\mathbf{A})|\mathbf{x} - z|,$$
$$0 \le \mathrm{rad}(\mathbf{x}) - \mathrm{rad}(f(\mathbf{x})) \le 2\,\mathrm{rad}(\mathbf{a}) + 2\,\mathrm{rad}(\mathbf{A})\,\mathrm{rad}(\mathbf{x}),$$

where $\mathrm{zl}(\mathbf{b}) \equiv \mathrm{w}(\{|b| \mid b \in \mathbf{b}\})$ is the zerolength of $\mathbf{b} \in \mathbb{I}^n$ and it is extended to vectors and matrices in a componentwise manner.

The concept of a Lipschitz set has been generalized to that of a slope set. In particular, it is defined as follows.

Definition A.25 (Slope Set, Interval Slope Matrix). Let $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ be a function. Assume that $\mathbf{x}$ and $\mathbf{y}$ are two interval vectors contained in D. A closed, convex and bounded set $\mathbf{A}$ of real matrices of size $m \times n$ is called a *slope set* at $[\mathbf{x}, \mathbf{y}]$ for f if for every $x \in \mathbf{x}$ and $y \in \mathbf{y}$ we have

$$\exists A \in \mathbf{A} : f(x) - f(y) = A(x - y).$$

In addition, if $\mathbf{A} \in \mathbb{I}^{m \times n}$ then $\mathbf{A}$ is called a *(interval) slope matrix* for f at $[\mathbf{x}, \mathbf{y}]$.

It is easy to see that a slope set/matrix at $[\mathbf{x}, \mathbf{x}]$ is a *Lipschitz set/matrix* on $\mathbf{x}$. However, in practice slope matrices can easily computed by using the inclusion algebra (see Theorem 2.3.8 in [Neumaier 1990]). The resulting slope matrix at $[x_0, \mathbf{x}]$ is always contained in the Lipschitz matrix on $\mathbf{x}$ and has roughly a halved radius (see Proposition 2.3.12 in [Neumaier 1990]).

See [Muñoz and Kearfott 2004] for more on slope intervals, generalized gradients, semigradients, slant derivatives, and csets.

Appendix B

Fixed Point Theory in Metric Spaces

B.1. Basic Concepts on Metric Spaces

We first recall the concept of a metric space (see [Singh *et al.* 1997], [Khamsi and Kirk 2001])

Definition B.1 (Metric Space). A *metric space*, denoted by (S, d), is a set S together with a real-valued function $d : S \times S \to \mathbb{R}$ (called a *metric* or a *distance function*) such that, for every three points x, y, and z in S, the followings hold:

(i) $d(x, y) \geq 0$, with equality if and only if $x = y$;
(ii) $d(x, y) = d(y, x)$;
(iii) $d(x, z) \leq d(x, y) + d(y, z)$.

In the above concept, we often replace the notation (S, d) with S, when not being confused. The *diameter*, $\delta(A)$, of a nonempty subset A of a metric space (S, d) is defined as $\delta(A) = \sup\{d(x, y) \mid x, y \in A\}$.

The next definition is from [Singh *et al.* 1997, Definition 1.6]

Definition B.2 (Precompact Set). A subset C of a metric space S is said to be *precompact* if, for any given $\varepsilon > 0$, there exist a finite number of subsets $C_1, \ldots, C_n$ of S with $\delta(C_i) < \varepsilon$ for all $i = 1, \ldots, n$, and $C \subseteq \bigcup_{i=1}^{n} C_i$.

Given a metric space (S, d), $x \in S$, and $r \in \mathbb{R}$. We recall that the *open ball* centered at x of radius r is the set $B_r(x) = \{y \in S \mid d(x, y) < r\}$. Similarly, the *closed ball* centered at x of radius r is the set $\{y \in S \mid d(x, y) \leq r\}$.

Definition B.3 (Closed/Open Set). Let C be a set in a metric space S. C is said to be *open* if it equals to a union of *open balls* in S. C is said to be *closed* if $S \setminus C$ is open.

Definition B.4 (Cover, Subcover). Consider a set A and a collection of sets $\{A_i\}_{i \in I}$ of a metric space. If $A \subseteq \bigcup A_i$, then $\{A_i\}_{i \in I}$ is called a *cover* of A. If every A_i is an open set, then the cover is called an *open cover*. If there exists a finite collection $A_1, \ldots, A_n$ among the collection $\{A_i\}_{i \in I}$ such that $A \subseteq \bigcup A_i$, then the $\{A_i\}_{i=1}^{n}$ is called a finite *subcover* of A.

The following definition is from [Singh *et al.* 1997, Definition 1.8].

Definition B.5 (Compact Metric Space, Compact Set). Let (S, d) be a metric space. It is said to be *compact* if every open cover of S has a finite subcover. A subset X of S is said to be *compact* if every open cover of X has a finite subcover.

A subset of a finite-dimensional metric space (such as $\mathbb{R}^n$) is compact if and only if it is closed and bounded. A metric space is said to be *complete* if every *Cauchy sequence*[1] converges to a point in that space. For example, the real set $\mathbb{R}^n$ together with the usual distance function is a complete metric space, but the rational set $\mathbb{Q}^n$ is not a complete metric space (because a Cauchy sequence in $\mathbb{Q}^n$ may converge to a real number, not a rational one).

Definition B.6 (Ring, Field). A *ring* $(R, +, \cdot)$ is a nonempty set R together with two binary operations, denoted $+ : R \times R \to R$ and $\cdot : R \times R \to R$, such that

(1) associativity: $(a + b) + c = a + (b + c)$ and $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ for all $a, b, c \in R$;
(2) commutativity: $a + b = b + a$ for all $a, b \in R$;
(3) additive identity: there exists $0 \in R$ such that $a + 0 = a$ for all $a \in R$;
(4) additive invertibility: for all $a \in R$, there exists $b \in R$ such that $a + b = 0$;
(5) distributivity: $(a+b) \cdot c = (a \cdot c) + (b \cdot c)$ and $c \cdot (a+b) = (c \cdot a) + (c \cdot b)$ for all $a, b, c \in R$.

A ring $(R, +, \cdot)$ is said to be *commutative* if the operation $\cdot$ is commutative. A *field* is a commutative ring $(R, +, \cdot)$ with a *multiplicative identity*, 1, such that $1 \neq 0$, and for all $0 \neq x \in R$ there exists $y \in R$ with $x \cdot y = 1$.

Definition B.7 (Vector Space). Let $F = (R, +, \cdot)$ be a field of which the multiplicative identity is 1. A *vector space* V over F is a set V with two binary operations, $+ : V \times V \to V$ and $* : R \times V \to V$, such that

(1) $(x + y) + z = x + (y + z)$ for all $x, y, z \in V$;
(2) $x + y = y + x$ for all $x, y \in V$;
(3) there exists an *additive identity* $\mathbf{0} \in V$ such that $x + \mathbf{0} = x$ for all $x \in V$;
(4) for any $x \in V$, there exists an *additive inverse* $y \in V$ such that $x + y = \mathbf{0}$;
(5) $1 * x = x$ for all $x \in V$;
(6) $a * (b * x) = (a \cdot b) * x$ for all $a, b \in R$ and $x \in V$;
(7) a * (x + y) = (a * x) + (a * y) for all $a \in R$ and $x, y \in V$;
(8) (a + b) * x = (a * x) + (b * y) for all $a, b \in R$ and $x \in V$.

Definition B.8 (Norm, Normed Vector Space). Let V be a vector space over $F = (\mathbb{R}, +, \cdot)$ or $F = (\mathbb{C}, +, \cdot)$ with the notations as in Definition B.7. A *normed vector space* over F is a pair $(V, \|.\|)$, where $\|.\| : V \to \mathbb{R}$ is a function, called a *norm*, such that

(*i*) $\|x\| \geq 0$ for all $x \in V$, and $\|x\| = 0$ if and only if $x = \mathbf{0}$;
(*ii*) $\|\alpha * x\| = |\alpha| * \|x\|$ for all $x \in V$ and $\alpha \in F$;
(*iii*) $\|x + y\| \leq \|x\| + \|y\|$.

[1] A sequence $x_1, x_2, x_3, \ldots$ in a metric space (S, d) is called a *Cauchy sequence* if, for any real number $\varepsilon > 0$, there exists a natural number K such that $d(x_m, x_n) < \varepsilon$ for all $m, n > K$.

A normed space $(V, \|.\|)$ is a metric space under the metric $d : V \times V \to \mathbb{R}$ given by $d(x, y) = \|x - y\|$ (called the *metric induced by the norm* $\|.\|$). A *complete normed vector space* is called a *Banach space*. It is known that the real vector space $\mathbb{R}^n$ with standard real arithmetic and usual ℓ_k-norms is a Banach space, while the space $\mathbb{Q}^n$ of rational numbers is not a Banach space because it is not complete.

Definition B.9 (Convex Set). A set C in a Banach space V is said to be *convex* if $\alpha x + (1 - \alpha)x \in C$ whenever $x, y \in C$ and $0 \leq \alpha \leq 1$.

One often uses the *Hausdorff distance* to measure the degree of mismatch between two sets in a normed vector space.

Definition B.10 (Euclidean Distance, Hausdorff Distance). Given two nonempty sets in a normed vector space. We define

- the Euclidean distance: $\delta_{\mathrm{E}}(X, Y) = \inf\{\|x - y\| \mid x \in X, y \in Y\}$;
- the one-side Hausdorff distance: $d_{\mathrm{H}}(X, Y) = \sup\{\delta_{\mathrm{E}}(\{x\}, Y) \mid x \in X\}$;
- the Hausdorff distance: $\delta_{\mathrm{H}}(X, Y) = \max\{d_{\mathrm{H}}(X, Y), d_{\mathrm{H}}(Y, X)\}$.

For convenience, we denote $\delta_{\mathrm{E}}(\{x\}, T)$ as $\delta_{\mathrm{E}}(x, T)$, and agree that $\delta_{\mathrm{E}}(X, Y) = +\infty$ if X or Y is empty, which agrees also to the standard convention $\inf\{\emptyset\} = +\infty$. We also agree that $\delta_{\mathrm{H}}(X, Y) = d_{\mathrm{H}}(X, Y) = d_{\mathrm{H}}(Y, X) = +\infty$ if X or Y is empty.

B.2. Fundamental Fixed Point Theorems

The following definition is composed from Definition 1.30, Page 15, and Definition 1.84 in [Singh *et al.* 1997].

Definition B.11 (Lipschitz/Nonexpansive/Contractive/Contraction Mapping). Let (S, d) be a metric space. A mapping $f : S \to S$ is called *Lipschitz mapping/function* if there exists a real constant $\alpha \geq 0$ satisfying the *Lipschitz condition*:

$$\forall x, y \in S : d(f(x), f(y)) \leq \alpha\, d(x, y). \tag{B.1}$$

If (B.1) holds, then α is called a *Lipschitz constant*. If (B.1) holds for some $\alpha < 1$, then f is called a *contraction mapping*, or a *contraction* for short. If (B.1) holds for $\alpha = 1$, then f is called a *nonexpansive mapping*. If f satisfies

$$\forall x \neq y \in S : d(f(x), f(y)) < d(x, y), \tag{B.2}$$

then it is called a *contractive mapping*.

Strictly speaking, the *Lipschitz constant* for the function f, defined in Definition B.11, is the greatest lower bound of all α satisfying (B.1), which has been proved to be the smallest among constant α satisfying (B.1). However, people often use the term *Lipschitz constant* to refer to any constant α satisfying (B.1). Therefore, we use the term *smallest Lipschitz constant*

for the greatest lower bound of all α satisfying (B.1) for practical convenience. The following definition is from [Agarwal *et al.* 2001b, Definition 4.6].

Definition B.12 (Compact Mapping). Let X and Y be normed vector spaces. A mapping $f : X \to Y$ is said to be *compact* if $f(X)$ can be contained in a compact subset of Y. In addition to that, f is said to be *finite-dimensional* if $f(X)$ is contained in a *finite-dimensional* vector subspace of Y (i.e., a vector subspace that has a finite basis).

To understand the origination of the fixed point methods presented in Chapter 3, we recall the concept of a fixed point and the most fundamental theorems in the fixed point theory.

Definition B.13 (Fixed Point). A *fixed point* of a function, f, is a point, x, which does not change upon the function f; that is, $f(x) = x$.

One of the most influential theorems in the application of the fixed point theory is Brouwer's fixed point theorem. We recall hereafter an extended form of this theorem, which is due to Schauder (see [Singh *et al.* 1997, Theorem 1.26], see also [Kirk and Sims 2001]).

Theorem B.14 (Brouwer Fixed Point Theorem). Every *continuous mapping* from a *compact convex* subset C of a Banach space to C has at least one fixed point (in C).

The following result follows directly from the Brouwer fixed point theorem.

Theorem B.15. Given a function $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ and a closed bounded box $\mathbf{x} \subseteq D$. If there exists a continuous function $g : D' \supseteq \mathbf{x} \to \mathbb{R}^n$ such that, for all x in $\mathbf{x}$,

$$\begin{aligned} & g(x) = x \Rightarrow f(x) = 0 \\ \text{and} \quad & g(x) \in \mathbf{x} \end{aligned}$$

hold, then there exists a zero of f in $\mathbf{x}$; that is, $f(x) = 0$ for some $x \in \mathbf{x}$.

Proof. Because a box is convex, it follows from the Brouwer fixed point theorem that there exists $x^* \in \mathbf{x}$ such that $g(x^*) = x^*$. Hence, $f(x^*) = 0$. ■

Theorem B.16. Given a function $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ and a closed bounded box $\mathbf{x} \subseteq D$. If there exists a continuous function $g : D' \supseteq \mathbf{x} \to \mathbb{R}^n$ and its interval form $[g] : \mathbb{I}^n \to \mathbb{I}^n$ such that $[g](\mathbf{x}) \subseteq \mathbf{x}$ and

$$\forall x \in \mathbf{x} : (g(x) = x \Rightarrow f(x) = 0)$$

hold, then there exists a zero of f in $\mathbf{x}$; that is, $f(x) = 0$ for some $x \in \mathbf{x}$.

Proof. The result follows Theorem B.15 directly, with noticing that $\forall x \in \mathbf{x} : g(x) \in [g](\mathbf{x})$. ■

Although the assumption in Theorem B.16 is stronger than the assumption in Theorem B.15, it is often used in interval-based solution methods because of its simplicity and efficiency (see Section 3.1.1).

The compactness condition on C in Theorem B.14 is a strong one. Many problems do not have a compact setting. Schauder has relaxed that condition and proved the following theorem (see [Agarwal *et al.* 2001b, Theorem 4.14]).

Theorem B.17 (Schauder Fixed Point Theorem). Every *compact continuous mapping* from a *closed convex* subset C of a normed vector space to C has at least one fixed point.

The uniqueness of solution of equations is often derived from, or related to, Banach's fundamental theorem in the fixed point theory (see [Kirk and Sims 2001], [Khamsi and Kirk 2001] and [Granas and Dugundji 2003]).

Theorem B.18 (Banach Fixed Point Theorem). Every *contraction mapping* on a complete metric space has a unique fixed point.

Edelstein has extended the above Banach fixed point theorem for contractive mappings on compact metric space. Here is the formal statement (see [Agarwal *et al.* 2001b, Theorem 1.2] and [Singh *et al.* 1997, Theorem 1.33]).

Theorem B.19 (Edelstein Fixed Point Theorem). Every *contractive mapping* on a compact metric space has a unique fixed point.

Appendix C

Numerical Benchmarks

C.1. Problems with Continuums of Solutions

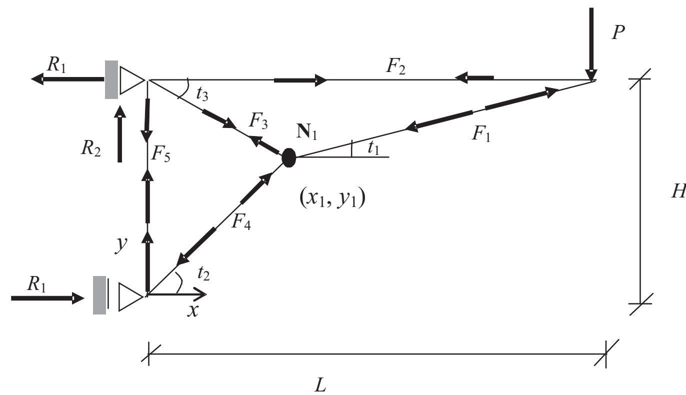

Figure C.1. The geometric design of a truss.

Problem (TD). Consider the geometric design problem of a truss depicted in Figure C.1. The goal is to find the coordinates of the moveable joint (x_1, y_1) in $[0.01, 10] \times [0.01, 10]$ of the node $\mathbf{N}_1$ of the truss such that all the following constraints are satisfied:

$$
\begin{aligned}
F_2 &< TA, \\
F_5 &< TA, \\
F_1 &< C_1 A, \\
F_1 &< TA, \\
F_4 &< C_4 A, \\
F_4 &< TA, \\
|F_3| &< TA, \\
F_3 \leq 0 &\Rightarrow -F_3 < C_3 A, \\
x_1 &< L, \\
y_1 &< H,
\end{aligned}
$$

where

$$
\begin{array}{rcll}
E & = & 210 * 10^6 & \text{Young's modulus of steel, unit} = \text{kN/m}^2; \\
T & = & 235 * 10^3 & \text{The yield stress of steel, unit} = \text{kN/m}^2; \\
A & = & 0.25 & \text{The area of cross section of truss members;} \\
r & = & 0.5 & \text{The radius of gyration of the cross section of truss members;} \\
P & = & 400 & \text{The loading capacity;} \\
H & = & 6 & \text{The height of truss;} \\
L & = & 10 & \text{The length of truss;}
\end{array}
$$

and the auxiliary variables are defined as follows:

$$
\begin{array}{rcl}
\tan t_1 & = & (H - y_1)/(L - x_1), \\
\tan t_2 & = & y_1/x_1, \\
\tan t_3 & = & (H - y_1)/x_1, \\
R_1 & = & PL/H, \\
F_1 & = & P/\sin t_1, \\
F_2 & = & P/\tan t_1, \\
F_3 & = & (R_1 - F_2)/\cos t_3, \\
F_4 & = & R_1/\cos t_2, \\
F_5 & = & R_1 \tan t_2, \\
L_1 & = & \sqrt{(L - x_1)^2 + (H - y_1)^2}, \\
L_3 & = & \sqrt{x_1^2 + (H - y_1)^2}, \\
L_4 & = & \sqrt{x_1^2 + y_1^2}, \\
C_1 & = & \pi^2 E/(L_1/r)^2, \\
C_3 & = & \pi^2 E/(L_3/r)^2, \\
C_4 & = & \pi^2 E/(L_4/r)^2.
\end{array}
$$

In fact, this problem is two-dimensional: the variables are x_1 and y_1. All the other variables can be easily eliminated in a preprocessing phase. The reduced constraints are however too complicated (in the number of elementary operations) to be read, hence is not listed here. ■

Problem (FD). Consider the design problem of the beam of a railway bridge under cyclic stress. The goal is to find $(L, q_f, Z) \in [10, 30] \times [70, 90] \times [0.1, 10]$ such that the following yield stress and fatigue stress are satisfied:[1]

$$
\begin{array}{rcl}
\sigma & < & f_y, \\
\sigma_e & < & \text{resistance,}
\end{array}
$$

where

$$
\begin{array}{rcll}
\sigma_c & = & 115000 & \text{Yield stress of steel, unit} = \text{kN/m}^2; \\
\gamma & = & 1.1 & \text{The safety factor;} \\
f_y & = & 460000 & \text{Unit} = \text{kN/m}^2; \\
\text{years} & = & 100 & \text{The number of years to fatigue failure;}
\end{array}
$$

[1] The variable Z is scaled up 100 times in unit in comparison to the original version.

and the auxiliary variables are defined as follows:

$$\alpha = \begin{cases} 1.3 & \text{if } L \leq 4, \\ 1.3 - 0.1(L-4) & \text{if } 4 < L \leq 7.5, \\ 0.95 - 0.008(L-7.5) & \text{if } 7.5 < L \leq 20, \\ 0.85 - (L-20)/300 & \text{if } 20 < L \leq 50, \\ 0.75 & \text{if } L > 50, \end{cases}$$

$$\begin{aligned} \phi &= 0.82 + 1.44/(\sqrt{L} - 0.2), \\ q_r &= q_f \phi, \\ \sigma &= q_r L^2/8/(Z/100), \\ \sigma_e &= \alpha\sigma, \\ \text{cycles} &= 0.05 \text{ years}, \\ \sigma_r &= \sigma_c(\min\{2.5, \text{cycles}/2\})^{-1/3}, \\ \text{resistance} &= \sigma_r/\gamma, \end{aligned}$$

■

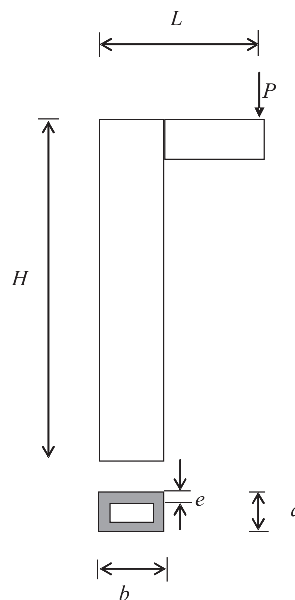

Figure C.2. The design of a column for combined axial load and moment.

Problem (CD). Consider the design problem of a column for combined axial load and moment, depicted in Figure C.2. The goal is to find $(a, b, e) \in [0.01, 2] \times [0.01, 1] \times [0.05, 0.1]$, where a and b are in meters, e is in decimeters.[2]

$$\begin{aligned} \text{loadByResistance} &< 1, \\ W &< \text{maxWeight}, \\ N_d &< N_{cr}, \\ a &\geq 4e/10, \end{aligned}$$

[2] The variable e is scaled up 10 times in unit in comparison to the original version.

$$b \geq 4e/10,$$

where the constants are given as follows

$$\begin{array}{rcll}
\rho &=& 78 & \text{The density of steel, unit} = \text{kN/m}^3;\\
f_y &=& 235 * 10^3 & \text{The yield stress of steel, unit} = \text{kN/m}^2;\\
E &=& 210 * 10^6 & \text{Young's modulus of steel, unit} = \text{kN/m}^2;\\
\gamma_R &=& 1.1 & \text{The safety factor;}\\
\alpha &=& 0.21; & \\
\lambda_E &=& 94; & \\
P &=& 400 & \text{The loading capacity, unit} = \text{kN};\\
H &=& 6 & \text{The length of column, unit} = \text{m};\\
L &=& 1 & \text{The height of column, unit} = \text{m};\\
\text{maxWeight} &=& 10 & \text{Unit} = \text{kN};
\end{array}$$

and the auxiliary variables are defined as follows:

$$\begin{array}{rcl}
N_d &=& P,\\
M_d &=& N_d(L - b/2),\\
A &=& 2b(e/10) + 2(a - 2(e/10))(e/10),\\
W &=& \rho A H,\\
I_y &=& ab^3/12 - (a - 2(e/10))(b - 2(e/10))^3/12,\\
l_k &=& 2H,\\
N_{cr} &=& \pi^2 E I_y / l_k^2,\\
Z &=& I_y/(b/2),\\
M_r &=& Z f_y,\\
r_y &=& \sqrt{I_y/A},\\
\lambda_K &=& 2H/r_y,\\
\lambda_{KBar} &=& \lambda_K/\lambda_E,\\
\phi &=& 0.5(1 + \alpha(\lambda_{KBar} - 0.2) + \lambda_{KBar}^2),\\
t_1 &=& \max\{0, \phi^2 - \lambda_{KBar}^2\},\\
k &=& \min\{1, 1/(\phi + \sqrt{t_1})\},\\
N_\Lambda &=& k f_y A,\\
t_2 &=& (1 - N_d/N_{cr}),\\
\text{loadByResistance} &=& N_d/(N_\Lambda/\gamma_R) + (1/t_2) M_d/(M_r/\gamma_R).
\end{array}$$

In fact, this problem has only three variables. However, if we eliminate the auxiliary variables, the formulas will become very complicated and extremely difficult to check for errors. This kind of problems is usually difficult for interval-based methods because each numerical evaluation of functions is very expensive. ■

Problem (WP). We consider the problem **WP** in Example 4.1:

$$\mathbf{WP} \equiv \begin{cases} 20 < \sqrt{x^2 + y^2} < 50,\\ 12y/\sqrt{(x-12)^2 + y^2} < 10,\\ x \in [-50, 50],\ y \in [0, 50]. \end{cases}$$

In fact, this is a simple problem but it makes many interval constraint propagation techniques, such as the **HC4** algorithm, slow (see Section 3.2.2.2). Hence, it is interesting to take experiments on this problem. ∎

Problem (P2). Consider an artificial problem that simulates the case where the solution set contains a large boundary region nearly parallel to axes.

$$\mathbf{P2} \equiv \begin{cases} x^2 \leq y, \\ \ln y + 1 \geq z, \\ xz \leq 1, \\ x \in [0, 15],\ y \in [1, 200],\ z \in [-10, 10]. \end{cases}$$

The arity of constraints are all less the arity of the problem. The problem is simple; hence, we expect the time percentage for evaluations is not dominant in the time for the whole solving process. The comparison of search techniques is therefore clearer. ∎

Problem (P3). This problem is similar to the problem **P2**, but a highly nonlinear is added to make the problem harder.

$$\mathbf{P3} \equiv \begin{cases} x^2 \leq y, \\ \ln y + 1 \geq z, \\ xz \leq 1, \\ x^{3/2} + \ln(1.5z + 1) \leq y + 1, \\ x \in [0, 15],\ y \in [1, 200],\ z \in [0, 10]. \end{cases}$$

The arity of a constraint equals to the arity of the problem. Since this problem has only a small number of operations, it will show better the difference between the search techniques (the time percentage of evaluations is not dominant). ∎

C.2. Test Case T_1: Problems with Isolated Solutions

Problem (BIF3). Bifurcation Problem:

$$\mathbf{BIF3} = \begin{cases} 5x^9 - 6x^5y^2 + xy^4 + 2xz = 0; \\ -2x^6y + 2x^2y^3 + 2yz = 0; \\ x^2 + y^2 = 0.265625; \end{cases}$$

where x, y, z in $[-10^8, 10^8]$. ∎

Problem (ECO5). An economic problem:

$$\mathbf{ECO5} = \begin{cases} (x_1 + x_1x_2 + x_2x_3 + x_3x_4)x_5 - 1 & = & 0; \\ (x_2 + x_1x_3 + x_2x_4)x_5 - 2 & = & 0; \\ (x_3 + x_1x_4)x_5 - 3 & = & 0; \\ x_4x_5 - 4 & = & 0; \\ x_1 + x_2 + x_3 + x_4 + 1 & = & 0; \end{cases}$$

where $x_1, \dots, x_5$ in $[-10, 10]$. ∎

Problem (ECO6). An economic problem:

$$\mathbf{ECO6} = \begin{cases} (x_1 + x_1x_2 + x_2x_3 + x_3x_4 + x_4x_5)x_6 - 1 & = & 0; \\ (x_2 + x_1x_3 + x_2x_4 + x_3x_5)x_6 - 2 & = & 0; \\ (x_3 + x_1x_4 + x_2x_5)x_6 - 3 & = & 0; \\ (x_4 + x_1x_5)x_6 - 4 & = & 0; \\ x_5x_6 - 5 & = & 0; \\ x_1 + x_2 + x_3 + x_4 + x_5 + 1 & = & 0; \end{cases}$$

where $x_1, \ldots, x_6$ in $[-10, 10]$. ■

Problem (ECO7). An economic problem:

$$\mathbf{ECO7} = \begin{cases} (x_1 + x_1x_2 + x_2x_3 + x_3x_4 + x_4x_5 + x_5x_6)x_7 - 1 & = & 0; \\ (x_2 + x_1x_3 + x_2x_4 + x_3x_5 + x_4x_6)x_7 - 2 & = & 0; \\ (x_3 + x_1x_4 + x_2x_5 + x_3x_6)x_7 - 3 & = & 0; \\ (x_4 + x_1x_5 + x_2x_6)x_7 - 4 & = & 0; \\ (x_5 + x_1x_6)x_7 - 5 & = & 0; \\ x_6x_7 - 6 & = & 0; \\ x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + 1 & = & 0; \end{cases}$$

where $x_1, \ldots, x_7$ in $[-10, 10]$. ■

Problem (ECO8). An economic problem:

$$\mathbf{ECO8} = \begin{cases} (x_1 + x_1x_2 + x_2x_3 + x_3x_4 + x_4x_5 + x_5x_6 + x_6x_7)x_8 - 1 & = & 0; \\ (x_2 + x_1x_3 + x_2x_4 + x_3x_5 + x_4x_6 + x_5x_7)x_8 - 2 & = & 0; \\ (x_2 + x_1x_3 + x_2x_4 + x_3x_5 + x_4x_6)x_7 - 2 & = & 0; \\ (x_4 + x_1x_5 + x_2x_6 + x_3x_7)x_8 - 4 & = & 0; \\ (x_5 + x_1x_6 + x_2x_7)x_8 - 5 & = & 0; \\ (x_6 + x_1x_7)x_8 - 6 & = & 0; \\ x_7x_8 - 7 & = & 0; \\ x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + 1 & = & 0; \end{cases}$$

where $x_1, \ldots, x_8$ in $[-10, 10]$. ■

Problem (NEU6). Neurophysiology Problem:

$$\mathbf{NEU6} = \begin{cases} x_1^2 + x_3^2 & = & 1; \\ x_2^2 + x_4^2 & = & 1; \\ x_5x_1^3 + x_6x_2^3 & = & 5; \\ x_5x_1x_3^2 + x_6x_4^2x_2 & = & 4; \\ x_5x_3^3 + x_6x_4^3 & = & 3; \\ x_5x_1^2x_3 + x_6x_2^2x_4 & = & 2; \\ x_1 & \geq & x_2; \\ x_1 & \geq & 0; \\ x_2 & \geq & 0; \end{cases}$$

where $x_1, \ldots, x_6$ in $[-100, 100]$. ■

Problem (REI3). Neurophysiology Problem:

$$\textbf{REI3} = \begin{cases} x^2 - y^2 + z^2 & = & 0.5; \\ x^3 - y^3 + z^3 & = & 0.5; \\ x^4 - y^4 + z^4 & = & 0.5; \\ 2xy + 6y^2 + 2yz - 2x - 4y - 2z + 1 & = & 0; \end{cases}$$

where x, y, z in $[-10, 10]$. ■

Problem (WIN3). Neurophysiology Problem:

$$\textbf{WIN3} = \begin{cases} 4xz - 4xy^2 - 16x^2 - 1 & = & 0; \\ 2y^2z + 4x + 1 & = & 0; \\ 2x^2z + 2y^2 + x & = & 0; \\ 2xy + 6y^2 + 2yz - 2x - 4y - 2z + 1 & = & 0; \end{cases}$$

where x, y, z in $[-10^5, 10^5]$. ■

C.3. Test Case T_2: Problems with Isolated Solutions

Problem (CYC5). Cyclic Problem:

$$\textbf{CYC5} = \begin{cases} a + b + c + d + e & = & 0; \\ ab + bc + cd + de + ea & = & 0; \\ abc + bcd + cde + dea + eab & = & 0; \\ abcd + bcde + cdea + deab + eabc & = & 0; \\ abcde - 1 & = & 0; \end{cases}$$

where a, b, c, d, e in $[-10, 10]$. ■

Problem (GS5.1). Gough Steward Problem:

$$\textbf{GS5.1} = \begin{cases} x_1^2 + y_1^2 + z_1^2 & = & 31, \\ x_2^2 + y_2^2 + z_2^2 & = & 39, \\ x_3^2 + y_3^2 + z_3^2 & = & 29, \\ x_1x_2 + y_1y_2 + z_1z_2 + 6x_1 - 6x_2 & = & 51, \\ x_1x_3 + y_1y_3 + z_1z_3 + 7x_1 - 2y_1 - 7x_3 + 2y_3 & = & 50, \\ x_2x_3 + y_2y_3 + z_2z_3 + x_2 - 2y_2 - x_3 + 2y_3 & = & 34, \\ -12x_1 + 15y_1 - 10x_2 - 25y_2 + 18x_3 + 18y_3 & = & -32, \\ -14x_1 + 35y_1 - 36x_2 - 45y_2 + 30x_3 + 18y_3 & = & 8, \\ 2x_1 + 2y_1 - 14x_2 - 2y_2 + 8x_3 - y_3 & = & 20, \end{cases}$$

where $x_1 \in [0.00; 5.57]$, $y_1 \in [0.00, 2.70]$, $z_1 \in [0.00, 5.57]$, $x_2 \in [-6.25, 0.00]$, $y_2 \in [-2.00, 0.00]$, $z_2 \in [0.00, 6.25]$, $x_3 \in [-5.39, -1.00]$, $y_3 \in [-5.39, 0.00]$, $z_3 \in [0.00, 5.39]$. ■

Problem (KOL2). Kolev's benchmark:

$$\textbf{KOL2} = \begin{cases} ((4x_3 + 3x_6)x_3 + 2x_5)x_3 + x_4 & = & 0, \\ ((4x_2 + 3x_6)x_2 + 2x_5)x_2 + x_4 & = & 0, \\ ((4x_1 + 3x_6)x_1 + 2x_5)x_1 + x_4 & = & 0, \\ x_4 + x_5 + x_6 + 1 & = & 0, \\ (((x_2 + x_6)x_2 + x_5)x_2 + x_4)x_2 + (((x_3 + x_6)x_3 + x_5)x_3 + x_4)x_3 & = & 0, \\ (((x_1 + x_6)x_1 + x_5)x_1 + x_4)x_1 + (((x_2 + x_6)x_2 + x_5)x_2 + x_4)x_3 & = & 0, \end{cases}$$

where $x_1 \in [0.0333, 0.2173]$, $x_2 \in [0.4000, 0.6000]$, $x_3 \in [0.7826, 0.9666]$, $x_4 \in [-0.3071, -0.1071]$, $x_5 \in [1.1071, 1.3071]$, $x_6 \in [-2.1000, -1.9000]$. ■

Problem (YAM60). Yama160 Problem:

$$(n+1)^2 x_{i-1} - 2(n+1)^2 x_i + (n+1)^2 x_{i+1} + e^{x_i} = 0, \quad (\text{for } i = 1, \ldots, n),$$

where $n = 60$, $x_0 = x_{n+1} = 0$, and $x_i \in [-10, 10]$ (for $i = 1, \ldots, n$), ■

C.4. Test Case T_3: Problems with Isolated Solutions

Problem (CAP4). The Caprasse system:

$$\mathbf{CAP4} = \begin{cases} y^2z + 2xyt - 2x - z & = & 0; \\ -x^3z + 4xy^2z + 4x^2yt + 2y^3t + 4x^2 - 10y^2 + 4xz - 10yt + 2 & = & 0; \\ 2yzt + xt^2 - x - 2z & = & 0; \\ -xz^3 + 4yz^2t + 4xzt^2 + 2yt^3 + 4xz + 4z^2 - 10yt - 10t^2 + 2 & = & 0; \end{cases}$$

where x, y, z, t in $\mathbb{R}$. ■

Problem (DID9). Didrit Problem:

$$\mathbf{DID9} = \begin{cases} x_1^2 + y_1^2 + z_1^2 = 31; \\ x_2^2 + y_2^2 + z_2^2 = 39; \\ x_3^2 + y_3^2 + z_3^2 = 29; \\ x_1x_2 + y_1y_2 + z_1z_2 + 6x_1 - 6x_2 = 51; \\ x_1x_3 + y_1y_3 + z_1z_3 + 7x_1 - 2y_1 - 7x_3 + 2y_3 = 50; \\ x_2x_3 + y_2y_3 + z_2z_3 + x_2 - 2y_2 - x_3 + 2y_3 = 34; \\ -12x_1 + 15y_1 - 10x_2 - 25y_2 + 18x_3 + 18y_3 = -32; \\ -14x_1 + 35y_1 - 36x_2 - 45y_2 + 30x_3 + 18y_3 = 8; \\ 2x_1 + 2y_1 - 14x_2 - 2y_2 + 8x_3 - y_3 = 20; \end{cases}$$

where x_i, y_i, z_i in $[-10, 10]$ for $i = 1, 2, 3$. ■

Problem (GS5.0). Gough Steward problem:

$$\mathbf{GS5.0} = \begin{cases} x_1^2 + y_1^2 + z_1^2 & = & 31, \\ x_2^2 + y_2^2 + z_2^2 & = & 39, \\ x_3^2 + y_3^2 + z_3^2 & = & 29, \\ x_1x_2 + y_1y_2 + z_1z_2 + 6x_1 - 6x_2 & = & 51, \\ x_1x_3 + y_1y_3 + z_1z_3 + 7x_1 - 2y_1 - 7x_3 + 2y_3 & = & 50, \\ x_2x_3 + y_2y_3 + z_2z_3 + x_2 - 2y_2 - x_3 + 2y_3 & = & 34, \\ -12x_1 + 15y_1 - 10x_2 - 25y_2 + 18x_3 + 18y_3 & = & -32, \\ -14x_1 + 35y_1 - 36x_2 - 45y_2 + 30x_3 + 18y_3 & = & 8, \\ 2x_1 + 2y_1 - 14x_2 - 2y_2 + 8x_3 - y_3 & = & 20, \end{cases}$$

where $x_1 \in [-2.00; 5.57]$, $y_1 \in [-5.57, 2.70]$, $z_1 \in [0.00, 5.57]$, $x_2 \in [-6.25, 1.30]$, $y_2 \in [-6.25, 2.70]$, $z_2 \in [-2.00, 6.25]$, $x_3 \in [-5.39, 0.70]$, $y_3 \in [-5.39, 3.11]$, $z_3 \in [-3.61, 5.39]$. ■

Problem (KAT8). Katsura Problem:

$$\mathbf{KAT8} = \begin{cases} -x_1 + 2x_8^2 + 2x_7^2 + 2x_6^2 + 2x_5^2 + 2x_4^2 + 2x_3^2 + 2x_2^2 + x_1^2 & = 0; \\ -x_2 + 2x_8x_7 + 2x_7x_6 + 2x_6x_5 + 2x_5x_4 + 2x_4x_3 + 2x_3x_2 + 2x_2x_1 & = 0; \\ -x_3 + 2x_8x_6 + 2x_7x_5 + 2x_6x_4 + 2x_5x_3 + 2x_4x_2 + 2x_3x_1 + x_2^2 & = 0; \\ -x_4 + 2x_8x_5 + 2x_7x_4 + 2x_6x_3 + 2x_5x_2 + 2x_4x_1 + 2x_3x_2 & = 0; \\ -x_5 + 2x_8x_4 + 2x_7x_3 + 2x_6x_2 + 2x_5x_1 + 2x_4x_2 + x_3^2 & = 0; \\ -x_6 + 2x_8x_3 + 2x_7x_2 + 2x_6x_1 + 2x_5x_2 + 2x_4x_3 & = 0; \\ -x_7 + 2x_8x_2 + 2x_7x_1 + 2x_6x_2 + 2x_5x_3 + x_4^2 & = 0; \\ -1 + 2x_8 + 2x_7 + 2x_6 + 2x_5 + 2x_4 + 2x_3 + 2x_2 + x_1 & = 0; \end{cases}$$

where $x_1, \ldots, x_8$ in $[-10, 10]$. ■

Problem (KIN9). Kinematics Problem:

$$\mathbf{KIN9} = \begin{cases} z_1^2 + z_2^2 + z_3^2 - 12z_1 - 68 & = 0; \\ z_4^2 + z_5^2 + z_6^2 - 12z_5 - 68 & = 0; \\ z_7^2 + z_8^2 + z_9^2 - 24z_8 - 12z_9 + 100 & = 0; \\ z_1z_4 + z_2z_5 + z_3z_6 - 6z_1 - 6z_5 - 52 & = 0; \\ z_1z_7 + z_2z_8 + z_3z_9 - 6z_1 - 12z_8 - 6z_9 + 64 & = 0; \\ z_4z_7 + z_5z_8 + z_6z_9 - 6z_5 - 12z_8 - 6z_9 + 32 & = 0; \\ 2z_2 + 2z_3 - z_4 - z_5 - 2z_6 - z_7 - z_9 + 18 & = 0; \\ z_1 + z_2 + 2z_3 + 2z_4 + 2z_6 - 2z_7 + z_8 - z_9 - 38 & = 0; \\ z_1 + z_3 - 2z_4 + z_5 - z_6 + 2z_7 - 2z_8 + 8 & = 0; \end{cases}$$

where $z_1, \ldots, z_9$ in $[-1000, 1000]$. ■

Problem (CAP4). The Caprasse system:

$$\mathbf{CAP4} = \begin{cases} y^2z + 2xyt - 2x - z & = 0; \\ -x^3z + 4xy^2z + 4x^2yt + 2y^3t + 4x^2 - 10y^2 + 4xz - 10yt + 2 & = 0; \\ 2yzt + xt^2 - x - 2z & = 0; \\ -xz^3 + 4yz^2t + 4xzt^2 + 2yt^3 + 4xz + 4z^2 - 10yt - 10t^2 + 2 & = 0; \end{cases}$$

where x, y, z, t in $\mathbb{R}$. ■

Problem (REI4). The Reinmer system:

$$\mathbf{REI4} = \begin{cases} x^2 - y^2 + z^2 - t^2 = 0.5; \\ x^3 - y^3 + z^3 - t^3 = 0.5; \\ x^4 - y^4 + z^4 - t^4 = 0.5; \\ x^5 - y^5 + z^5 - t^5 = 0.5; \end{cases}$$

where x, y, z, t in $[-10, 10]$. ■

Problem (REI5). The Reinmer system:

$$\mathbf{REI5} = \begin{cases} -1 + 2x_1^2 - 2x_2^2 + 2x_3^2 - 2x_4^2 + 2x_5^2 = 0; \\ -1 + 2x_1^3 - 2x_2^3 + 2x_3^3 - 2x_4^3 + 2x_5^3 = 0; \\ -1 + 2x_1^4 - 2x_2^4 + 2x_3^4 - 2x_4^4 + 2x_5^4 = 0; \\ -1 + 2x_1^5 - 2x_2^5 + 2x_3^5 - 2x_4^5 + 2x_5^5 = 0; \\ -1 + 2x_1^6 - 2x_2^6 + 2x_3^6 - 2x_4^6 + 2x_5^6 = 0; \end{cases}$$

where $x_1, \ldots, x_5$ in $[-1, 1]$. ■

Problem (REI6). The Reinmer system:

$$\mathbf{REI6} = \langle -0.5 + \sum_{i=1}^{n} (-1)^{i+1} x_i^k = 0 \ (k = 1, \dots, n);\ n = 6,\ x_i \in [-1, 1] \text{ (for } i = 1, \dots, n)\rangle \quad \blacksquare$$

C.5. Test Case T_4: Problems with Continuums of Solutions

Problem (F2.2). Tricuspoid and Circle:

$$\mathbf{F2.2} = \begin{cases} (x^2 + y^2 + 12x + 9)^2 \leq 4(2x + 3)^3; \\ x^2 + y^2 \geq 2; \end{cases}$$

where x, y in $[-2, 2]$. $\blacksquare$

Problem (F2.3). Foliumd, Circle, and Trifolium:

$$\mathbf{F2.3} = \begin{cases} x^3 + y^3 \geq 3xy; \\ x^2 + y^2 \geq 0.1; \\ (x^2 + y^2)(y^2 + x(x + 1)) \leq 4xy^2; \end{cases}$$

where x, y in $[-3, 3]$. $\blacksquare$

Problem (S04). Circle:

$$\mathbf{S05} = \langle x^2 + y^2 \leq 1;\ x, y \in [-2, 2]\rangle \quad \blacksquare$$

Problem (S05). Circle:

$$\mathbf{S05} = \langle \frac{x}{\sqrt{(y - 5)^2 + 1}} \leq 1;\ x, y \in [1, 10]\rangle \quad \blacksquare$$

Problem (S06). Circle:

$$\mathbf{S06} = \langle \frac{12y}{\sqrt{(x - 12)^2 + y^2}} \leq 10;\ x \in [-50, 50],\ y \in [0, 50]\rangle \quad \blacksquare$$

Problem (WP). Circle:

$$\mathbf{WP} = \langle 20 \leq \sqrt{x^2 + y^2} \leq 50,\ \frac{12y}{\sqrt{(x - 12)^2 + y^2}} \leq 10;\ x \in [-50, 50],\ y \in [0, 50]\rangle \quad \blacksquare$$

C.6. Test Case T_5: Problems with Continuums of Solutions

Problem (G1.1).

$$\mathbf{G1.1} = \begin{cases} x_1^2 + 0.5x_2 + 2(x_3 - 6) \geq 0; \\ x_1^2 + x_2^2 + x_3^2 \leq 25; \end{cases}$$

where x_1, x_2, x_3 in $[-8, 8]$. $\blacksquare$

Problem (G1.2).

$$\mathbf{G1.2} = \begin{cases} x_1^2 + 0.5x_2 + 2(x_3 - 3) \geq 0; \\ x_1^2 + x_2^2 + x_3^2 \leq 25; \end{cases}$$

where x_1, x_2, x_3 in $[-8, 8]$. ■

Problem (H1.1).

$$\mathbf{H1.1} = \begin{cases} x_1^2 + x_2^2 + x_3^2 \leq 9; \\ (x_1 - 0.5)^2 + (x_2 - 1)^2 + x_3^2 \geq 4; \\ x_1^2 + (x_2 - 0.2)^2 \geq x_3; \end{cases}$$

where x_1, x_2, x_3 in $[-4, 4]$. ■

Problem (P1.4).

$$\mathbf{P1.4} = \begin{cases} x^2 + y^2 + z^2 <= 4; \\ (x - 2)^2 + y^2 + z^2 >= 4; \end{cases}$$

where x, y, z in $[-4, 4]$. ■

Problem (P2).

$$\mathbf{P2} \equiv \begin{cases} x^2 \leq y, \\ \ln y + 1 \geq z, \\ xz \leq 1, \end{cases}$$

where $x \in [0, 15],\ y \in [1, 200],\ z \in [-10, 10]$. ■

Problem (P3).

$$\mathbf{P3} \equiv \begin{cases} x^2 \leq y, \\ \ln y + 1 \geq z, \\ xz \leq 1, \\ x^{3/2} + \ln(1.5z + 1) \leq y + 1, \end{cases}$$

where $x \in [0, 15],\ y \in [1, 200],\ z \in [0, 10]$. ■

Bibliography

P. K. Agarwal, M. T. de Berg, J. G. Gudmundsson, M. Hammar, and H. J. Haverkort. Box-Trees and R-Trees with Near-Optimal Query Time. In *Proceedings of the 17th ACM Symposium on Computational Geometry*, pages 124–133. ACM Press, 2001a. @pages 106, 189, 190

R. P. Agarwal, M. Meehan, and D. O'Regan. *Fixed Point Theory and Applications.* Cambridge University Press, Cambridge, UK, 2001b. ISBN 0-521-80250-4. @pages 220, 221

A. Aguilera. *Orthogonal Polyhedra: Study and Application.* PhD thesis, Universitat Politècnica de Catalunya, Barcelona, Spain, April 1998. @pages 103, 105, 116, 118, 119

A. Aguilera and D. Ayala. The Extreme Vertices Model for Orthogonal Polyhedra. Technical Report LSI-97-6-R, LSI-Universitat Politècnica de Catalunya, Barcelona, Spain, 1997. @pages 116

F. A. Al-Khayyal. Jointly Constrained Biconvex Programming and Related Problems: An Overview. *Computers and Methematics with Applications*, 19(11):53–62, 1990. @pages 91

F. A. Al-Khayyal and J. E. Falk. Jointly Constrained Biconvex Programming. *Mathematics of Operations Research*, 8(2):273–286, 1983. @pages 91

G. Alefeld and J. Herzberger. ALGOL-60 Algorithmen zur Auflösung linearer Gleichungssysteme mit Fehlerfassung. *Computing*, 6:28–34, 1970. @pages 58

G. Alefeld and J. Herzberger. *Introduction to Interval Computations.* Academic Press, New York, NY, 1983. @pages 32

E. Allgower, M. Erdmann, and K. Georg. On the Complexity of Exclusion Algorithms for Optimization. *Journal of Complexity*, 18(2):573–588, June 2002. @pages 70

E. Allgower and K. Georg. Simplicial and Continuation Methods for Approximating Fixed Points and Solutions to Systems of Equations. *SIAM Review*, 22(1):28–85, 1980. @pages 100

K. R. Apt. The Essence of Constraint Propagation. *Theoretical Computer Science*, 221(1-2): 179–210, 1999. @pages 90

K. R. Apt. *Principles of Constraint Programming.* Cambridge University Press, Cambridge, UK, 2003. ISBN 0-521-82583-0. @pages 1, 2, 9, 11, 14, 21, 22, 23, 43, 44, 46, 47, 54, 79, 100

Archimedes. *The Works of Archimedes*, chapter Measurement of A Circle, pages 91–98. Dover Publications, New York, 1953. Collected by Heath, Thomas L. @pages 209

E. Asarin, O. Bournez, T. Dang, O. Maler, and A. Pnueli. Effective Synthesis of Switching Controllers for Linear Systems. *Proceedings of the IEEE, Special Issue on "Hybrid Systems"*, 88(7):1011–1025, July 2000. @pages 101

A. Avizienis. Signed-Digit Number Representations for Fast, Parallel Arithmetic. *IRE Transactions on Electronic Computing*, EC-10:389–400, September 1961. @pages 31

A. Baar, E. Feigenbaum, and P. R. Cohen. *The Handbook of Artificial Intelligence*. Morgan Kaufmann, 1981. @pages 78

F. Benhamou. Heterogeneous Constraint Solving. In *Proceedings of 5th International Conference on Algebraic and Logic Programming (ALP'96)*, volume LNCS 1139, pages 62–76, Germany, 1996. @pages 90

F. Benhamou and F. Goualard. Universally Quantified Interval Constraints. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'2000)*, pages 67–82, 2000. @pages 102, 107, 109, 110, 125

F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revising Hull and Box Consistency. In *Proceedings of the International Conference on Logic Programming (ICLP'99)*, pages 230–244, Las Cruces, USA, 1999. @pages 52, 81, 82, 83, 84, 85, 86, 87, 88, 89, 151, 155, 160, 166, 168, 171, 183, 208

F. Benhamou and L. Granvilliers. Automatic Generation of Numerical Redundancies for Nonlinear Constraint Solving. *Reliable Computing*, 3(3):335–344, 1997. @pages 90

F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(Intervals) Revisited. In *Proceedings of the International Logic Programming Symposium*, pages 109–123, 1994. @pages 51, 169, 183

F. Benhamou and W. J. Older. Applying Interval Arithmetic to Real, Integer and Boolean Constraints. Technical Report BNR Technical Report, Bell Northern Research, Ontario, Canada, 1992. @pages 48, 81, 151, 169

F. Benhamou and W. J. Older. Applying Interval Arithmetic to Real, Integer and Boolean Constraints. *Journal of Logic Programming*, pages 32–81, 1997. Extension of a technical report of Bell Northern Research, Canada, 1992. @pages 48, 81, 151, 169

C. Bliek. *Computer Methods for Design Automation*. PhD thesis, Massachusetts Institute of Technology, Department of Ocean Engineering, 1992. @pages 66

C. Bliek. Generalizing Dynamic and Partial Order Backtracking. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 319–325, Wisconsin, USA, July 1998. @pages 79

C. Bliek and D. Sam-Haroud. Path Consistency for Triangulated Constraint Graphs. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1999. @pages 46

C. Bliek, P. Spellucci, L. N. Vincent, A. Neumaier, L. Granvilliers, E. Monfroy, F. Benhamou, E. Huens, P. Van Hentenryck, D. Sam-Haroud, and B. Faltings. Algorithms for Solving Nonlinear Constrained and Optimization Problems: The State of the Art. Technical Report D2, COCONUT Project, June 2001. @pages 89

G. Boole. *An Investigation of the Laws of Thought*. Walton, London, UK, 1847. @pages 9

L. Bordeaux, E. Monfroy, and F. Benhamou. Improved Bounds on the Complexity of kB-Consistency. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'2001)*, 2001. @pages 87, 88

G. Borradaile and P. Van Hentenryck. Safe and Tight Linear Estimators for Global Optimization. *Mathematical Programming*, 2004. To appear. @pages 91, 186, 208

O. Bournez and O. Maler. On the Representation of Timed Polyhedra. In *Proceedings of International Colloquium on Automata Languages and Programming (ICALP'2000)*, volume LNCS 1853, pages 793–807. Springer, 2000. @pages 116, 118, 119

O. Bournez, O. Maler, and A. Pnueli. Orthogonal Polyhedra: Representation and Computation. In *Hybrid Systems: Computation and Control*, volume LNCS 1569, pages 46–60. Springer, 1999. @pages 103, 116, 118

J. C. Burkill. Functions of Intervals. In *Proceedings of the London Mathematical Society*, volume 22, pages 375–446, 1924. @pages 209

H. Collavizza, F. Delobel, and M. Rueher. Extending Consistent Domains of Numeric CSP. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1999. @pages 107

A. Colmerauer. PROLOG II Reference Manual and Theoretical Model. Technical report, Groupe Intelligence Artificielle, Université Aix – Marseille II, October 1982. @pages 10

J. L. D. Comba and J. Stolfi. Affine Arithmetic and its Applications to Computer Graphics. In *Proceedings of SIBGRAPI'93*, Brazil, October 1993. @pages 36

J. Cruz and P. Barahona. Maintaining Global Hull Consistency with Local Search for Continuous CSPs. In *Global Optimization and Constraint Satisfaction: First International Workshop on Global Constraint Optimization and Constraint Satisfaction, COCOS 2002*, volume LNCS 2861, pages 178–193, Berlin Heidelberg, 2003. Springer-Verlag. @pages 49

A. Davenport, E. Tsang, W. C., and Z. K. GENET: A Connectionist Architecture for Solving Constraint Satisfaction Problems by Iterative Improvement. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 287–309, Seatle, WA, USA, 1994. @pages 100

E. Davis. Constraint Propagation with Interval Labels. *Artificial Intelligence*, 32(3):281–331, 1987. @pages 45, 49

R. Dechter. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41(3):273–312, January 1990. @pages 79

R. Dechter. *Constraint Processing*. Morgan Kaufmann, San Francisco, California, 2003. ISBN 1-55860-890-7. @pages 14

R. Dechter and J. Pearl. Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence*, 34:1–38, 1988. @pages 45, 46

R. Dechter and P. van Beek. Local and Global Relational Consistency. In *Proceedings of the First International Conference on Principles and Practices of Constraint Programming*, pages 240–257, Cassis, France, September 1995. @pages 53

R. Dechter and P. van Beek. Local and Global Relational Consistency. *Theoretical Computer Science*, 173:283–308, 1997. @pages 17, 53

M. J. Dent and R. E. Mercer. Minimal Forward Checking. In *Proceedings of the 6th IEEE International Conference on Tools with Artificial Intelligence*, pages 432–438, New Orleans, 1994. @pages 79

P. S. Dwyer. *Linear Computations*, chapter Computation with Approximate Numbers, pages 11–34. John Wiley & Son, Inc., New York, 1951. @pages 209

J. Eckhoff. *Handbook of Convex Geometry*, chapter Helly, Radon, and Carathéodory Type Theorems, pages 389–448. North-Holland, Amsterdam, Netherlands, 1993. @pages 54

A. Eiben, P.-E. Eaué, and Z. Ruttkay. Solving Constraint Satisfaction Problems Using Genetic Algorithms. In *Proceedings of the 1st IEEE Conference on Evolutionary Computing*, pages 543–547, 1994. @pages 100

C.-S. Fahn and J.-L. Wang. Efficient Time-Interrupted and Time-Continuous Collision Detection Among Polyhedral Objects in Arbitrary Motion. *Journal of Information Science and Engineering*, 15:769–799, 1999. @pages 190

C.-S. Fahn and J.-L. Wang. Adaptive Space Decomposition for Fast Visualization of Soft Object. *Journal of Visualization and Computer Animation*, 14(1):1–19, 2003. @pages 190

B. Faltings. Arc-Consistency for Continuous Variables. *Artificial Intelligence*, 65(2), 1994. @pages 45, 207

B. Faltings and E. Gelle. Local Consistency for Ternary Numeric Constraints. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 392–397, Japan, August 1997. @pages 30, 54, 207

C. Frei. *Abtraction Techniques for Resource Allocation in Communication Networks*. PhD thesis, Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland, 2000. @pages 78, 79

E. C. Freuder. Synthesizing Constraint Expressions. *Communications of the ACM*, 21:958–966, 1978. @pages 47

E. C. Freuder. A Sufficient Condition for Backtrack-Bounded Search. *Journal of the ACM*, 32 (4):755–761, 1985. @pages 53

E. C. Freuder and R. J. Wallace. Partial Constraint Satisfaction. *Artificial Intelligence*, 58 (1-3):31–70, December 1992. @pages 79

F. Ganovelli, J. Dingliana, and C. O'Sullivan. BucketTree: Improving Collision Detection Between Deformable Objects. In *Spring Conference on Computer Graphics (SCCG'2000)*, 2000. @pages 190

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, 1979. @pages 190

J. Garloff and B. Graf. Solving Strict Polynomial Inequalities by Bernstein Expansion. In *Proceedings of Symbolic Methods in Control System Analysis and Design*, pages 339–352, 1999. @pages 107

J. Garloff, C. Jansson, and A. P. Smith. Lower Bound Functions for Polynomials. *Journal of Computational and Applied Mathematics*, 157(1):207–225, 2003. @pages 99, 187

J. Gaschnig. A General Backtrack Algorithm that Eliminates Most Redundant Tests. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI-77)*, volume 1, page 457, Florida, USA, 1977. @pages 79

J. Gaschnig. Experimental Case Studies of Backtrack vs. Waltz-type vs. New Algorithms for Satisficing Assignment Problems. In *Proceedings of the Second National Conference of the Canadian Society for Computational Studies of Intelligence*, pages 268–277, 1978. @pages 79

D. M. Gay. Solving Interval Linear Equations. *SIAM Journal of Numerical Analysis*, 19: 858–870, 1982. @pages 60

E. Gelle. *On the Generation of Locally Consistent Solution Spaces in Mixed Dynamic Constraints Problems*. PhD thesis, Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland, 1998. @pages 30, 54

E. Gelle and B. Faltings. Solving Mixed and Conditional Constraint Satisfaction Problems. *Constraints*, 8(2):107–141, April 2001. @pages 30, 54

K. Georg. Improving the Efficiency of Exclusion Algorithms. *Advanced Geometry*, 1:193–210, 2001. @pages 68, 70, 71

K. Georg. A New Exclusion Test. *Journal of Computational and Applied Mathematics*, 152 (1-2):147–160, March 2003. @pages 68, 70, 71, 91, 95, 96

M. L. Ginsberg. Dynamic Backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993. @pages 79

M. L. Ginsberg and W. D. Harvey. Itarative Broadening. *Artificial Intelligence*, 55:367–383, 1992. @pages 78

M. L. Ginsberg and D. A. McAllester. GSAT and Dynamic Backtracking. In *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning (KR'94)*, pages 226–237, 1994. @pages 79

F. Glover. Future Paths for Integer Programming and Links to Artificial Intelligence. *Computer Operation Research*, 5:533–549, 1986. @pages 100

F. Glover. Tabu Search – Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989. @pages 100

F. Glover. Tabu Search – Part II. *ORSA Journal on Computing*, 2:4–32, 1990. @pages 100

D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991. @pages 30

S. W. Golomb and L. D. Baumert. Backtrack Programming. *Journal of ACM*, 12:516–524, 1965. @pages 78

A. Granas and J. Dugundji. *Fixed Point Theory*. Springer, New York, USA, 2003. ISBN 0-387-00173-5. @pages 221

L. Granvilliers. *Consistance Locales et Transformation Symboliques de Contraintes d'Intervalles*. PhD thesis, Université d'Orléans, France, 1998. @pages 48

L. Granvilliers. On the Combination of Interval Constraint Solvers. *Reliable Computing*, 7(6): 467–483, 2001. @pages 89

L. Granvilliers and F. Benhamou. Progress in the Solving of a Circuit Design Problem. *Journal of Global Optimization*, 2001. @pages 82

L. Granvilliers, F. Goualard, and F. Benhamou. Box Consistency through Weak Box Consistency. In *Proceedings of the 11th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'99)*, pages 373–380, November 1999. @pages 52, 88, 89

L. Granvilliers and G. Hains. A Conservative Scheme for Parallel Interval Narrowing. *Information Processing Letters*, 74:141–146, 2000. @pages 89

L. Granvilliers and E. Monfroy. Declarative Modelling of Constraint Propagation Strategies. In *Proceedings of the International Conference on Advances in Information Systems*, volume LNAI 1909, Turkey, 2000. Springer-Verlag. @pages 89

L. Granvilliers, E. Monfroy, and F. Benhamou. Symbolic-Interval Cooperation in Constraint Programming. In *Proceedings of International Symposium on Symbolic and Algebraic Computation (ISSAC'2001)*, pages 150–166. ACM Press, July 2001. @pages 89

E. R. Hansen. On Solving Systems of Equations Using Interval Arithmetic. *Mathematical Programming*, 22:374–384, 1968. @pages 210

E. R. Hansen. A Generalized Interval Arithmetic. In *Interval Mathematics*, volume LNCS 29, pages 7–18. Springer, 1975. @pages 36, 39, 144

E. R. Hansen. Interval Forms of Newton's Method. *Computing*, 20:153–163, 1978. @pages 214

E. R. Hansen. Bounding the Solution of Interval Linear Equations. *SIAM Journal of Numerical Analysis*, 29:1493–1503, 1992. @pages 66

E. R. Hansen. Preconditioning Linearized Equations. *Computing*, 58:187–196, 1997. @pages 74

E. R. Hansen. The Hull of Preconditioned Interval Linear Equations. *Reliable Computing*, 6: 2, 2000. @pages 66, 67

E. R. Hansen and S. Sengupta. Bounding Solutions of Systems of Equations Using Interval Analysis. *BIT Numerical Mathematics*, 21:203–211, 1981. @pages 60, 62

E. R. Hansen and G. W. Walster. *Global Optimization Using Interval Analysis.* Marcel Dekker, second edition, 2004. ISBN 0-8247-4059-9. @pages 32, 65, 66, 67

R. M. Haralick and L. G. Elliot. Increasing Tree Search Consistency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980. @pages 45, 79

W. D. Harvey and M. L. Ginsberg. Limited Discrepancy Search. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence*, pages 607–615, August 1995. @pages 78

H. J. Haverkort, M. de Berg, and J. Gudmundsson. Box-Trees for Collision Checking in Industrial Installations. In *Proceedings of the 18th ACM Symposium on Computational Geometry*, pages 53–62. ACM Press, 2002. @pages 190

G. Heindl, V. Kreinovich, and A. Lakeyev. Solving Linear Interval Systems Is NP-Hard Even If We Exclude Overflow and Underflow. *Reliable Computing*, 4:383–388, 1998. @pages 57

T. J. Hickey, Q. Ju, and M. H. Van Emden. Interval Arithmetic: from Principles to Implementation. *Journal of the ACM (JACM)*, 48(5):1038–1068, 2001. @pages 33, 132

S. Hongthong and R. B. Kearfott. Rigorous Linear Overestimators and Underestimators. *Mathematical Programming B*, 2004. Submitted. @pages 91, 186, 208

ILOG. *ILOG Solver 5.3. Reference Manual*, 2003. @pages 101, 103, 115, 126

Y. Inoue, S. Kusanobu, and K. Yamamura. A Practical Approach for the Fixed-Point Homotopy Method Using a Solution-Tracing Circuit. *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences*, E85-A(1):222–233, January 2002. @pages 100

Y. Inoue, S. Kusanobu, K. Yamamura, and M. Ando. An Initial Solution Algorithm for Globally Convergent Homotopy Methods. *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences*, E87-A(4):780–786, April 2004. @pages 100

Y. Inoue, S. Kusanobu, K. Yamamura, and T. Takahashi. Newton-Fixed-Point Homotopy Method for Finding DC Operating-Points of Nonlinear Circuits. In *Proceedings of the 2001 International Technical Conference on Circuits/Systems, Computers and Communications*, pages 370–373, July 2001. @pages 100

J. Jaffar, J.-L. Lassez, and M. Maher. A Logic Programming Language Scheme. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Relations, Functions and Equations*, pages 441–468. Prentice Hall, 1986. @pages 10

J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *Proceedings of the 4th International Conference on Logic Programming*, pages 196–218, Melbourne, Australia, May 1987. @pages 10

C. Jansson. Convex-Concave Extensions. *BIT Numerical Mathematics*, 40(2):291–313, January 2000. @pages 99, 187

L. Jaulin. *Solution Globale et Guarantie de Problèmes Ensemblistes: Application à l'Estimation Non Linéaire et à la Commande Robuste.* PhD thesis, Université Paris-Sud, Orsay, France, 1994. @pages 102, 107

L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis*. Springer, first edition, 2001. ISBN 1-85233-219-0. @pages 32, 35, 76, 78, 82, 89, 101, 146, 210, 211

L. Jaulin and E. Walter. Set Inversion via Interval Analysis for Nonlinear Bounded-Error Estimation. *Automatica*, 29(4):1053–1064, 1993. @pages 76, 78

W. M. Kahan. *A More Complete Interval Arithmetic*. Lecture Notes for an Engineering Summer Course in Numerical Analysis. University of Michigan, USA, 1968. @pages 61

M. Kashiwagi. Interval Arithmetic with Linear Programming – Extentions of Yamamura's Idea. In *Proceedings of 1996 International Symposium on Nonlinear Theory and its Applications*, pages 61–64, Kochi, Japan, October 1996. @pages 90, 95

M. Kashiwagi. Simplex Method for Calculating Optimal Value with Guaranteed Accuracy. In *Proceedings of 1997 International Symposium on Nonlinear Theory and its Applications*, pages 317–320, Hawaii, USA, 1997. @pages 90, 95

R. B. Kearfott. Abstract Generalized Bisection and A Cost Bound. *Mathematical Computing*, 49(179):187–202, July 1987a. @pages 75, 76

R. B. Kearfott. Some Tests of Generalized Bisection. *ACM Transactions on Mathematical Software (TOMS)*, 13(3):197–220, September 1987b. @pages 75, 76

R. B. Kearfott and K. Du. The Cluster Problem in Global Optimization: The Univariate case. *Computing (Suppl.)*, 9:117–127, 1992. @pages 36, 68

R. B. Kearfott and K. Du. The Cluster Problem in Multivariate Global Optimization. *Journal of Global Optimization*, 4:253–265, 1994. @pages 36, 68

R. B. Kearfott and X. Shi. Optimal Preconditionners for Interval Gauss-Seidel Methods. In *Scientific Computing and Validated Numerics*, pages 173–178, 1996. @pages 89

M. A. Khamsi and W. A. Kirk. *An Introduction to Metric Spaces and Fixed Point Theory*. Wiley, New York, USA, 2001. ISBN 0-471-41825-0. @pages 217, 221

W. A. Kirk and B. Sims. *Handbook of Metric Fixed Point Theory*. Kluwer Academic Publishers, Dordrecht, 2001. ISBN 0-7923-7073-2. @pages 220, 221

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–690, 1983. @pages 100

L. V. Kolev. A New Method for Global Solution of Systems of Non-Linear Equations. *Realiable Computing*, 4:125–146, 1998. @pages 29, 39, 96, 97, 98

L. V. Kolev. An Improved Method for Global Solution of Non-Linear Systems. *Realiable Computing*, 5:103–111, 1999. @pages 29, 96, 99

L. V. Kolev. Automatic Computation of a Linear Interval Enclosure. *Realiable Computing*, 7: 17–18, 2001. @pages 29, 39, 40, 41, 96, 99, 169

L. V. Kolev. An Improved Interval Linearization for Solving Non-Linear Problems. In *10th GAMM – IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN2002)*, France, September 2002. @pages 29, 39, 42, 96, 99, 137, 169, 185, 186, 208

A. N. Kolmogorov. On the Representation of Continuous Functions of Many Variables by Superposition of Continuous Functions of One Variable and Addition. *Amerian Mathematical Biophysics*, 5:55–59, 1963. (Translated version of the Russian version in 1957). @pages 29

R. E. Korf. Improved Limited Discrepancy Search. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 286–291, 1996. @pages 78

R. Krawczyk. Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. *Computing*, 4:187–201, 1969. @pages 57, 61

R. Krawczyk. A Remark about the Convergence of Interval Sequences. *Computing*, 31:255–259, 1983. @pages 61

R. Krawczyk. A Class of Interval-Newton-Operators. *Computing*, 37:179–183, 1986. @pages 61

R. Krawczyk. Conditionally Isotone Interval Operators. *Computing*, 39:261–270, 1987. @pages 62

R. Krawczyk and A. Neumaier. Interval Slopes for Rational Functions and Associated Centered Forms. *SIAM Journal of Numerical Analysis*, 22:604–616, 1985. @pages 214

R. Krawczyk and A. Neumaier. An Improved Interval Newton Operator. *Journal of Mathematical Analysis and Applications*, 118:194–207, 1986. @pages 215

R. Krawczyk and A. Neumaier. Interval Newton Operators for Function Strips. *Journal of Mathematical Analysis and Applications*, 124:52–72, 1987. @pages 65

N. Labroche, N. Monmarché, and G. Venturini. A New Clustering Algorithm Based on the Chemical Recognition System of Ants. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI'2002)*, pages 345–349, France, July 2002. IOS Press. @pages 190, 192

A. H. Land and A. G. Doig. An Automated Method for Solving Discrete Programming Methods. *Econometrica*, 28:497–520, 1960. @pages 10, 79

T. Larsson and T. Akenine-Moller. Collection Detection for Continuously Deforming Bodies. In *Eurographics'2001*, Manchester, UK, 2001. @pages 190

Y. Lebbah and O. Lhomme. Accelerating Filtering Techniques for Numeric CSPs. *Artificial Intelligence*, 139(1):109–132, July 2002. @pages 187

Y. Lebbah, C. Michel, and M. Rueher. Global Filtering Algorithms Based on Linear Relaxations. In *Notes of the 2nd International Workshop on Global Constrained Optimization and Constraint Satisfaction (COCOS 2003)*, Switzerland, November 2003a. @pages 91, 169, 185, 186, 208

Y. Lebbah, M. Rueher, and C. Michel. A Global Filtering Algorithm for Handling Systems of Quadratic Equations and Inequations. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'2003)*, volume LNCS 2470, pages 109–123. Springer, 2003b. @pages 91, 169, 185, 186, 208

O. Lhomme. Consistency Techniques for Numeric CSPs. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 232–238, 1993. @pages 49, 50, 88, 169

O. Lhomme, A. Gotlieb, and M. Rueher. Dynamic Optimization of Interval Narrowing Algorithms. *Journal of Logic Programming*, 37(1-2):165–183, 1998. @pages 89

D. C. Little, K. C. Murty, D. W. Sweeney, and C. Karel. An Algorithm for the Travelling Salesman Problem. *Operation Research*, 11:972–989, 1963. @pages 10, 79

C. Lottaz. Rewriting Numeric CSPs for Consistency Algorithms. In *Workshop on Non-Binary Constraints at the International Joint Conference on Artificial Intelligence (IJCAI)*, pages E:1–E:13, Stockholm, Sweden, August 1999. @pages 30, 54

C. Lottaz. *Collaborative Design using Solution Spaces*. PhD thesis, Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland, 2000. @pages 30, 54, 78, 124, 127, 207

A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977. @pages 44, 45, 81, 83, 86

K. Marriott and P. J. Stuckey. *Programming With Constraints: An Introduction*. The MIT Press, 1998. @pages 9, 13, 14, 45

R. Martin, H. Shou, I. Voiculescu, A. Bowyer, and G. Wang. Comparison of Interval Methods for Plotting Algebraic Curves. *Computer Aided Geometric Design*, 19(7):553–587, 2002. @pages 37

D. W. Matula and P. Kornerup. Finite Precision Rational Arithmetic: Slash Number System. *IEEE Transactions on Computers*, 34(1):3–18, 1985. @pages 31

G. Mayer. Enclosing the Solution Set of Linear Systems with Inaccurate Data by Iterative Methods Based on Incomplete LU-Decomposition. *Computing*, 35:189–206, 1985. @pages 66

G. Mayer. Comparison Theorems for an Iterative Method Based on Strong Splittings. *SIAM Journal of Numerical Analysis*, 24:215–227, 1987. @pages 66

G. Mayer. Enclosing the Solutions of Systems of Linear Equations for Interval Iterative Processes. *Computing Suppl.*, 6:47–58, 1988. @pages 66

G. Mayer. Epsilon-inflation in verification algorithms. *Journal of Computational and Applied Mathematics*, 60(1-2):147–169, June 1995. @pages 75

G. Mayer. Epsilon-Inflation with Contractive Interval Functions. *Applications of Mathematics*, 43(4):241–254, 1998. @pages 75

D. A. McAllester. Partial Order Backtracking. Research note, Artificial Intelligence Laboratory, MIT, USA, 1993. @pages 79

G. P. McCormick. Computability of Global Solutions to Factorable Nonconvex Programs: Part I – Convex Underestimating Problems. *Mathematical Programming*, 10:147–175, 1976. @pages 27, 81, 91

G. P. McCormick. *Nonlinear Programming: Theory, Algorithms and Applications.* John Wiley & Sons, 1983. @pages 27, 81, 91

J. J. McGregor. Relational Consistency Algorithms and Their Application in Finding Subgraph and Graph Isomorphisms. *Information Science*, 19:229–250, 1979. @pages 79

J. M. McNamee. A Bibliography on Roots of Polynomials. *Journal of Computational and Applied Mathematics*, 47(3):391–394, September 1993. @pages 55

J. M. McNamee. A Supplementary Bibliography on Roots of Polynomials. *Journal of Computational and Applied Mathematics*, 78:1, 1997. @pages 55

J. M. McNamee. An Updated Supplementary Bibliography on Roots of Polynomials. *Journal of Computational and Applied Mathematics*, 110(2):305–306, October 1999. @pages 55

J. M. McNamee. A 2002 Update of the Supplementary Bibliography on Roots of Polynomials. *Journal of Computational and Applied Mathematics*, 142(2):433–434, May 2002. @pages 55, 56

J. M. McNamee. A 2003 Update of The Supplementary Bibliography on Roots of Polynomials. *Journal of Computational and Applied Mathematics*, 2003. @pages 55

O. Mencer. *Rational Arithmetic Units in Computer System.* PhD thesis, Stanford University, California, USA, February 2000. @pages 31

P. Meseguer. Interleaved Depth-First Search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1382–1387, 1997. @pages 78

F. Messine. New Affine Forms in Interval Branch and Bound Algorithms. Technical Report R2I 99-02, Université de Pau et des Pays de l'Adour (UPPA), France, October 1999. @pages 41

F. Messine. Extensions of Affine Arithmetic in Interval Global Optimization Algorithms. In *SCAN 2000 and INTERVAL 2000 - IMACS/GAMM International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, Germany, 2000. @pages 41

F. Messine. Extentions of Affine Arithmetic: Application to Unconstrained Global Optimization. *Journal of Universal Computer Science*, 8(11):992–1015, November 2002. @pages 37, 41, 186, 208

C. Michel, Y. Lebbah, and M. Rueher. Safe Embedding of the Simplex Algorithm in a CSP Framework. In *Proceeding of the 5th International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems (CPAIOR 2003)*, pages 210–220, Canada, 2003. @pages 90

W. Miller. Quadratic Convergence in Interval Arithmetic, Part 2. *BIT Numerical Mathematics*, 12:291–298, 1972. @pages 66

W. Miller. More on Quadratic Convergence in Interval Arithmetic. *BIT Numerical Mathematics*, 13(76–83), 1973. @pages 66

S. Minton, M. D. Johnson, A. Philips, and P. Laird. Minimizing Conflicts: A Heuristics Repair Methods for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence*, 58: 161–205, 1992. @pages 100

S. Miyajima. On the Improvement of the Division of the Affine Arithmetic. Bachelor thesis, Kashiwagi Laboratory, Waseda University, Japan, 2000. It is in Japanese, but easy to guess. @pages 38

S. Miyajima, T. Miyata, and M. Kashiwagi. A New Dividing Method in Affine Arithmetic. *IEICE Transaction on Fundamentals of Electronics, Communications and Computer Sciences*, E86-A(9):2192–2196, September 2003. @pages 39, 137, 186, 208

R. Mohr and G. Masini. Good Old Discrete Relaxation. In *Proceedings of the 8th Euro Conference on Artificial Intelligence (ECAI-88)*, pages 651–656, 1988. @pages 45

E. Monfroy and J.-H. Réty. Chaotic Iteration for Distributed Constraint Propagation. In *Proceedings of ACM Symposium on Applied Computing, Artificial Intelligence and Computational Logic Track*, pages 19–24, Texas, USA, March 1999. @pages 89

U. Montanari. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Science*, 7:95–132, 1974. @pages 44, 45, 46

R. E. Moore. Automatic Error Analysis in Digital Computation. Technical Report LMSD-84821, Missiles and Space Division, Lockheed Aircraft Corporation, Sunnyvale, California, USA, 1959. @pages 209

R. E. Moore. *Interval Arithmetic and Automatic Error Analysis in Digital Computing.* PhD thesis, Stanford University, USA, October 1962. @pages 209

R. E. Moore. *Interval Analysis.* Prentice Hall, Englewood Cliffs, NJ, 1966. @pages 32, 57, 63, 209

R. E. Moore. A Test for Existence of Solutions to Nonlinear Systems. *SIAM Journal of Numerical Analysis*, 14(4):611–615, 1977. @pages 61

R. E. Moore. *Methods and Applications of Interval Analysis.* SIAM Studies in Applied Mathematics. Philadelphia, 1979. ISBN 0-89871-161-4. @pages 32, 68

R. E. Moore and L. Qi. A Successive Interval Test for Nonlinear Systems. *SIAM Journal of Numerical Analysis*, 19(4):845–850, 1982. @pages 62

H. Muñoz and R. B. Kearfott. Slope Intervals, Generalized Gradients, Semigradients, Slant Derivatives, and Csets. *Reliable Computing*, 10(3):163–193, 2004. @pages 216

Y. Nakaya and S. Oishi. Finding All Solutions of Nonlinear Systems of Equations Using Linear Programming with Guaranteed Accuracy. *Journal of Universal Computer Science*, 4(2): 171–177, 1998. @pages 95

A. Neumaier. New Techniques for the Analysis of Linear Interval Equations. *Linear Algebra Applications*, 58:273–325, 1984. @pages 60

A. Neumaier. Interval Iteration for Zeros of Systems of Equations. *BIT Numerical Mathematics*, 25(1):256–273, 1985. @pages 64

A. Neumaier. Existence of Solutions of Piecewise Differentiable Systems of Equations. *Arch. Math.*, 47:443–447, 1986. @pages 64

A. Neumaier. Further Results on Linear Interval Equations. *Linear Algebra Applications*, 87: 155–179, 1987a. @pages 66

A. Neumaier. Overestimation in Linear Interval Equation. *SIAM Journal of Numerical Analysis*, 24:207–214, 1987b. @pages 58

A. Neumaier. *Interval Methods for Systems of Equations.* Cambridge Univ. Press, Cambridge, 1990. @pages 27, 32, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 74, 81, 97, 211, 212, 213, 214, 215, 216

A. Neumaier. A Simple Derivation of the Hansen-Bliek-Rohn-Ning-Kearfott Enclosure for Interval Linear Equations. *Reliable Computing*, 5(2):131–136, 1999. @pages 66

A. Neumaier. Erratum to: A Simple Derivation of the Hansen-Bliek-Rohn-Ning-Kearfott Enclosure for Interval Linear Equations. *Reliable Computing*, 6(2):227, 2000. @pages 66

A. Neumaier. Taylor Forms - Use and Limits. *Reliable Computing*, 9:43–79, 2002. @pages 23, 215

A. Neumaier. Complete Search in Continuous Global Optimization and Constraint Satisfaction. *Acta Numerica*, 2004:271–369, 2004. @pages 3, 4, 21, 23, 68, 79, 100

A. Neumaier and J.-P. Merlet. Constraint Satisfaction and Global Optimization in Robotics. `http://www.mat.univie.ac.at/~neum/ms/robslides.pdf`, 2002. @pages 101

A. Neumaier and O. Shcherbina. Safe Bounds in Linear and Mixed-Integer Programming. *Mathematical Programming A*, 99:283–296, 2004. @pages 90, 182

K. Nickel. On the Newton Method in Interval Analysis. MRC Technical Summary Report #1136, University of Wisconsin – Madison, USA, 1971. @pages 63

K. Nickel. A Globally Convergent Ball Newton Method. *SIAM Journal of Numerical Analysis*, 18(6):988–1003, December 1981. @pages 63

S. Ning and R. Kearfott. A Comparison of Some Methods for Solving Linear Interval Equations. *SIAM Journal of Numerical Analysis*, 34:1289–1305, 1997. @pages 66

J. Nocedal and S. J. Wright. *Numerical Optimization.* Springer Series in Operations Research. Springer, 2000. @pages 4

J. M. Ortega and W. C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables.* Academic Press, New York, 1970. @pages 74, 76

J. M. Ortega and W. C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables*, volume 30 of *Classics in Applied Mathematics.* Soc for Industrial & Applied Math, reprint edition, March 2000. @pages 74

W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C, The Art of Scientific Computing.* Cambridge University Press, New York, second edition, 1992. @pages 92, 93

P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268–299, 1993. @pages 79

P. Prosser, K. Stergiou, and T. Walsh. Singleton Consistencies. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'2000)*, volume LNCS 1894, pages 353–368, Singapore, September 2000. Springer-Verlag. @pages 54

F. N. Ris. *Interval Analysis and Applications to Linear Algebra*. PhD thesis, University of Oxford, 1972. @pages 58, 60

J. Rohn. Cheap and Tight Bounds: The Recent Result of E. Hansen Can Be Made More Efficient. *Interval Computation*, 4:13–21, 1993. @pages 66

J. Rohn and R. Georg. Enclosing Solutions of Linear Equations. *SIAM Journal of Numerical Analysis*, 35(2):524–539, April 1998. @pages 75

S. M. Rump. *Kleine Fehlerschranken bei Matrixproblemen*. PhD thesis, Universitäte Karlsruhe, 1980. @pages 75

S. M. Rump. Solution of Linear and Nonlinear algebraic Problems with Sharp, Guaranteed Bounds. *Computing Suppl.*, 5:147–168, 1984. @pages 58

S. M. Rump. On the Solution of Interval Linear Systems. *Computing*, 47:337–352, 1992. @pages 75

S. M. Rump. Verification Methods for Dense and Sparse Systems of Equations. In *Topics in Validated Computations*, Studies in Computational Mathematics, pages 63–136. Elsevier Science Publishers, Amsterdam, 1994. @pages 75

S. M. Rump. A Note on Epsilon-Inflation. *Reliable Computing*, 4(4):371–375, 1998. @pages 75

S. M. Rump and E. Kaucher. Small Bounds for the Solution of Systems of Linear Equations. *Computing Suppl.*, 2:157–164, 1980. @pages 58

D. Sabin and E. C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI-94)*, pages 125–129, 1994. @pages 79

D. Sam-Haroud. *Constraint Consistency Techniques for Continuous Domains*. PhD thesis, Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland, 1995. @pages 30, 44, 54, 78, 102, 127, 207

D. Sam-Haroud and B. Faltings. Consistency Techniques for Continuous Constraints. *Constraints*, 1:85–118, September 1996. @pages 30, 54, 102, 107, 124, 127, 207

H. Schichl. *Mathematical Modeling and Global Optimization*. Habilitation thesis, Faculty of Mathematics, University of Vienna, Autralia, November 2003. @pages 153, 154

H. Schichl and A. Neumaier. Exclusion Regions for Systems of Equations. *SIAM Journal on Numerical Analysis*, 42:383–408, 2004a. @pages 61, 68, 72, 73, 74

H. Schichl and A. Neumaier. Interval Analysis on Directed Acyclic Graphs for Global Optimization. *Journal of Global Optimization*, 2004b. To appear. @pages 151, 152, 153, 155, 168, 187, 206, 208

H. Schwandt. Iterative Methods for Systems of Equations with Interval Coefficients and Linear Form. *Computing*, 38:143–161, 1987. @pages 66

B. Selman and H. A. Kautz. Domain-Independent Extentions to GSAT: Solving Large Structured Satisfiability Problems. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93)*, France, 1993. @pages 100

Z. Shen and A. Neumaier. The Krawczyk Operator and Kantorovich's theorem. *Journal of Mathematical Analysis and Applications*, 149:437–443, 1990. @pages 74

M.-C. Silaghi. *Asynchronously Solving Distributed Problems with Privacy Requirements.* PhD thesis, Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland, 2002. @pages 14, 119

M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Search Techniques for Non-linear CSPs with Inequalities. In *Proceedings of the 14th Canadian Conference on Artificial Intelligence*, 2001. @pages 102, 107, 109, 110, 119

S. Singh, B. Watson, and P. Srivastava. *Fixed Point Theory and Best Approximation: The KKM-map Principle.* Kluwer Academic Publishers, Dordrecht, 1997. @pages 130, 217, 218, 219, 220, 221

J. M. Snyder. *Generative Modeling for Computer Graphics and CAD.* Academic Press, Inc., London, UK, 1992. @pages 101

P. Spiteri. A New Characterization of M-Matrices and H-Matrices. *BIT Numerical Mathematics*, 43(5):1019–1032, January 2003. @pages 212

J. Stolfi and L. H. de Figueiredo. Self-Validated Numerical Methods and Applications. In *Monograph for 21st Brazilian Mathematics Colloquium (IMPA)*, Brazil, July 1997. @pages 37, 38, 137

D. Struik. *A Concise History of Mathematics.* Dover Publications, 1948. @pages 9

T. Sunaga. Theory of an Interval Algebra and its Applications to Numerical Analysis. *RAAG Memoirs*, 2:29–46, 1958. @pages 209

G. Sussman and G. Steele. CONSTRAINTS – A Language for Expressing Almost-Hierarchical Descriptions. *Artificial Intelligence*, 14(1):1–39, 1980. @pages 9

I. Sutherland. SKETCHPAD: A Man-Machine Graphical Communication System. In *Proceedings of the Spring Joint Computer Conference*, pages 329–346, 1963. @pages 9

E. E. Swartzlander and G. A. Alexopoulos. The Sign/Logarithm Number System. *IEEE Transactions on Computers*, 24(12):1238–1242, 1975. @pages 31

M. Tawarmalani and N. V. Sahinidis. *Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming.* Nonconvex Optimization and Its Applications. Kluwer, September 2002. @pages 99, 187

S. Thiel. A Generalization of the Interval Newton Single Step Method for Nonlinear Systems of Equations. *Computing*, 43:73–84, 1989. @pages 63

M. Torrens Arnal. *Scalable Intelligent Electronic Catalogs.* PhD thesis, Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland, 2003. @pages 78, 79, 100

E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993. @pages 14, 21, 22, 23, 43, 79, 81, 83, 86, 100, 192

P. van Beek. On the Minimality and Decomposibility of Constraint Networks. In *Proceedings of the 10th National Conference on AI*, 1992. @pages 44

P. van Beek and R. Dechter. On the Minimality and Global Consistency of Row-Convex Constraint Networks. *Journal of the ACM*, 42(3):543–561, 1995. @pages 44

G. van den Bergen. Efficient Collision Detection of Complex Deformable Models using AABB Trees. *Journal of Graphics Tools*, 4(2):7–25, 1997. @pages 190

P. Van Hentenryck. Numerica: A Modeling Language for Global Optimization. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, 1997. @pages 151

P. Van Hentenryck. A Gentle Introduction to NUMERICA. *Artificial Intelligence*, 103(1-2): 209–235, 1998. @pages 101

P. Van Hentenryck, D. McAllester, and D. Kapur. Solving Polynomial Systems Using a Branch and Prune Approach. *SIAM Journal of Numerical Analysis*, 34(2):797–827, 1997a. @pages 88

P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica: A Modeling Language for Global Optimization.* MIT press, 1997b. @pages 36, 68, 88

J. I. van Hermet. *Application of Evolutionary Computation to Constraint Satisfaction and Data Mining.* PhD thesis, University of Leiden, Netherland, November 2002. @pages 100

R. J. Van Iwaarden. *An Improved Unconstrainted Global Optimization Algorithm.* PhD thesis, University of Colorado at Denver, USA, 1996. @pages 114

X.-H. Vu, D. Sam-Haroud, and B. Faltings. Clustering for Disconnected Solution Sets of Numerical CSPs. In *Recent Advances in Constraints: International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2003*, volume LNAI 3010, pages 25–43, Budapest, Hungary, July 2004a. Springer-Verlag. @pages 189

X.-H. Vu, D. Sam-Haroud, and B. Faltings. Combining Multiple Inclusion Representations in Numerical Constraint Propagation. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, pages 458–467, Florida, USA, November 2004b. IEEE Computer Society Press. @pages 129, 135, 169

X.-H. Vu, D. Sam-Haroud, and M.-C. Silaghi. Approximation Techniques for Non-linear Problems with Continuum of Solutions. In *Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation (SARA 2002)*, volume LNAI 2371, pages 224–241, Alberta, Canada, August 2002. Springer-Verlag. @pages 101

X.-H. Vu, D. Sam-Haroud, and M.-C. Silaghi. Numerical Constraint Satisfaction Problems with Non-isolated Solutions. In *Global Optimization and Constraint Satisfaction: First International Workshop on Global Constraint Optimization and Constraint Satisfaction, COCOS 2002*, volume LNCS 2861, pages 194–210, Valbonne-Sophia Antipolis, France, October 2003. Springer-Verlag. @pages 101, 202

X.-H. Vu, H. Schichl, and D. Sam-Haroud. Using Directed Acyclic Graphs to Coordinate Propagation and Search for Numerical Constraint Satisfaction Problems. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, pages 72–81, Florida, USA, November 2004c. IEEE Computer Society Press. @pages 129, 151

T. Walsh. Depth-bounded Discrepancy Search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, 1997. @pages 78

G. W. Walster, E. R. Hansen, and J. D. Pryce. Extended Real Intervals and the Topological Closure of Extended Real Relations. Technical report, Sun Microsystems, February 2000. `http://wwws.sun.com/software/sundev/whitepapers/extended-real.pdf`. @pages 133

D. L. Waltz. Generating Semantic Descriptions from Drawings of Scenes with Shadows. Technical report, Massachusetts Institute of Technology, USA, 1972. @pages 44

D. L. Waltz. *The Psychology of Computer Vision*, chapter Understanding Line Drawings of Scenes with Shadows, pages 19–91. McGraw Hill, New York, 1975. @pages 44

M. Warmus. Caculus of Approximations. *Bulletin de l'Académie Polonaise des Sciences*, 9(5): 253–259, 1956. @pages 209

M. Warmus. Approximations and Inequalities in the Calculus of Approximations. Classification of Approximate Numbers. *Bulletin de l'Académie Polonaise des Sciences – Série de Sci. Math. Astronom. Phys.*, 9(4):241–245, 1961. @pages 209

N. Wiener. A Contribution to the Theory of Relative Position. In *Proceedings of Cambridge Philosophy Society*, volume 17, pages 441–449, 1914. @pages 209

N. Wiener. A New Theory of Measurements: A Study in the Topic of Mathematics. In *Proceedings of the London Mathematical Society*, volume 19, pages 181–205, 1921. @pages 209

J. H. Wilkinson. *The Algebraic Eigenvalue Problem.* Oxford University Press, London, UK, 1965. @pages 65

Z.-B. Xu, J.-S. Zhang, and Y.-W. Leung. A general CDC Formulation for Specializing the Cell Exclusion Algorithms of Finding All Zeros of Vector Functions. *Applied Mathematics and Computation*, 86(2-3):235–259, October 1997. @pages 70

Z.-B. Xu, J.-S. Zhang, and W. Wang. A Cell Exclusion Algorithm for Determining All the Solutions of a Nonlinear System of Equations. *Applied Mathematics and Computation*, 80 (2-3):181–208, December 1996. @pages 69

K. Yamamura. An Algorithm for Representing Functions of Several Variables by Superposition of Functions of One Variable and Addition. In *Proceedings of the 1993 International Symposium of Nonlinear Theory and its Applications*, pages 1045–1048, December 1993. @pages 29, 96

K. Yamamura. An Algorithm for Representing Functions of Many Variables by Superpositions of Functions of One Variable and Addition. *IEEE Transaction on Circuits and Systems*, I-43 (3):338–340, 1996. @pages 29, 96

K. Yamamura. Finding All Solutions of Nonlinear Equations Using Linear Combinations of Functions. *Reliable Computing*, 6(2):105–113, 2000. @pages 93

K. Yamamura. Finding All Solution Sets of Piecewise-Trapezoidal Equations Described by Set-Valued Functions. *Reliable Computing*, 9(3):241–250, June 2003. @pages 93

K. Yamamura and T. Fujioka. Finding All Solutions of Nonlinear Equations Using the Dual Simplex Method. *Journal of Computational and Applied Mathematics*, 152(1-2):587–595, April 2003. @pages 92, 93

K. Yamamura and Y. Hata. Finding All Solutions of Weakly Nonlinear Equations Using Linear Programming. *IEICE Trans.on Fundamentals of Electronics, Communications and Computer Sciences*, E83-A(12):2758–2761, December 2000. @pages 95

K. Yamamura, H. Kawata, and A. Tokue. Interval Solution of Nonlinear Equations Using Linear Programming. *BIT Numerical Mathematics*, 38(1):186–199, 1998. @pages 95

K. Yamamura and T. Kumakura. Finding All Characteristic Curves of Nonlinear Resistive Circuits Using the Dual Simplex Method. In *Proceedings of the IEEE 2001 International Symposium on Circuits and Systems (ISCAS 2001)*, volume 3, pages 25–28, May 2001. @pages 95

K. Yamamura and M. Nishizawa. Finding All Solutions of A Class of Nonlinear Equations Using an Improved LP Test. *Japan Journal of Industrial and Applied Mathematics*, 16(3): 349–368, 1999. @pages 95

K. Yamamura and S. Tanaka. Finding All Solutions of Systems of Nonlinear Equations Using the Dual Simplex Method. *BIT Numerical Mathematics*, 42(1):214–230, March 2002. @pages 92, 93, 95

R. C. Young. The Algebra of Multi-Valued Quantities. *Mathematische Annalen*, 104:260–290, 1931. @pages 209

W. H. Young. Sull due funzioni a piu valori constituite dai limiti d'una funzione di variable reale a destra ed a sinistra di ciascun punto. *Rendiconti Academia di Lincei, Classes di Scienza Fiziche*, 17(5):582–587, 1908. @pages 209

Y. Zhou and S. Suri. Analysis of a Bounding Box Heuristic for Object Intersection. In *Proceedings of the 10th Annual Symposium on Discrete Algorithms (SODA'99)*, 1999. @pages 190

Glossary

Symbol	Meaning	Page
$(\langle x_1, a_1\rangle, \ldots, \langle x_k, a_k\rangle)$	the instantiation of values $(a_1, \ldots, a_k)$ to variables $(x_1, \ldots, x_k)$.	11
$\mathbb{C}$	the set of all complex numbers.	212
$\mathbb{F}$	the set of all floating-point numbers, except infinities.	34
$\mathbb{I}_\diamond$	the set of all floating-point intervals.	34
$\mathbb{I}_\circ$	the set of all (real) intervals.	33
$\mathbb{I}$	the set of all closed (real) intervals.	33
$\mathbb{N}$	the set of all natural numbers: $\{0, 1, 2, \ldots\}$.	11
$\mathbb{Q}$	the set of all rational numbers.	218
$\mathbb{R}$	the set of all real numbers.	15
$\mathbb{R}_+$	the set of all positive real numbers.	115
$\mathbb{Z}$	the set of all integers: $\{\ldots, -2, -1, 0, -1, 2, \ldots\}$.	13
$\mathbb{Z}_-$	the set of all negative integers: $\{-1, -2, -3, \ldots\}$.	13
$\mathbb{Z}_+$	the set of all positive integers: $\{1, 2, 3, \ldots\}$.	13
I	the identity matrix of suitable size.	55
$(\mathcal{V}, \mathcal{D}, \mathcal{C})$	the notation of a constraint satisfaction problem.	12
$\langle \mathcal{C};\ \mathcal{V} \in \mathcal{D}\rangle$	the notation of a constraint satisfaction problem.	12
X^{T}	the transpose of a vector/matrix X.	25
$\square^{\mathbb{F}} S$	the $\mathbb{F}$-hull (or floating-point interval hull) of a real set $S \subseteq \mathbb{R}^n$.	34
$\square S$	the (interval) hull of a real set $S \subseteq \mathbb{R}^n$.	33
$\mathrm{int}(S)$	the interior of a set S.	61
$\Phi_{k\mathrm{B}}$	the domain reduction operator achieving kB-consistency.	50
$\Phi_{k\mathrm{B}}^{\mathbb{F}}$	the domain reduction operator achieving $k\mathrm{B}(\mathbb{F})$-consistency.	51
$\Sigma(\mathbf{A}, \mathbf{b})$	the solution set of the linear interval equation (A.15): $\mathbf{A}\mathbf{x} = \mathbf{b}$.	213
$[-\infty, +\infty]$	the universal interval, the same as $]-\infty, +\infty[$.	33
$[-\infty, x]$	the same as $]-\infty, x]$.	32
$[x, +\infty]$	the same as $[x, +\infty[$.	32
$\langle \beta_1, \beta_2\rangle \in \mathbb{I}_\circ$	the notation of a general interval in $\mathbb{I}_\circ$.	36
$\mathbf{C}_X$	the conjunction constraint on a sequence X of variables.	43
$C[X]$	the projection of a constraint C on a subsequence X of variables.	15
$d[X]$	the projection of a compound label d on a subsequence X of variables.	15
$(A_{ij})_{m\times n}$	the notation of a matrix of size $m \times n$ with components A_{ij}, where $1 \le i \le m$ and $1 \le j \le n$.	57
$\langle \mathbf{A}\rangle$	the comparison matrix of an interval matrix $\mathbf{A}$.	212
$\mathbf{A}^{-1}$	the inverse of an interval matrix $\mathbf{A}$, see Definition A.18.	214

A^{-1}	the inverse of a matrix A, see also Definition A.7.	212
$\rho(A)$	the spectral radius of a matrix $A \in \mathbb{R}^{n\times n}$.	212
$\boxplus^{\mathcal{I}}[S]$	an inner union approximation of a set S.	106
$\boxplus^{\mathcal{B}}[S]$	a boundary union approximation of a set S.	106
$\boxplus^{\mathcal{O}}[S]$	an outer union approximation of a set S.	106
$\delta(A)$	the diameter of a set, A, in a metric space.	217
$\delta_{\mathrm{E}}(X,Y)$	the Euclidean distance between two sets, X and Y, in a normed space.	219
$\delta_{\mathrm{H}}(X,Y)$	the Hausdorff distance between two sets, X and Y, in a normed space.	219
$\lfloor E \rfloor$	a downward-rounded evaluation of an expression, E, such that $E \geq \lfloor E \rfloor$.	139
$\lceil E \rceil$	an upward-rounded evaluation of an expression, E, such that $E \leq \lceil E \rceil$.	139
$f[.,.]$	the notation of a (first order) slope function.	215
$f[.,.,.]$	the notation of a second order slope function.	215
$F^{-1}(Y)$	the inverse image of a set Y under a function/multifunction F.	130
$\\|x\\|_\infty$	the ℓ_∞-norm of a vector $x \in \mathbb{R}^n$, also called the *maximum norm*: $\\|x\\|_\infty \equiv \max\{\lvert x_i\rvert \mid i = 1,\dots,n\}$.	66
$\\|A\\|_\infty$	the ℓ_∞-norm of a matrix $A \in \mathbb{R}^{m\times n}$, also called the *row sum norm*: $\\|A\\|_\infty \equiv \max\{\sum_{j=1}^n \lvert A_{ij}\rvert \mid i = 1,\dots,m\}$.	74
$\\|x\\|_k$	the ℓ_k-norm of a vector $x \in \mathbb{R}^n$: $\\|x\\|_k \equiv (\sum_{i=1}^n \lvert x_i\rvert^k)^{1/k}$, where $k \in \mathbb{Z}_+$.	70
$\\|A\\|_k$	the ℓ_k-norm of a matrix $A \in \mathbb{R}^{m\times n}$: $\\|A\\|_k \equiv (\sum_{i=1}^m (\sum_{j=1}^n \lvert A_{ij}\rvert)^k)^{1/k}$, where $k \in \mathbb{Z}_+$.	74
$\\|x\\|_u$	the scaled maximum norm of a vector $x \in \mathbb{R}^n$, where $0 < u \in \mathbb{R}^n$, i.e., $\\|x\\|_u \equiv \max\{\lvert x_i\rvert/u_i \mid i = 1,\dots,n\}$.	58
$\\|A\\|_u$	the scaled maximum norm of a matrix $A \in \mathbb{R}^{n\times n}$, where $0 < u \in \mathbb{R}^n$, i.e., $\\|A\\|_u \equiv \\|\ \lvert A\rvert u\ \\|_u = \max\{\sum_{j=1}^n \lvert A_{ij}\rvert u_j/u_i \mid i = 1,\dots,n\}$.	58
$\mathfrak{O}_{\mathrm{GS}}(\mathbf{A},\mathbf{b},\mathbf{x})$	the Gauss-Seidel operator for the linear equations (3.10), i.e., $\mathbf{Ax} = \mathbf{b}$.	59
$\mathfrak{O}_{\mathrm{HS}}(f,\mathbf{A},C,\mathbf{x},c)$	the Hansen-Sengupta operator for the equation $f(x) = 0$.	62
$\mathfrak{O}_{\mathrm{KL}}(f,\mathbf{x})$	the Kolev operator for the equation f(x) = 0.	98
$\mathfrak{O}_{\mathrm{K}}(\mathbf{A},\mathbf{b},C,\mathbf{x})$	the Krawczyk operator for the linear equations (3.10), i.e., $\mathbf{Ax} = \mathbf{b}$.	58
$\mathfrak{O}_{\mathrm{K}}(f,\mathbf{A},C,\mathbf{x},c)$	the Krawczyk operator for the equation $f(x) = 0$.	60
$\prec_k$	the domination relation of order k.	70
$\prec\prec$	the domination relation of order ∞.	70
$C\rvert_{x_i=p}$	the section of a constraint C at $x_i = p$.	15
$\mathcal{C}\rvert_{x_i=p}$	the section of a set $\mathcal{C}$ of constraints at $x_i = p$.	15
$\mathcal{P}\rvert_{x=a}$	the section of a CSP $\mathcal{P}$ at $x = a$.	16
$C\rvert^*_{x_i=p}$	the cross section of a constraint C at $x_i = p$.	15
$\mathcal{C}\rvert^*_{x_i=p}$	the cross section of a set $\mathcal{C}$ of constraints at $x_i = p$.	15
$\mathcal{P}\rvert^*_{x=a}$	the cross section of a CSP $\mathcal{P}$ at $x = a$.	16
$C\rvert_{x_i\in S}$	the slice of a constraint C in the slot $x_i \in S$.	16
$\mathcal{C}\rvert_{x_i\in S}$	the slice of a set $\mathcal{C}$ of constraints in the slot $x_i \in S$.	16
$\mathcal{P}\rvert_{x\in S}$	the slice of a CSP $\mathcal{P}$ in the slot $x \in S$.	17
$\hat{x}$	the notation of an affine form.	37
$\tilde{x}$	the notation of a Kolev affine form.	39
$\breve{x}$	the notation of a Messine affine form.	41
$\widehat{x}$	the notation of a revised affine form.	134

$\mathbf{x} \equiv [\underline{x}, \overline{x}]$	the notation of a closed interval.	32
$\mathbf{x} \equiv]\underline{x}, \overline{x}[$	the notation of an open interval.	32
$\mathbf{x} \equiv]\underline{x}, \overline{x}]$	the notation of a left-open interval.	32
$\mathbf{x} \equiv [\underline{x}, \overline{x}[$	the notation of a right-open interval.	32
$\mathrm{mid}(\mathbf{x})$	the (componentwise) midpoint of an interval (vector/matrix) $\mathbf{x}$.	33
$\mathrm{rad}(\mathbf{x})$	the (componentwise) radius of an interval (vector/matrix) $\mathbf{x}$.	33
$\mathrm{w}(\mathbf{x})$	the (componentwise) width of an interval (vector/matrix) $\mathbf{x}$.	33
$\mathrm{vol}(\mathbf{x})$	the volume of a box $\mathbf{x} \in \mathbb{I}^n_\circ$.	64
$(x_1, \ldots, x_k)$; $(x_i)_{i=1}^k$	the notation of a sequence (or a tuple).	11
$\{x_1, \ldots, x_k\}$; $\{x_i\}_{i=1}^k$	the notation of a set.	10
$x^{\langle k \rangle}$	the notation of super index, like the usual index x_k.	56
$\mathrm{pts}(\mathcal{R})$	the set of points of the subset $S \subseteq \mathbb{R}^n$ that is exactly represented by $\mathcal{R}$.	105
$\lfloor Z \rfloor$	a number $\lfloor Z \rfloor \in \mathbb{F}$ as close to Z as possible such that $Z \geq \lfloor Z \rfloor$.	34
$\lceil Z \rceil$	a number $\lceil Z \rceil \in \mathbb{F}$ as close to Z as possible such that $Z \leq \lceil Z \rceil$.	34
$\langle Z \rangle \pm z$	a floating-point number $\langle Z \rangle \in \mathbb{F}$ as close to Z as possible such that $\langle Z \rangle - z \leq Z \leq \langle Z \rangle + z$, where $z \in \mathbb{F}$.	42
AI	artificial intelligence.	9
BFS	breadth-first search.	78
BJ	backjumping.	79
BM	backmarking.	78
BM-BJ	backmarking with backjumping.	79
BM-CBJ	backmarking with conflict-directed backjumping.	79
BT	backtracking, chronological backtracking, simple backtracking.	78
CBJ	conflict-directed backjumping.	79
CIRD	a scheme for combining inclusion representations.	178
CIRD[ai]	an algorithm instance of the **CIRD** scheme using affine arithmetic, interval arithmetic, interval constraint propagation, and linear programming.	183
Colonization	the colonization algorithm.	192
CSP	constraint satisfaction problem.	12
DAG	directed acyclic graph.	153
DB	dynamic backtracking.	79
DBR	disjoint box representation.	116
DDS	depth-bounded discrepancy search.	78
DFS	depth-first search.	78
DMBC	dichotomous maintaining bounds by consistency.	102
DMBC$^+$	a revised version of dichotomous maintaining bounds by consistency.	125
EVR	extreme vertex representation.	118
FBPD	forward-backward propagation on DAGs.	160

FC	forward checking.	79
FC-BJ	forward checking with backjumping.	79
FC-CBJ	forward checking with conflict-directed backjumping.	79
FPB	flexible partial order backtracking.	79
GBJ	graph-based backjumping search.	79
GDB	general partial order backtracking.	79
IB	iterative broadening.	78
IDFS	interleaved depth-first search.	78
iff	an abbreviation of "if and only if".	54
ILDS	improved limited discrepancy search.	78
LDS	limited discrepancy search.	78
LP	linear program(ming).	3
MCC	max-connected clustering.	197
MFC	maintaining arc consistency.	79
MFC	minimal forward checking.	79
MILP	mixed integer linear program.	3
MINLP	mixed integer nonlinear program.	3
NCSP	numerical constraint satisfaction problem.	25
NLP	nonlinear program.	3
OMCC	optimal max-connected clustering.	201
OR	operation research.	9
OSDC	optimized separator-driven clustering.	202
PBT	partial chronological backtracking.	79
PDB	partial dynamic backtracking.	79
PFC	partial forward checking.	79
POB	partial order backtracking.	79
SDC	separator-driven clustering.	199
SIVIA	set inverter via interval analysis.	78
UCA5	a version of union-constructive approximation.	125
UCA6	a version of union-constructive approximation.	119
UCA6^{+}	a version of union-constructive approximation.	121

Index

Symbols

2B-consistency **50**
2B($\mathbb{F}$)-consistency **51**
2^k-tree 78, 107
 ∼ representation 54, 78
Z-extended function **130**
NaN . . . . . *see* Not a Number, **32**
$\mathbb{F}$-hull **34**
 ∼ consistency **48**, 81
$\prec\prec$ relation . . . . . *see* domination relation, **70**
$\prec_k$ relation . . . . . *see* domination relation, **70**
ε-bounded box **68**, 107
ε-inflation . . . . . *see* epsilon-inflation, **75**
k-ary CSP **12**
k-ary constraint **12**
k-compound label **11**
k-consistency **47**
 strong ∼ **47**
k-instantiation **11**
 consistent ∼ **12**
k-path consistency **46**
kB-consistency 16, **50**, **87**
kB($\mathbb{F}$)-consistency **51**, **87**
(i, j)-consistency **53**
 strong ∼ **53**

A

AABB tree **190**
abstraction 90
active variable **115**
additive identity **218**
additive invertibility **218**
affine arithmetic 5, **36**, 133
 Kolev ∼ **42**
 Kolev generalized ∼ **40**
 Messine ∼ **41**
 revised ∼ **133**
affine form **37**
 coefficient of ∼ **37**
 Kolev ∼ **39**
 length of ∼ **37**
 Messine ∼ **41**
 revised ∼ 133, **134**
affine forms
 division of ∼ **39**
 multiplication of ∼ **38**
  ≈ in Kolev affine arithmetic **42**
  ≈ in Messine affine arithmetic **41**
  ≈ in revised affine arithmetic **135**
affine function 37
affine operation **37**, 40
algorithm **9**
 asymptotical completeness of ∼ **21**
 asymptotically complete ∼ **21**
 complete ∼ **21**
 complete search ∼ **23**
 completeness of ∼ **21**
 incomplete ∼ **21**
 incompleteness of ∼ **21**
 rigor of ∼ **21**
 rigorous ∼ **21**
 solution ∼ **3**
 sound ∼ **21**
 soundness of ∼ **21**
ancestor 23, **153**
approximation
 boundary union ∼ **106**
 Chebyshev affine ∼ **137**
 complete ∼ **105**
 inner ∼ **105**
 inner union ∼ **106**
 outer ∼ **105**
 outer union ∼ **106**
 sound ∼ **105**
approximation property
 linear ∼ 68
 ∼ of order k **35**
 quadratic ∼ 60, 66, 68
arc consistency **44**
 directional ≈ **45**
arc of graph **14**
arc of multigraph **152**
arithmetic
 affine ∼ 5, **36**, 133
 ∼ constraint 9
 floating-point ∼ **31**

interval ~ 5, **32**
rational ~ 31
arithmetic expression **27**
composition of ~ **27**
arity of constraint **12**
arity of CSP **12**
artificial intelligence 9
artificial variable **92**
associativity **218**
asymptotical completeness of algorithm **21**
asymptotically complete algorithm **21**

B

backtrack-free **23**
backtracking 2, **23**
chronological ~ **78**
dependency directed ~ 78
intelligent ~ 78
simple ~ **78**
backward propagation **158**
ball
closed ~ **217**
open ~ **217**
Banach space **219**
basic feasible solution **92**
basic tableau **92**
basic variable **92**
binary constraint **13**
binary CSP **14**
strictly ~ **14**
bisection **22**
~ search 79
Boolean constraint 9
Boolean function 12
bound
lower ~ **32**
upper ~ **32**
boundary union approximation **106**
bounding-box tree **190**
bounding-volume tree **190**
box **33**
ε-bounded ~ **68**, 107
complementary ~ **110**
~ consistency **51**, 85, 86
interval ~ **33**
primitive ~ **190**
~ splitting operator **114**
undiscernible ~ **107**
box(Γ) consistency **52**, **86**
box$\langle\pm\varphi\rangle$ consistency **52**
box-tree **190**
boxes
connected ~ **190**
disjoint ~ 36, **105**
max-connected ~ **191**
branch-and-bound 2, **79**
branch-and-prune 2, 5, **79**

C

canonical representation **117**
Cartesian product **11**
Cauchy sequence **218**
`CB` operator **110**
$\texttt{CB}_{\texttt{rd}}$ operator **115**
cell subdivision **22**
centered interval form **210**
Chebyshev affine approximation **137**
Chebyshev approximation theory 38
child **153**
chord method **76**
closed ball **217**
closed interval **32**
closed set **217**
cluster effect 36, **68**
clustering 189
max-connected ~ **191**
optimal max-connected ~ **191**
coefficient of affine form **37**
color function **117**
column vector **33**
commutative ring **218**
commutativity **218**
compact CSP **17**
compact mapping **220**
compact set 17, **218**
compact tree representation **81**
comparison matrix **212**
complementariness of `CB` operator **110**
complementary box **110**
complementary boxing **110**
~ operator **110**
restricted-dimensional ≈ **115**
complete algorithm **21**
complete approximation **105**
complete graph 14
complete method 55
complete metric space **218**
complete search 5, **23**
~ algorithm **23**
~ method 23
completeness of algorithm **21**
completeness of domain reduction operator . **19**
completing the square method 1
completion of a constraint graph **46**
composition of arithmetic expression **27**
composition of factorable expression **28**

compound label **11**
  k- ∼ **11**
  redundant ∼ **20**
condition
  Karush-Kuhn-Tucker ∼ 4
  Lipschitz ∼ 69, **219**
  second order necessary optimality ∼ 4
  second order sufficient optimality ∼ 4
conjunction constraint **43**
connected boxes **190**
connectionist approach 100
consistency
  2B- ∼ **50**
  2B($\mathbb{F}$)- ∼ **51**
  $\mathbb{F}$-hull ∼ **48**, 81
  k- ∼ **47**
  k-path ∼ **46**
  kB- ∼ 16, **50**, **87**
  kB($\mathbb{F}$)- ∼ **51**, **87**
  (i, j)- ∼ **53**
  arc ∼ **44**
  arc-B ∼ . . . . *see* 2B-consistency, **49**
  box ∼ **51**, 85, 86
  box(Γ) ∼ **52**, **86**
  box$\langle \pm\varphi \rangle$ ∼ **52**
  directional arc ∼ **45**
  directional path ∼ **46**
  global ∼ 2, **44**
  hull ∼ **48**, 80, 81
  hyper-arc ∼ **45**
  interval ∼ **48**
  interval hyper-arc ∼ **48**
  local ∼ 2, 9, 23, **44**
  monotonicity of ∼ **43**
  node ∼ **44**
  path ∼ **46**
  relational $(3, 2)$- ∼ 78
  relational (n_v, n_c)- ∼ **53**
  singleton ∼ 16, **54**
  strong (i, j)- ∼ **53**
  strong k- ∼ **47**
  strong path ∼ **46**
  strongness of ∼ **43**
consistency-preserving **18**
consistency-relaxing **19**
consistent k-instantiation **12**
consistent CSP **13**
constrained optimization problem 2
constraint 2, **11**
  k-ary ∼ **12**
  arithmetic ∼ 9
  arity of ∼ **12**
  binary ∼ **13**
  Boolean ∼ 9
  conjunction ∼ **43**
  continuous ∼ **25**
  convex ∼ 30
  equality ∼ **26**
  factorable ∼ **28**
  ∼ hypergraph **14**
  implied ∼ **20**
  inequality ∼ **26**
  ∼ is not satisfied **11**
  ∼ is satisfied **11**
  ∼ is violated **11**
  logic ∼ **13**
  ∼ logic problem **13**
  numerical ∼ 3, **25**
  primitive ∼ **28**, 49
  ∼ programming 1, 4, 10
  ∼ propagation 2, 9, **23**
  ∼ range **25**, 81
  redundant ∼ **20**
  running ∼ **109**
  ∼ satisfaction 4, **10**
  ∼ satisfaction problem **12**
  satisfiability of ∼ **11**
  separable ∼ **30**
  ∼ solver 3
  ∼ solving **9**
  ternarized ∼ **30**
  ternary ∼ **13**, 78
  two-sided inequality ∼ **26**
  unary ∼ **13**
  universal ∼ **11**, 14
constraint graph **14**
  completion of a ∼ **46**
  triangulation of a ∼ **46**
constraint network . . . . *see* constraint graph, **14**
constraint programming **2**
  ∼ language 3
  ∼ technique 4
constraint propagation 2, 5
  interval ∼ 5
  numerical ∼ 5
  ∼ technique 2
constraint satisfaction 4, **10**
  co-evolutionary ∼ 100
  ∼ problem 2
constraint satisfaction problem **12**
  continuous ∼ . . . . . . *see* numerical CSP, **25**
  numerical ∼ 3, **25**
constraint solver
  interval ∼ 189
constraints
  cross section of ∼ **15**

negation of ∼ . . . **15**
section of ∼ . . . **15**
slice of ∼ . . . **16**
continuation method . . . 24, 55
continuous constraint . . . **25**
continuous CSP . . . *see* numerical CSP, **25**
continuous domain . . . 3, 10, **24**
continuous function . . . 25
continuous interval contraction . . . **57**
continuous variable . . . 3, **24**
continuum of solutions . . . 6, **102**
contraction
continuous interval ∼ . . . **57**
interval ∼ . . . **57**
P- ∼ . . . 75
strong interval ∼ . . . **57**
contraction mapping . . . **219**
contractive mapping . . . **219**
contractiveness of CB operator . . . **110**
contractiveness of domain reduction operator **19**
contractiveness of DR operator . . . **108**
contractivity factor . . . **57**
contractor . . . **57**
convex constraint . . . 30
convex CSP . . . 46
convex set . . . 71, **219**
cooperative strategy . . . 89
correctness of domain reduction operator . . . **19**
correctness of DR operator . . . **108**
cover . . . 106, **217**
open ∼ . . . **217**
cross section of constraints . . . **15**
cross section of CSP . . . **16**
CSP . . . *see* constraint satisfaction problem, **12**
k-ary ∼ . . . **12**
arity of ∼ . . . **12**
binary ∼ . . . **14**
compact ∼ . . . **17**
consistent ∼ . . . **13**
continuous ∼ . . . *see* numerical CSP, **25**
convex ∼ . . . 46
cross section of ∼ . . . **16**
factorable ∼ . . . **29**
inconsistent ∼ . . . **13**
normalized ∼ . . . **14**
numerical ∼ . . . **25**
regular ∼ . . . **14**
satisfiable ∼ . . . **13**
section of ∼ . . . **16**
separable ∼ . . . **30**
singleton ∼ . . . **16**
slice of ∼ . . . **17**
standardized ∼ . . . **14**
strictly binary ∼ . . . **14**
transformation of a ∼ . . . **18**
cycle . . . **152**

D

DAG . . . **153**
partial ∼ representation . . . **160**
∼ representation . . . 20, 82, **153**
deadend . . . **23**
decision problem . . . 13
denormalized number . . . **32**
dependency of interval arithmetic . . . **33**, 89
derivative . . . 57, 62, 138
first order ∼ test . . . 1
partial ∼ . . . 70
second order ∼ test . . . 1
descendant . . . **153**
descent method
iterative ∼ . . . 92
local ∼ . . . 100
microgenetic iterative ∼ . . . 100
diameter . . . **217**
dichotomous splitting operator . . . **113**
directed acyclic multigraph . . . **152**
directed multigraph . . . **152**
∼ with ordered edges . . . **153**
directed path . . . **152**
directed pseudograph . . . **152**
directional arc consistency . . . **45**
directional path consistency . . . **46**
discrete domain . . . 10
discretization technique . . . 4
disjoint . . . **191**
disjoint boxes . . . 36, **105**
distance
Euclidean ∼ . . . **219**
∼ function . . . **217**
Hausdorff ∼ . . . **219**
division of affine forms . . . **39**
division of intervals . . . **33**, **132**
domain . . . **10**
continuous ∼ . . . 3, 10, **24**
discrete ∼ . . . 10
heterogeneous ∼ . . . 10
∼ of variable . . . **10**
domain reduction . . . **19**, 23
∼ operator . . . **19**, **108**
restricted-dimensional ≈ . . . **115**
∼ technique . . . 20
domain-specific method . . . 2, 55
domination relation . . . **70**, 71, 95
DR operator . . . **108**
$\mathtt{DR_{rd}}$ operator . . . **115**

dual simplex method **92**

E

early quiescence 49
edge of multigraph **152**
elementary binary operation **27**
elementary operation 3, **28**
elementary unary function **27**
elimination
  Gauss ∼ 1, 9, 65
  Gauss-Jordan ∼ 2
  interval Gauss ∼ **65**
  redundant constraint ∼ **23**
  variable ∼ **19**
empty interval **32**
enclosure
  rigorous ∼ 34, 37
  ∼ technique 5
enumeration **22**
epsilon-inflation **75**
equality constraint **26**
equation **26**
  linear interval ∼ 57, 64, **213**
equivalence-preserving **18**
Euclidean distance **219**
evaluating function **141**
evaluation function **100**
exclusion region **72**
exclusion test **68**
  Lipschitz ∼ **69**
  power series ∼ **70**
  Taylor ∼ **71**
exhaustive search **108**
existence of solution 61, 62, 64
existence test **73**
expansion
  power series ∼ 70
  Taylor ∼ **70**
exponent of floating-point number **31**
expression
  arithmetic ∼ **27**
  factorable ∼ **28**, 40
  real-valued ∼ 34
  separable ∼ **29**
extended function **130**
extending axis **193**
extending interval **193**
extension **193**
extreme test 1
extreme vertex **118**
  ∼ representation **118**

F

factorability **27**
factorable constraint **28**
factorable CSP **29**
factorable expression **28**, 40
  composition of ∼ **28**
factorable form 3, **27**
factorable function **28**, 40
fair strategy **89**
feasibility checker **107**
feasible region **102**, 114
feasible tableau 93
  optimal ∼ 93
fibers **130**
field **218**
finite-dimensional function **220**
finite-dimensional vector space **220**
first order derivative test 1
first order slope . . . . . *see* slope matrix, **215**
first order slope matrix **215**
fitted tree **193**
fitting process **194**
fixed point **220**
  Banach ∼ theorem **221**
  Brouwer ∼ theorem **220**
  Edelstein ∼ theorem **221**
  ∼ inverse 63, **213**
  ∼ method **56**
  Schauder ∼ theorem **221**
fixed point method **56**
  interval ∼ **55**
floating-point arithmetic **31**
floating-point interval **34**
floating-point representation **31**
form
  affine ∼ **37**
  factorable ∼ 3, **27**
  Gauss-Jordan ∼ 92
  globally solved ∼ **17**, 44
  separable ∼ **29**
  slope ∼ **215**
  ternary ∼ 30
forward cone **117**
forward evaluation **156**
forward-backward propagation **151**
  ∼ on DAGs **160**
function
  Z-extended ∼ **130**
  affine ∼ 37
  Boolean ∼ 12
  color ∼ **117**
  composite inclusion ∼ **148**
  continuous ∼ 25
  continuously differentiable ∼ . . . 57, 98, 137
  differentiable ∼ 137

distance ∼ **217**
elementary unary ∼ **27**
evaluating ∼ **141**
evaluation ∼ **100**
extended ∼ **130**
factorable ∼ **28**, 40
finite-dimensional ∼ **220**
inclusion ∼ 35, **146**
interval ∼ **35**, 56
interval form of ∼ **35**
linear ∼ 2, 143, 144
Lipschitz ∼ **219**
Lipschitz continuous ∼ 61–63, 69
monotone ∼ 70
monotone decreasing ∼ 137
monotone increasing ∼ 137
neighborhood ∼ **100**
nonlinear ∼ 37, 57, 99
objective 2
real-valued ∼ 34
representing ∼ **141**
semi-affine ∼ **39**
separable ∼ **29**
slope ∼ **215**
twice continuously differentiable ∼ . . 70, 72
twice differentiable ∼ 137
zero of ∼ **56**

G

Gauss elimination 1, 9, 65
interval ∼ **65**, 186
Gauss inverse **65**
Gauss-Jordan elimination 2
Gauss-Jordan form 92
Gauss-Seidel iteration
interval ∼ **59**, 186
preconditioned interval ∼ **59**
Gauss-Seidel operator
interval ∼ **59**
preconditioned interval ∼ **59**
generalized interval
Hansen ∼ 39
Kolev ∼ **39**
genetic method 55
global consistency 2, **44**
global method 55
gradient 210
graph
arc of ∼ **14**
complete ∼ 14
constraint ∼ **14**
node of ∼ **14**
griddy polyhedron 116
Gröbner computation 90

H

H-matrix 63–66, **213**
Hansen generalized interval 39
Hansen interval arithmetic 39
Hansen interval representation **144**
Hansen-Sengupta iteration **62**
Hansen-Sengupta operator **62**
Hausdorff distance **219**
Hessian matrix **211**
heterogeneous domain 10
high order term 91
hill climbing search 100
homogeneity 190
homotopy method 24
hull **33**
∼ consistency **48**, 80, 81
floating-point interval ∼ . . . . *see* $\mathbb{F}$-hull, **34**
interval ∼ **33**
∼ method **66**, 186
hull-disjoint **191**
hyper-arc consistency **45**
hyper-arc of hypergraph **14**
hypergraph
constraint ∼ **14**
hyper-arc of ∼ **14**
node of ∼ **14**

I

idempotence of domain reduction operator . . **19**
IEEE 754 . . . . . **31**
IEEE 854 . . . . . **31**
image under (multi)function **130**
implied constraint **20**
in-edge **152**
inactive variable **115**
inclusion converter **148**
inclusion function 35, **146**
composite ∼ **148**
natural ∼ **147**
inclusion property 35, **132**
inclusion relation **146**
inclusion representation **141**
inclusion technique 5
inclusion test **76**
generalized ∼ **76**
root ∼ **75**
incomplete algorithm **21**
incomplete search **24**
incompleteness of algorithm **21**
inconsistent CSP **13**
inequality **26**

linear interval ∼ . . . 64, 67, **213**
two-sided ∼ . . . **26**
inequality constraint . . . **26**
two-sided ∼ . . . **26**
inner approximation . . . **105**
inner union approximation . . . **106**
instantiation . . . **11**
interval . . . **32**
∼ addition . . . **33**
∼ box . . . **33**
closed ∼ . . . **32**
∼ consistency . . . **48**
∼ constraint propagation . . . 5
∼ constraint solver . . . 189
∼ division . . . **33**, **132**
empty ∼ . . . **32**
extending ∼ . . . **193**
∼ extension . . . **35**
floating-point ∼ . . . **34**
∼ function . . . **35**, 56
∼ Gauss elimination . . . **65**, 186
∼ Gauss-Seidel iteration . . . **59**, 186
∼ Gauss-Seidel operator . . . **59**
∼ hull . . . **33**
∼ hyper-arc consistency . . . **48**
∼ Jacobian matrix . . . 57, 75
left-open ∼ . . . **32**
∼ matrix . . . **212**
strongly regular ≈ . . . **214**
midpoint of ∼ . . . **33**
∼ multiplication . . . **33**
∼ Newton iteration . . . **63**
∼ Newton operator . . . **63**
open ∼ . . . **32**
radius of ∼ . . . **33**
real ∼ . . . **32**
∼ relaxation . . . 23
∼ representation . . . **142**
right-open ∼ . . . **32**
separating ∼ . . . **193**
∼ slope . . . 74
∼ square . . . **34**
∼ square root . . . **34**
∼ subtraction . . . **33**
∼ tree . . . **190**
universal ∼ . . . **33**
width of ∼ . . . **33**
interval arithmetic . . . 5, **32**
dependency of ∼ . . . **33**, 89
exact ∼ . . . **33**
Hansen ∼ . . . 39
idealized ∼ . . . **33**
rounded ∼ . . . **34**
spirit of ∼ . . . **33**, 34
standard ∼ . . . **33**
subdistributivity of ∼ . . . **33**
interval contraction . . . **57**
continuous ∼ . . . **57**
interval form
centered ∼ . . . **210**
∼ of function . . . **35**
mixed centered ∼ . . . **210**
∼ of multifunction . . . **132**
natural ∼ . . . 82, **210**
∼ of relation . . . **35**
Taylor ∼ . . . **211**
interval matrix
midpoint of ∼ . . . **33**
radius of ∼ . . . **33**
width of ∼ . . . **33**
interval vector . . . **33**
midpoint of ∼ . . . **33**
radius of ∼ . . . **33**
width of ∼ . . . **33**
interval-based precision . . . **108**
intervals
addition of ∼ . . . **33**
division of ∼ . . . **33**, **132**
multiplication of ∼ . . . **33**
partial order on ∼ . . . **33**
subtraction of ∼ . . . **33**
introduced variable . . . **20**
inverse
fixed point ∼ . . . 63, **213**
Gauss ∼ . . . **65**
∼ image . . . 78, **130**
matrix ∼ . . . **212**
inverse image under (multi)function . . . **130**
isolated solution . . . **101**
iteration
Hansen-Sengupta ∼ . . . **62**
interval Gauss-Seidel ∼ . . . **59**, 186
preconditioned ≈ . . . **59**
interval Newton ∼ . . . **63**
Kolev ∼ . . . **98**
Krawczyk ∼
≈ for linear interval equation . . . **58**, 186
≈ for nonlinear equation . . . **60**
Newton ∼ . . . 55
iterative descent method . . . 92
iterative method . . . 55

J

Jacobian matrix
interval ∼ . . . 57, 75

K

Karush-Kuhn-Tucker condition . . . 4
Kolev affine arithmetic . . . **42**
Kolev affine form . . . **39**
Kolev affine representation . . . **144**
Kolev generalized affine arithmetic . . . **40**
Kolev generalized interval . . . **39**
Kolev iteration . . . **98**
Kolev noise variable . . . **39**
Krawczyk iteration
  ∼ for linear interval equation . . . **58**, 186
  ∼ for nonlinear equation . . . **60**
Krawczyk operator
  ∼ for linear interval equation . . . **58**
  ∼ for nonlinear equation . . . **60**

L

labeling . . . **22**
leaf . . . 23, **152**
left-open interval . . . **32**
length of affine form . . . **37**
linear approximation property . . . 68
linear function . . . 2, 143, 144
linear interval equation . . . 57, 64, **213**
  Krawczyk iteration for ∼ . . . **58**, 186
  Krawczyk operator for ∼ . . . **58**
linear interval inequality . . . 64, 67, **213**
linear mapping . . . **211**
linear problem . . . 3
  mixed integer ∼ . . . 3
linear program . . . 3, **90**
  mixed integer ∼ . . . 3
linear programming . . . 2, 5, 90
linear relaxation . . . 23
Lipschitz condition . . . 69, **219**
Lipschitz constant . . . 69, **219**
Lipschitz continuous function . . . 61–63, 69
Lipschitz exclusion test . . . **69**
Lipschitz function . . . 61–63, 69, **219**
Lipschitz mapping . . . **219**
Lipschitz matrix . . . 57, **214**
  strongly regular ∼ . . . 61, 63, 64
Lipschitz set . . . 63, 64, **214**, 216
local consistency . . . 2, 9, 23, **44**
local method . . . 55
local search . . . 24
logic constraint . . . **13**
logic problem . . . **13**
logic programming . . . 1, 10, **13**
logic reasoning . . . 1
lower bound . . . **32**
lower triangular matrix . . . **65**
lower-bound tree . . . **197**

M

M-matrix . . . 65, **212**
mantissa . . . *see* significand, **31**
mapping
  compact ∼ . . . **220**
  contraction ∼ . . . **219**
  contractive ∼ . . . **219**
  linear ∼ . . . **211**
  Lipschitz ∼ . . . **219**
  nonexpansive ∼ . . . **219**
  regular ∼ . . . 211
  sublinear ∼ . . . 64, **211**
mathematical computing . . . 1
mathematical programming . . . 1
mathematical rigor . . . 3
matrix
  comparison ∼ . . . **212**
  H- ∼ . . . 63–66, **213**
  Hessian ∼ . . . **211**
  interval ∼ . . . **212**
  interval Jacobian ∼ . . . 57, 75
  ∼ inverse . . . **212**
  Lipschitz ∼ . . . 57, **214**
  lower triangular ∼ . . . **65**
  M- ∼ . . . 65, **212**
  preconditioning ∼ . . . 58, 59
  real ∼ . . . **212**
  slope ∼ . . . 61, 62, 64, 70, 72, 74, **215**
  strongly regular interval ∼ . . . **214**
  strongly regular Lipschitz ∼ . . . 61, 63, 64
  upper triangular ∼ . . . **65**
matrix set . . . **214**
  regular ∼ . . . **214**
max-connected boxes . . . **191**
max-connected clustering . . . **191**
maximum absolute error . . . **38**
Messine affine arithmetic . . . **41**
Messine affine form . . . **41**
method
  chord ∼ . . . **76**
  clustering ∼ . . . 100
  complete ∼ . . . 55
  complete search ∼ . . . 23
  completing the square ∼ . . . 1
  connectionist ∼ . . . 100
  continuation ∼ . . . 24, 55, 100
  domain-specific ∼ . . . 2, 55
  fixed point ∼ . . . **56**
  Gauss elimination ∼ . . . 1, 9, 65
  Gauss-Jordan elimination ∼ . . . 2
  general ∼ . . . 2
  genetic ∼ . . . 55
  global ∼ . . . 55

grouping genetic ∼ . . . 100
homotopy ∼ . . . 24, 55, 100
hull ∼ . . . **66**, 186
initiative evolutionary ∼ . . . 100
interval fixed point ∼ . . . **55**
interval Gauss elimination ∼ . . . **65**, 186
iterative ∼ . . . 55
iterative descent ∼ . . . 92
local ∼ . . . 55
local descent ∼ . . . 100
microgenetic iterative descent ∼ . . . 100
Newton-like ∼ . . . 24, 55, 72
response surface ∼ . . . 100
search ∼ . . . 2
simplex ∼ . . . 2, **92**
solution ∼ . . . **3**
stepwise adaptation of weights ∼ . . . 100
stochastic ∼ . . . 55
zooming adaptation of weights ∼ . . . 100
metric . . . **217**
metric space . . . **217**
compact ∼ . . . **218**
complete ∼ . . . **218**
midpoint of interval . . . **33**
midpoint of interval matrix . . . **33**
midpoint of interval vector . . . **33**
min-conflicts heuristic . . . 100
mixed centered interval form . . . **210**
mixed integer linear problem . . . 3
mixed integer linear program . . . 3
mixed integer nonlinear problem . . . 3
mixed integer nonlinear program . . . 3
modeling
problem ∼ . . . 2, 13
monotone function . . . 70
monotonicity of `CB` operator . . . **113**
monotonicity of domain reduction operator . **19**
monotonicity of `DR` operator . . . **113**
monotonicity of consistency . . . **43**
multi-index . . . **70**
multifunction . . . 35, **130**
multigraph . . . **152**
arc of ∼ . . . **152**
directed ∼ . . . **152**
≈ with ordered edges . . . **153**
directed acyclic ∼ . . . **152**
edge of ∼ . . . **152**
node of ∼ . . . **152**
vertex of ∼ . . . **152**
multiple random start search . . . 100
multiplication of affine forms . . . **38**
∼ in Kolev affine arithmetic . . . **42**
∼ in Messine affine arithmetic . . . **41**
∼ in revised affine arithmetic . . . **135**
multiplication of intervals . . . **33**
multiplication of Kolev affine forms . . . **40**
multiplication of Messine affine forms . . . **41**
multiplicative identity . . . **218**

N

natural extension . . . 34, **147**
natural inclusion function . . . **147**
natural interval form . . . 82, **210**
NCSP . . . *see* numerical CSP, **25**
relational $(k, k-1)$-consistency for ∼ . . . 54
ternarized ∼ . . . 207
negation of constraints . . . **15**
negation test . . . **109**
neighborhood function . . . **100**
neighborhood of grid point . . . **118**
Newton iteration . . . 55
general interval ∼ . . . **64**
interval ∼ . . . **63**
Newton operator
general interval ∼ . . . **64**
interval ∼ . . . **63**
Newton-like method . . . 24, 55, 72
node
∼ of graph . . . **14**
∼ of hypergraph . . . **14**
∼ of multigraph . . . **152**
search ∼ . . . **23**
node consistency . . . **44**
node range . . . **153**
backward ∼ . . . **82**
forward ∼ . . . **82**
noise symbol . . . **37**
noise variable . . . **37**
Kolev ∼ . . . **39**
non-affine operation . . . **37**
non-isolated solution . . . **101**
nonbasic variable . . . **92**
nonexpansive mapping . . . **219**
nonlinear equation
Krawczyk iteration for ∼ . . . **60**
Krawczyk operator for ∼ . . . **60**
∼ system . . . **57**
nonlinear function . . . 37, 57, 99
nonlinear problem . . . 3
mixed integer ∼ . . . 3
nonlinear program . . . 3
mixed integer ∼ . . . 3
nonzero coefficient . . . **37**
normalized CSP . . . **14**
normalized number . . . **31**
Not a Number . . . **32**

number system
  floating-point ∼ **30**
  sign/logarithm ∼ 31
  signed-digit ∼ 31
  slash ∼ 31
numerical constraint 3, **25**
numerical constraint propagation 5
numerical constraint satisfaction problem 3, **25**
numerical CSP **25**

O

objective function 2
open ball **217**
open cover **217**
open interval **32**
open set **217**
operation
  affine ∼ **37**, 40
  elementary ∼ 3, **28**
  elementary binary ∼ **27**
  non-affine ∼ **37**
operation research 9
operator
  box splitting ∼ **114**
  `CB` ∼ **110**
  $\mathtt{CB}_{\mathtt{rd}}$ ∼ **115**
  complementary boxing ∼ **110**
  contracting ∼ **57**
  ∼ dichotomous splitting **113**
  domain reduction ∼ **19**, **108**
  `DR` ∼ **108**
  $\mathtt{DR}_{\mathtt{rd}}$ ∼ **115**
  Hansen-Sengupta ∼ **62**
  interval Gauss-Seidel ∼ **59**
    preconditioned ≈ **59**
  interval Newton ∼ **63**
  Kolev ∼ **98**
  Krawczyk ∼
    ≈ for linear interval equation **58**
    ≈ for nonlinear equation **60**
  narrowing ∼ **57**
  projection narrowing ∼ 82
optimal max-connected clustering **191**
optimal solution 2
optimal tableau 94
optimality condition
  second order necessary ∼ 4
  second order sufficient ∼ 4
order k
  approximation property of ∼ **35**
ordering
  dynamic variable ∼ 79
  value ∼ 79
orthogonal polyhedron 116
orthogonal-separable tree **191**
out-edge **152**
outer approximation **105**
outer union approximation **106**

P

P-contraction 75
parent 23, **153**
partial derivative 70
partial order on intervals **33**
path consistency **46**
  directional ∼ **46**
  strong ∼ **46**
point-wise approach 101
polyhedron
  griddy ∼ 116
  orthogonal ∼ 116
polytope 73
post-order 82
post-processing 6, **190**
  ∼ technique 204
power series exclusion test **70**
power series expansion 70
power term **91**
pre-order 82
pre-process **21**
precision **108**
precompact set **217**
primitive box **190**
primitive constraint **28**, 49
problem
  constrained optimization ∼ 2
  constraint logic ∼ **13**
  constraint satisfaction ∼ 2, 9
  decision ∼ 13
  ∼ equivalence **18**
  linear ∼ 3
  logic ∼ **13**
  mixed integer linear ∼ 3
  mixed integer nonlinear ∼ 3
  ∼ modeling 2, 13
  nonlinear ∼ 3
  numerical constraint satisfaction ∼ **25**
  ∼ relaxation **18**
  satisfiability ∼ 2, 13
  set inversion ∼ **78**
  under-constrained ∼ **27**
  well-constrained ∼ **27**
problem reduction **23**
  ∼ technique 20
program
  linear ∼ 3, **90**

mixed integer linear ∼ . . . 3
mixed integer nonlinear ∼ . . . 3
nonlinear ∼ . . . 3
programming
constraint ∼ . . . 1, **2**, 10
constraint logic ∼ . . . 10, **13**
linear ∼ . . . 2, 5, 90
logic ∼ . . . 1, 10, **13**
mathematical ∼ . . . 1
projection . . . **15**
∼ narrowing operator . . . 82
propagation
backward ∼ . . . **158**
constraint ∼ . . . 9, **23**
forward-backward ∼ . . . **151**
property
inclusion ∼ . . . 35, **132**
linear approximation ∼ . . . 68
quadratic approximation ∼ . . . 60, 66, 68
pseudograph . . . **152**
directed ∼ . . . **152**

Q

quadratic approximation property . . . 60, 66, 68
quadratic formula . . . 1
quadratic term . . . **91**

R

radius
spectral ∼ . . . **212**
radius of interval . . . **33**
radius of interval matrix . . . **33**
radius of interval vector . . . **33**
range
constraint ∼ . . . **25**, 81
node ∼ . . . **153**
rational arithmetic . . . 31
real addition . . . 38, 40
real evaluation generator . . . **141**
real inclusion representation . . . **141**
real interval . . . **32**
real matrix . . . **212**
real multiplication . . . 38, 40
real representation . . . **141**
real slope . . . **96**
real-valued expression . . . 34
real-valued function . . . 34
redundant compound label . . . **20**
redundant constraint . . . **20**
∼ elimination . . . **23**
∼ removal . . . **23**
redundant value . . . **20**
region
exclusion ∼ . . . **72**
feasible ∼ . . . **102**, 114
regular CSP . . . **14**
regular mapping . . . 211
regular matrix set . . . **214**
relation
$\prec\prec$ ∼ . . . *see* domination relation, **70**
$\prec_k$ ∼ . . . *see* domination relation, **70**
domination ∼ . . . **70**, 71, 95
inclusion ∼ . . . **146**
interval form of ∼ . . . **35**
relational (n_v, n_c)-consistency . . . **53**
relaxation . . . 23
interval ∼ . . . 23
linear ∼ . . . 23
problem ∼ . . . **18**
∼ system . . . **18**
relaxing . . . **19**
removal
redundant constraint ∼ . . . **23**
representation
2^k-tree ∼ . . . 54, 78
canonical ∼ . . . **117**
compact tree ∼ . . . **81**
DAG ∼ . . . 20, 82, **153**
disjoint box ∼ . . . 105
extreme vertex ∼ . . . **118**
floating-point ∼ . . . **31**
Hansen interval ∼ . . . **144**
inclusion ∼ . . . **141**
interval ∼ . . . **142**
Kolev affine ∼ . . . **144**
linear relaxation ∼ . . . **144**
∼ object . . . **141**
partial DAG ∼ . . . **160**
real ∼ . . . **141**
real inclusion ∼ . . . **141**
revised affine ∼ . . . **143**
standard affine ∼ . . . **143**
tree ∼ . . . **81**
representing function . . . **141**
restricted-dimensional
∼ complementary boxing operator . . . **115**
∼ domain reduction operator . . . **115**
revised affine arithmetic . . . **133**
multiplication in ∼ . . . **135**
revised affine form . . . 133, **134**
revised affine representation . . . **143**
right-open interval . . . **32**
rigor . . . 3
rigor of algorithm . . . **21**
rigorous algorithm . . . **21**
rigorous enclosure . . . 34, 37

ring **218**
- commutative ∼ **218**

root **152**
rounding control 34
- rigorous ∼ 42, 185

running constraint **109**

S

satisfiability of constraint **11**
satisfiability problem 2, 13
satisfiable CSP **13**
search 5
- backjumping ∼ 79
  - backmarking with ≈ 79
  - conflict-directed ≈ 79
  - forward checking with ≈ 79
  - graph-based ≈ 79
- backmarking ∼ 78
- backtracking ∼ 78
  - dynamic ≈ 79
  - flexible partial order ≈ 79
  - general partial order ≈ 79
  - partial chronological ≈ 79
  - partial dynamic ≈ 79
  - partial order ≈ 79
- bisection ∼ 79
- breadth-first ∼ 78
- cell subdivision ∼ 79
- complete ∼ 5, **23**
- conflict-directed backjumping ∼ 79
  - backmarking with ≈ 79
  - forward checking with ≈ 79
- depth-first ∼ 78
  - interleaved ≈ 78
- discrepancy ∼
  - depth-bounded ≈ 78
  - improved limited ≈ 78
  - limited ≈ 78
- exhaustive ∼ **108**
- forward checking ∼ 79
  - minimal ≈ 79
  - partial ≈ 79
- genetic ∼ 100
- griding ∼ 79
- hill climbing ∼ 100
- incomplete ∼ **24**
- iterative broadening ∼ 78
- local ∼ 24
- look-ahead ∼
  - full ≈ 79
  - partial ≈ 79
- maintaining arc consistency ∼ 79
- ∼ method 2
- multiple random start ∼ 100
- ∼ node **23**
- random-walk ∼ 100
- simulated annealing ∼ 100
- ∼ space **23**
- stochastic ∼ 24
- tabu ∼ 100
- ∼ tree **23**
- uniform dichotomous ∼ **70**

second order derivative test 1
second order necessary optimality condition 4
second order slope *see* slope matrix, **215**
second order slope function **215**
second order slope matrix **215**
second order sufficient optimality condition 4
section of constraints **15**
section of CSP **16**
semi-affine function **39**
separable constraint **30**
separable CSP **30**
separable expression **29**
separable form **29**
separable function **29**
separating axis **193**
separating interval **193**
separating process **196**
separating set **194**
separator **193**
set
- closed ∼ **217**
- compact ∼ 17, **218**
- connected ∼ **32**
- convex ∼ 71, **219**
- disconnected ∼ **32**
- ∼ inversion problem **78**
- Lipschitz ∼ 63, 64, **214**, 216
- open ∼ **217**
- precompact ∼ **217**
- separating ∼ **194**
- solution ∼ **13**

significand **31**
simple gridding solver **124**
simplex method 2, **92**
- dual ∼ **92**

singleton consistency 16, **54**
singleton CSP **16**
singleton variable **19**
slack variable **20**
slice of constraints **16**
slice of CSP **17**
slope 75
- first order ∼ *see* slope matrix, **215**
- ∼ form **215**

∼ function ... **215**
interval ∼ ... 74
real ∼ ... **96**
second order ∼ ... *see* slope matrix, **215**
slope function ... **215**
second order ∼ ... **215**
slope matrix ... 61, 62, 64, 70, 72, 74, **215**
first order ∼ ... **215**
second order ∼ ... **215**
solution ... 2, **13**
basic feasible ∼ ... **92**
existence of ∼ ... 61, 62, 64
isolated ∼ ... **101**
non-isolated ∼ ... **101**
optimal ∼ ... 2
∼ set ... **13**
sound ∼ ... **21**
∼ synthesis ... **22**
uniqueness of ∼ ... 56, 61–63
solution algorithm ... **3**
asymptotically complete ∼ ... 3
complete ∼ ... 3
incomplete ∼ ... 3
rigorous ∼ ... 3
solution method ... **3**
asymptotically complete ∼ ... 3
complete ∼ ... 3
incomplete ∼ ... 3
rigorous ∼ ... 3
solution technique ... **3**
asymptotically complete ∼ ... 3
complete ∼ ... 3
incomplete ∼ ... 3
interval-based ∼ ... 6
rigorous ∼ ... 3
solutions
continuum of ∼ ... 6
solver
constraint ∼ ... 3
interval constraint ∼ ... 189
simple gridding ∼ ... **124**
solving
constraint ∼ ... **9**
∼ process ... **21**
sound algorithm ... **21**
sound approximation ... **105**
sound solution ... **21**
soundness of algorithm ... **21**
source of edge ... **152**
space
Banach ∼ ... **219**
metric ∼ ... **217**
normed vector ∼ ... **218**
search ∼ ... **23**
vector ∼ ... **218**
spectral radius ... **212**
splitting ... **22**
∼ technique ... 4
square
interval ∼ ... **34**
square root
interval ∼ ... **34**
standard affine representation ... **143**
standardized CSP ... **14**
stochastic method ... 55
stochastic search ... 24
strategy
cooperative ∼ ... 89
fair ∼ ... **89**
strong k-consistency ... **47**
strong convergence ... **56**, 63, 64
strong interval contraction ... **57**
strong path consistency ... **46**
strong (i, j)-consistency ... **53**
strongly regular interval matrix ... **214**
strongly regular Lipschitz matrix ... 61, 63, 64
strongness of consistency ... **43**
subcover ... **217**
subdistributivity of interval arithmetic ... **33**
subdivision
cell ∼ ... **22**
uniform cell ∼ ... **70**
sublinear mapping ... 64, **211**
system
nonlinear equation ∼ ... **57**
relaxation ∼ ... **18**

T

tableau
basic ∼ ... **92**
feasible ∼ ... 93
optimal ∼ ... 94
optimal feasible ∼ ... 93
target of edge ... **152**
Taylor exclusion test ... **71**
Taylor expansion ... **70**
Taylor interval form ... **211**
technique
constraint programming ∼ ... 4
constraint propagation ∼ ... 2
discretization ∼ ... 4
domain reduction ∼ ... 20
enclosure ∼ ... 5
inclusion ∼ ... 5
post-processing ∼ ... 204
problem reduction ∼ ... 20

solution ∼ **3**
splitting ∼ 4
term
high order ∼ 91
power ∼ **91**
quadratic ∼ **91**
ternarization **30**, 54
ternarized constraint **30**
ternarized NCSP 207
ternary constraint **13**, 78
ternary form 30
test
exclusion ∼ **68**
existence ∼ **73**
extreme ∼ 1
first order derivative ∼ 1
inclusion ∼ **76**
negation ∼ **109**
second order derivative ∼ 1
uniqueness ∼ **73**
transformation of a CSP **18**
tree
2^k- ∼ 78
AABB ∼ **190**
bounding-box ∼ **190**
bounding-volume ∼ **190**
fitted ∼ **193**
interval ∼ **190**
lower-bound ∼ **197**
orthogonal-separable ∼ **191**
∼ representation **81**
search ∼ **23**
upper-bound ∼ **197**
triangular matrix
lower ∼ **65**
upper ∼ **65**
triangulation of a constraint graph **46**
two-sided inequality **26**
two-sided inequality constraint **26**

U

unary constraint **13**
under-constrained problem **27**
undiscernible box **107**
uniform dichotomization 69
uniform dichotomous search **70**
uniqueness of solution 56, 61–63
uniqueness test **73**
universal constraint **11**, 14
universal interval **33**
upper bound **32**
upper triangular matrix **65**
upper-bound tree **197**

V

variable **10**
active ∼ **115**
artificial ∼ **92**
basic ∼ **92**
continuous ∼ 3, **24**
domain ∼ **10**
∼ elimination **19**
inactive ∼ **115**
introduced ∼ **20**
∼ introduction **20**
noise ∼ **37**
nonbasic ∼ **92**
singleton ∼ **19**
slack ∼ **20**
vector
column ∼ **33**
∼ of constraint ranges **25**
interval ∼ **33**
vector space **218**
complete normed ∼ **219**
finite-dimensional ∼ **220**
normed ∼ **218**
vertex of multigraph **152**

W

well-constrained problem **27**
width of interval **33**
width of interval matrix **33**
width of interval vector **33**

Z

zero of function **56**

Curriculum Vitae

Personal Data

Family Name:	**Vu**	(In Vietnamese: **Vũ**)
First Name:	**Xuan-Ha**	(In Vietnamese: **Xuân Hạ**)
Sex:	Male	
Marital Status:	Single	
Date of Birth:	June 1973	
Place of Birth:	Vietnam	
Nationality:	Vietnamese	

Education

03/2005 — **Doctor of Philosophy (Docteur ès Sciences) in Computer Science, Constraint Programming**, awarded by the Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland. **PhD dissertation:** *Rigorous Solution Techniques for Numerical Constraint Satisfaction Problems*, accepted on December 7, 2004.

05/1995 — **Engineer/Bachelor of Engineering in Computer Science** (a 5-year program); awarded by Hanoi University of Technology (HUT), Vietnam.

Awards and Honors

1995 — **Nguyen Truong To Scholarship**, awarded to about 100 best (among hundreds of thousands of) undergraduate students in Vietnam. This is a merit scholarship in memory of Nguyen Truong To, a famous Vietnamese innovator well known for his relentless efforts to modernize Vietnam.

1990 — **Bronze Medal** in the *31st International Mathematical Olympiad*, in Beijing, China.

1989 — All top prizes: **First Prize** (in the final result), **Excellent Prize** (for many excellent solutions), **Special Prize** (for competition with students in higher class); awarded by *Vietnamese Journal of Mathematics and Youth* for the solution of mathematical problems in the *Annual Contest* organized by this journal.

Professional Experience

12/2000 – 12/2004 **Research Assistant/PhD Student** at the Artificial Intelligence Laboratory (LIA) in the Swiss Federal Institute of Technology in Lausanne (EPFL). Main task: developing new constraint programming techniques, which were documented in my PhD thesis, for the COCONUT project[1].

06/1999 – 11/2000 **Engineer** in computer science at Vietnam Data-communication Company (VDC), Vietnam.

06/1995 – 05/1999 **System/Software Engineer** in computer science at the Corporation for Financing and Promoting Technology (FPT), Vietnam.

Publications

Three Representative Papers

1. X.-H. Vu, D. Sam-Haroud and B. Faltings. Combining Multiple Inclusion Representations in Numerical Constraint Propagation. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, pages 458–467, Florida, USA, November 2004. [Acceptance rate: 54/205 ≈ 26%. Nominated for the Best Paper Award by the reviewers of this paper.]

2. X.-H. Vu, H. Schichl and D. Sam-Haroud. Using Directed Acyclic Graphs to Coordinate Propagation and Search for Numerical Constraint Satisfaction Problems. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, pages 72–81, Florida, USA, November 2004. [Acceptance rate: 54/205 ≈ 26%.]

3. X.-H. Vu, D. Sam-Haroud and M.-C. Silaghi. Numerical Constraint Satisfaction Problems with Non-isolated Solutions. *Global Optimization and Constraint Satisfaction: First International Workshop on Global Constraint Optimization and Constraint Satisfaction, COCOS 2002, Revised Selected Papers*, LNCS 2861, pages 194–210, Springer-Verlag 2003. [Two rounds of reviews in COCOS 2002.]

Journal Papers

1. X.-H. Vu, M.-C. Silaghi, D. Sam-Haroud and B. Faltings. Numerical Constraint Satisfaction Problems: Part I - General Search Strategies. *ACM Transactions on Computational Logic* (TOCL). To be submitted in 2005.

2. X.-H. Vu, H. Schichl and D. Sam-Haroud. Numerical Constraint Satisfaction Problems: Part II - Coordinating Propagation and Search on Directed Acyclic Graphs. *ACM Transactions on Computational Logic* (TOCL). To be submitted in 2005.

3. X.-H. Vu, D. Sam-Haroud and B. Faltings. Numerical Constraint Satisfaction Problems: Part III - Combining Multiple Inclusions in Constraint Propagation. *ACM Transactions on Computational Logic* (TOCL). To be submitted in 2005.

[1] The COCONUT (Continuous Constraints: Updating the Technology) project is a research project funded by the European Union under the IST contract IST-2000-26063.

Refereed Conference/Workshop Papers

1. X.-H. Vu, D. Sam-Haroud and B. Faltings. Combining Multiple Inclusion Representations in Numerical Constraint Propagation. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, pages 458–467, Florida, USA, November 2004. [Acceptance rate: 54/205 ≈ 26%. Nominated for the Best Paper Award by the reviewers of this paper.]

2. X.-H. Vu, H. Schichl and D. Sam-Haroud. Using Directed Acyclic Graphs to Coordinate Propagation and Search for Numerical Constraint Satisfaction Problems. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, pages 72–81, Florida, USA, November 2004. [Acceptance rate: 54/205 ≈ 26%.]

3. X.-H. Vu, D. Sam-Haroud and B. Faltings. Clustering for Disconnected Solution Sets of Numerical CSPs. *Recent Advances in Constraints: International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2003, Budapest, Hungary, June 30 – July 2, 2003, Selected Papers*, pages 25–43, LNAI 3010, Springer-Verlag 2004.

4. X.-H. Vu, D. Sam-Haroud and M.-C. Silaghi. Numerical Constraint Satisfaction Problems with Non-isolated Solutions. *Global Optimization and Constraint Satisfaction: First International Workshop on Global Constraint Optimization and Constraint Satisfaction, COCOS 2002, Valbonne-Sophia Antipolis, France, October 2–4, 2002, Revised Selected Papers*, pages 194–210, LNCS 2861, Springer-Verlag 2003.

5. O. Shcherbina, A. Neumaier, D. Sam-Haroud, X.-H. Vu and T.-V. Nguyen. Benchmarking Global Optimization and Constraint Satisfaction Codes. *Global Optimization and Constraint Satisfaction: First International Workshop on Global Constraint Optimization and Constraint Satisfaction, COCOS 2002, Valbonne-Sophia Antipolis, France, October 2–4, 2002, Revised Selected Papers*, pages 211–222, LNCS 2861, Springer-Verlag 2003.

6. X.-H. Vu, D. Sam-Haroud and M.-C. Silaghi. Approximation Techniques for Non-linear Problems with Continuum of Solutions. In *Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation, SARA 2002*, pages 224–241, Alberta, Canada, August 2002, LNAI 2371, Springer-Verlag 2002.

7. X.-H. Vu, D. Sam-Haroud and M.-C. Silaghi. Résolution de problèmes non-linéaires avec continuum de solutions. In *Programmation en logique avec contraintes: Actes des Onzièmes Journées Francophones de Programmation Logique et Programmation par Contraintes, JFPLC 2002*, pages 27–41, Nice, France, May 2002, HERMES Science Publications.

Lausanne, March 14, 2005