UNIVERSITY OF TECHNOLOGY SYDNEY
Faculty of Engineering and Information Technology

Exploring Centralized and Distributed Constraint Propagation Algorithms for Solving Constraint Satisfaction Problems

by

Shufeng Kong

Dissertation directed by Professor Sanjiang Li

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Sydney, Australia

2018

Certificate of Authorship/Originality

I certify that the work in this thesis has not been previously submitted for a degree nor has it been submitted as a part of the requirements for other degree except as fully acknowledged within the text.

I also certify that this thesis has been written by me. Any help that I have received in my research and in the preparation of the thesis itself has been fully acknowledged. In addition, I certify that all information sources and literature used are quoted in the thesis.

Signature of student:

Production Note:
Signature removed prior to publication.

Dedication

To my beloved mom, Miaoping Deng, for raising me up and her unwavering support.

Acknowledgements

This thesis would not be possible without the help and contribution of many others. Most significantly, I am much indebted to my advisor, Professor Sanjiang Li, for his guidance and support throughout my PhD study. Sanjiang always paves the way for me and also his high standard and constructive criticisms have made me a better researcher and helped me grow as a person.

I am also grateful to my collaborators, Dr. Zhiguo Long, Dr. Jae Hee Lee and Dr. Michael Sioutis. Without their help, my thesis would not be as good as it is now. Zhiguo cheered me up when I was feeling depressed and tried his best to offer me help when I needed it. I learn a lot from Jae Hee about critical thinking and how to write a good research paper. He never just imposes his opinions on me but always tries to persuade me with right reasons.

I owe a great debt of gratitude to Professor Rina Dechter for her insightful ideas and wise advices when I was a visiting student at UC Irvine. Rina is very knowledgeable and far-sighted. She let me explore and pursue the questions I found of interest, but also provided direction when I needed it. It has been a pleasure and a privilege to work with her. I also want to thank the members of Rina's automated reasoning group, Junkyu Lee, Filjor Broka, Bobak Pezeshki and Qi Lou, for their warmest welcome and support when I stayed at UC Irvine.

Last but not least, I could not have done this without the incredible support of my family, especially, my beloved mom. She always believes in me even when I have doubts about myself.

Shufeng Kong

Sydney, Australia, 2018.

List of Publications

Journal Papers

J-1. **Shufeng Kong**, Sanjiang Li, Yongming Li and Zhiguo Long: On tree-preserving constraints. *Annals of Mathematics and Artificial Intelligence*, 81(3-4): 241-271 (2017)

J-2. **Shufeng Kong**, Sanjiang Li and Michale Sioutis: Exploring directional path-consistency for solving constraint networks. *The Computer Journal*, first online: 27 December 2017. doi: 10.1093/comjnl/bxx122

J-3. **Shufeng Kong**, Jae Hee Lee and Sanjiang Li: A new distributed algorithm for efficient generalized arc-consistency propagation. *Autonomous Agent and Multi-Agent Systems*, first online: 10 May 2018. doi: 10.1007/s10458-018-9388-x

Conference Papers

C-1. **Shufeng Kong**, Sanjiang Li, Yongming Li and Zhiguo Long: On tree-preserving constraints. In *Proceeding of the 21st International Conference on Principles and Practice of Constraint Programming* (CP'15), pp. 244-261 (2015)

C-2. **Shufeng Kong**, Jae Hee Lee and Sanjiang Li: A deterministic distributed algorithm for reasoning with connected row-convex constraints. In *Proceeding of the 16th International Conference on Autonoumous Agent and Multiagent Systems* (AAMAS'17), pp. 203-211 (2017)

C-3. **Shufeng Kong**, Jae Hee Lee and Sanjiang Li: Multiagent simple temporal problem: the arc-consistency approach. In *Proceeding of the 32th AAAI Conference on Artificial Intelligence* (AAAI'18), New Orleans, Louisiana, USA, February 2-7, 2018.

Contents

Certificate ii

Dedication iii

Acknowledgments iv

List of Publications v

List of Figures x

List of Tables xiv

Abbreviation xv

Abstract xvi

1 Introduction 1

1.1 Related Works 3

1.1.1 Local Consistency Algorithms for CSPs 3

1.1.2 Tractable Subclasses of CSPs 4

1.1.3 Local Consistency Algorithms for DisCSPs 6

1.2 Thesis Outline and Contributions 7

1.2.1 Tree-Preserving Constraints 7

1.2.2 CSPs Solvable with Directional PC 8

1.2.3 A Distributed Partial PC Algorithm for Solving CRC Constraint Networks 9

1.2.4 A Distributed AC Algorithm for Solving Multiagent STPs . . 10

1.2.5 A New Distributed Generalized AC Algorithm 11

1.3 Preliminaries 11

1.3.1 Constraint Satisfaction Problem 11

1.3.2 Local Consistency and Properties of Constraint Networks . . . 15

1.3.3 Triangulated Constraint Networks 17

1.3.4 Connected Row-Convex Constraint Networks 18

1.3.5 Algebraic Closure Properties of Constraints 20

2 Tree-Preserving Constraints 23

2.1 Contribution 23

2.2 Introduction and Chapter Outline 23

2.3 Preliminaries 27

2.4 Tree-Preserving Constraints 32

2.5 Partial PC for Tree-Preserving Constraints 38

2.6 The Scene Labelling Problem 47

2.7 Algebraic Closure Properties of Tree-Preserving Constraints 49

2.8 Evaluations 56

2.9 Conclusion 59

3 CSPs Solvable with Directional PC 70

3.1 Contribution 70

3.2 Introduction and Chapter Outline 71

3.3 Preliminaries 73

3.4 The Strong Directional PC Algorithm 75

3.5 Directional PC and Variable Elimination 76

3.6 Weak VEP Classes and Majority-Closed Classes 83

3.7 The Variable Elimination Algorithm **DPC*** 85

3.8 Evaluations 90

3.9 Conclusion 94

4 A Distributed Partial PC Algorithm for Solving CRC Constraint Networks 95

4.1 Contribution 95

4.2 Introduction and Chapter Outline 95

4.3 Preliminaries 98

4.4 An Efficient Centralized Algorithm for CRC Constraints 98

4.5 An Efficient Distributed Algorithm for CRC Constraints 107

4.6 Evaluations 111

4.6.1 Theoretical Comparisons 111

4.6.2 Experimental Comparisons 113

4.6.3 In-depth Evaluation of DΔCRC 114

4.7 Conclusion 115

5 A Distributed AC Algorithm for Solving Multiagent STPs 116

5.1 Contribution 116

5.2 Introduction and Chapter Outline 116

5.3 Preliminaries 118

5.4 Solving the STP with Arc-Consistency 121

5.4.1 A Centralized AC Algorithm for the STP 125

5.5 Solving the MaSTP with Arc-Consistency 128

5.6 Evaluation 132

5.6.1 ACSTP vs. P^{3}C 134

5.6.2 DisACSTP vs. D△PPC 135

5.7 Conclusion 136

6 A New Distributed Generalized AC Algorithm 137

6.1 Contribution 137

6.2 Introduction and Chapter Outline 137

6.3 Preliminaries 140

6.4 A New Distributed Arc-Consistency Algorithm 141

6.4.1 The Algorithm 144

6.4.2 Handling Termination 149

6.5 Analysis of DisAC3.1 154

6.5.1 Privacy of Individual Agents 154

6.5.2 Termination and Correctness 158

6.5.3 Time and Space Complexities 162

6.6 Non-binary Constraints 164

6.6.1 Generalized Arc-Consistency 164

6.6.2 A Distributed Generalized Arc-Consistency Algorithm 167

6.7 Evaluation 171

6.7.1 DisAC3.1 vs. DisAC9 173

6.7.2 DisGAC3.1 vs. GAC2001/3.1 178

6.8 Conclusion 178

7 Conclusion 183

7.1 Directions for Future Research 185

Bibliography 187

List of Figures

1.1 A CSP example and its constraint graph. 13

1.2 A distributed binary CSP example: $V_i = \{v_1, v_2\}, C_i = \{R_{12}\}$, $V_j = \{v_3, v_4\}, C_j = \{R_{34}\}$, $V_w = \{v_7\}$, $V_p = \{v_6\}$, $V_k = \{v_5\}$, $V_q = \{v_8\}$, $V_l = \{v_9\}$, $C_{ij} = \{R_{23}, R_{24}\}$, $C_{ip} = \{R_{26}\}$, $C_{jk} = \{R_{45}\}$, $C_{wp} = \{R_{76}\}$, $C_{pk} = \{R_{65}\}$, $C_{kl} = \{R_{59}\}$ and $C_{kq} = \{R_{58}\}$. 14

1.3 Path-Consistency [13]. 16

1.4 A graph $G = (V, E)$. 18

1.5 A CRC constraint and a non-CRC constraint. 19

2.1 (a) A tree T. Subgraphs with node sets $\{a, b, c, d\}$, $\{a, b, c\}$, and $\{c, b, d\}$ form, respectively, a subtree, a chain, and a path of T. The degree of node b is 3 and the three branches of b are the subtrees with node sets $\{c\}, \{d\}$ and $\{a, e\}$ respectively. (b) A binary constraint R_{xy} between T_x and T_y, where a dashed arrow from a node u in T_x to a node v in T_y indicates that (u, v) is in R_{xy}. Node c is supported under R_{xy} with image $\{f, g, h\}$, and b is unsupported under R_{xy}. 28

2.2 (a) R_{xy} is a chain- but not path-preserving constraint as the the image of the path $\{c, b, d\}$ of T_x is $\{e, f, g, h\}$, which is not a path of T_y; (b) R_{xy} is a path- but not chain-preserving constraint as the image of the chain $\{a, c\}$ of T_x is $\{e, f, g\}$, which is not a chain of T_y; (c) R_{xy} is a tree-preserving but neither path- nor chain-preserving constraint as the image of the path $\{a, b\}$ of T_x, which is also a chain of T_x, is $\{e, f, g, h\}$ that is neither a path nor a chain of T_y. . . 29

2.3 M_{ab} is a trunk of tree T, i.e., the subtree induced by vertices with red colour. 34

2.4 The flow diagram of proof of Theorem 2.2. 35

2.5 A tree-preserving network $\mathcal{N} = \{R_{12}, R_{13}, R_{23}\}$ over forest domains F_1, F_2, and F_3, where each forest domain consists of three trees which contain only one node. 38

2.6 A triangulated graph G, where $\langle v_1, v_2, v_3, v_4, v_5\rangle$ is a perfect vertex elimination ordering of G. 39

2.7 Illustration of proof of Lemma 2.8. The figure is taken from [13]. . . . 40

2.8 Illustration of proof of Theorem 2.3. 42

2.9 Possible labelled line configurations of a junction in a picture and their corresponding forest structures. 48

2.10 A scene labelling instance and its corresponding constraint network. . 50

2.11 (a) R is a relation from tree domain T to T, and $R = \{(a, a), (b, a), (c, c)\}$. R is closed under the standard majority operation on T, but it is not tree-preserving w.r.t. T. (b) R is a relation from tree domain T to T, and $R = \{(a, a), (b, b), (c, b), (d, a)\}$. R is tree-preserving w.r.t. T, but it is not closed under the standard majority operation on T. 51

2.12 Illustration of proof of proposition 2.8. Broken arrow lines indicate images of the node and empty circles indicates empty support nodes. 52

2.13 Performance evaluation of different local consistency algorithms for solving tree-preserving constraint networks with different densities. We set $n = 100$, $d = 30$ and $l = 0.5$. 58

2.14 Illustration of proof of Lemma 2.7. 63

2.15 Possible configurations of two connected trunks $M_{a,b}$ and $M_{c,d}$, where $M_{a,b} \cup M_{c,d}$ is the whole tree in (c) and (d), a trunk in (a), (b) and (f), and a branch in (e). 66

2.16 Illustration of proof of Lemma 2.13. 67

3.1 A graph coloring problem with domain $D_i = \{red, blue\}$ for $i = 1, 2, 3, 4$ [37]. 76

3.2 Two binary constraint networks $\mathcal{N}$ and $\mathcal{N}_{-4}$. 78

3.3 A constraint graph that is a chain. 78

3.4 An illustration of Example 3.3. 80

3.5 Illustration of proof of Theorem 3.2. 84

3.6 A constraint network $\mathcal{N}$ and its elimination $\mathcal{N}_{-w}$. 88

3.7 Performance comparisons among **DPC***, **SAC3-SDS** and **PC2001** for solving tree-preserving constraint networks. 91

3.8 Performance comparisons among **DPC***, **SAC3-SDS** and **PC2001** for solving random majority-closed constraint networks. 92

4.1 A constraint graph of Example 4.1. 96

4.2 Illustration of proof of Theorem 4.1. 104

4.3 Execution of DΔCRC on Example 4.1. 107

4.4 Performance comparisons between DΔCRC and D-CRC. 112

4.5 Evaluation of our algorithm DΔCRC for different number of agents and network density. We set $n = 30$ and $d = 20$. 114

5.1 An illustration of Example 5.1. Alice, Bob, company X, company Y are four agents, each owning a local simple temporal network. The circles represent variables and edges constraints. Red edges represent constraints that are shared by two different agents. 117

5.2 Evaluation of ACSTP and P^{3}C. The y-axes (on the log scale) represent the number constraint checks. 133

5.3 Evaluation of DisACSTP and D$\triangle$PPC. The y-axis (on the log scale) represent the number of NCCCs. 134

6.1 The standard agent communication graph of the distributed binary CSP in Figure 1.2. 143

6.2 A T-tree for the standard agent communication graph in Figure 6.1. 144

6.3 The relevant parts of variable v_2 of the distributed binary CSP in Figure 1.2. 147

6.4 Illustrations of messages flow. 150

6.5 Time lines of sending and receiving messages, where circles stand for the time points that agent i sends "domain update" messages to agent j, squares stand for the time points that agent j reports the latest received time stamps of "domain update" messages to the root agent via "up to date" messages, and triangles stand for the time points that the root agent receives "up to date" messages from agent j. 152

6.6 A ternary constraint (s, R), where $s = (v_1, v_2, v_3)$ and $R = \{(a, c, e), (a, c, f), (b, d, f)\}$. 165

6.7 Performance comparisons between DisAC3.1 and DisAC9 on random instances. 172

6.8 Number of messages sent by agents w.r.t. height of spanning trees. We set $n = 10, d = 10, \rho = 1.0, \beta = 0.5$ and #agent = 10. 176

6.9 Speed-up vs. number of agents on benchmark problems. 177

List of Tables

3.1 Comparison of time complexities among state-of-the-art algorithms for solving majority-closed constraint networks. 90

6.1 Comparison of time and space complexities between DisAC3.1 and DisAC9. 173

6.2 Performance comparisons between DisAC3.1 and DisAC9 on the DOMINO problem. Note that m is the number of constraints and p is the number of agents used by both the algorithms. 174

6.3 Performance comparisons between DisAC3.1 and DisAC9 on the Radio Link Frequency Assignment Problem. Note that m is the number of constraints and p is the number of agents used by both the algorithms. 174

Abbreviation

(Dis)CSP - (Distributed) Constraint Satisfaction Problem

BCN - Binary Constraint Network

CRC - Connected Row-Convex

AC - Arc-Consistency

PC - Path-Consistency

DPC - Directional Path-Consistency

PPC - Partial Path-Consistency

STN - Simple Temporal Network

MaSTN - Multiagent Simple Temporal Network

ABSTRACT

Constraint propagation is central to the process of solving a constraint satisfaction problem (CSP). It can be used to solve several large tractable classes of CSPs directly and is also predominantly used to reduce the space of combinations that will be explored by a search algorithm. Constraint propagation, also known as local consistency enforcing, is the process of reducing domains of variables, strengthening constraints, or creating new ones. Arc-consistency (AC) and path-consistency (PC) are two well-known forms of local consistency. Designing efficient local consistency algorithms is a central research task in constraint processing. A related important question is to finding large tractable classes that can be solved by enforcing local consistency.

The class of connected row-convex (CRC) constraints defined over linear domains is a prominent tractable class which is a subclass of the class of row-convex constraints. While the class of row-convex constraints is intractable, it was shown that enforcing PC solves the CSPs over CRC constraints. The CRC constraint class is very expressive and can model problems in domains such as temporal reasoning, VLSI design, geometric reasoning, scene labelling as well as logical filtering.

In Chapter 2 we generalize the class of CRC constraints from linear domains to tree domains and obtain the new tractable class of tree-preserving constraints. We show that enforcing PC can transform a consistent tree-preserving constraint network into an equivalent globally consistent network. We also observe that CRC and tree-preserving constraint networks also can be solved by enforcing directional PC (DPC), a weaker form of PC which can be enforced more efficiently. A natural research question then is to characterize CSPs that are solvable with DPC. In Chapter 3 we provide such a characterization and prove that any class of majority-closed constraints is solvable with DPC and thus give a more efficient algorithm for solving these constraints.

In above, we assume that the knowledge about a CSP (i.e. domains and con-

straints) is known by one central agent, which is often not available when the knowledge about the problem is distributed among autonomous agents. Because of privacy reasons, simply collecting all such knowledge from the individual agents is undesirable or impossible. To address the issue, we need to develop distributed algorithms for solving distributed CSPs. We propose in Chapter 4 the first deterministic distributed algorithm to solve multiagent CRC constraint networks. Our algorithm is a distributed partial PC algorithm which can efficiently transform a CRC constraint network into an equivalent constraint network such that all constraints are minimal (i.e., they are the tightest constraints) and all solutions can be generated in a backtrack-free manner.

We then consider the class of simple temporal constraints in Chapter 5, which is closely related to the class of CRC constraints and is widely used in temporal planning and scheduling. In fact, discretized simple temporal constraints over finite domains are CRC constraints. Previous approaches focus on enforcing partial PC or directional PC to solve a simple temporal network (STN). We show that enforcing AC is sufficient to solve an STN, which not only provides a more efficient algorithm for STNs but also provides the first privacy-preserving distributed algorithm for solving multiagent STNs.

While the above algorithms are complete for certain tractable constraint classes, in Chapter 6 we propose a new distributed AC algorithm for general distributed CSPs, which is more efficient and leaks less private information of agents than existing ones. In particular, our new distributed AC algorithm uses a novel termination determination mechanism, which allows the agents to share domains, constraints and communication addresses only with relevant agents. We further extend it to the first distributed algorithm that enforces generalized AC (GAC) on k-ary ($k \geq 2$) distributed CSPs.

Chapter 1

Introduction

A *constraint satisfaction problem* (CSP) comprises a set of variables ranging over some domains of possible values, and a set of constraints that specify allowed value combinations for these variables. An instance of a CSP is also called a *constraint network*. Solving a constraint network amounts to assigning values to its variables such that its constraints are satisfied. CSP is widely used in Artificial Intelligence (AI). To name a few, CSP has been used to model problems in temporal planning [38, 122], vehicle routing [78], planning and scheduling [3] and spatial reasoning [84].

However, deciding the consistency of CSPs is NP-complete in general [64]. Therefore, developing efficient techniques to solve CSPs has always been an important research topic in AI. There are two research mainstreams to tackle the problem: one is to identify tractable classes that can be solved efficiently and the other is to develop effective general schemes to solve the problem. A number of tractable classes have been proposed in the literature. The class of *connected row-convex* (CRC) constraints proposed by Deville, Barette and van Hentenryck [40] is perhaps the most well-known one, which is a subclass of the class of *row-convex* constraints proposed by van Beek and Dechter [7]. The class of CRC constraints generalizes several tractable classes of constraints such as 2SAT, binary integer linear constraints and monotone constraints [40]. The CRC constraint class is very expressive and can model problems in domains such as temporal reasoning [73, 99], VLSI design [15], geometric reasoning [72], scene labelling [7] as well as logical filtering [76]. Efficient algorithms, such as the variable elimination algorithm by Zhang and Marisettis

[135], have been proposed to solve CRC constraints.

On the other hand, *backtracking search* [33, 129] is the principal mechanism for solving a general CSP; it assigns values to variables in a depth-first manner, and backtracks to the previous variable assignment if there are no consistent values for the variable at hand, and it needs only linear memory space. Lots of interests are in improving the efficiency of backtracking search. *Constraint propagation*, also known as *local consistency enforcing* [37], are in the core place of improving the efficiency of backtracking search. Local consistency techqunies are often integrated within the search to prune branches of the search tree that will lead the search to a dead end, and thus could greatly shrink the search space. The most well-known forms of local consistency are *arc- and path-consistency* [53, 113].

In contrast to the centralized setting, where a single CSP is solved by one agent, in multiagent setting different CSPs belong to different agents, where some of those problems are linked together through extra constraints. Such a *distributed constraint satisfaction problem* (DisCSP) cannot be solved by a centralized method, as we require a reliable protocol that can coordinate concurrent processes efficiently and preserve the privacy of individual agents to the greatest extent possible. Formally, a DisCSP consists of a set of local CSPs and a set of external constraints, where each local CSP is owned by an autonomous agent and each external constraint is shared by at least two different agents [128]. These agents aim to assign values to their own local variables cooperatively such that all constraints of the network are satisfied. DisCSPs can be used to model many combinatorial problems that are distributed by nature, e.g., distributed resource allocation problems [28], distributed scheduling problems [117], distributed interpretation tasks [94], multiagent truth maintenance tasks [59] and multiagent temporal reasoning [14]. The backtracking search was extended to *asynchronous backtracking* search for solving DisCSPs [128].

1.1 Related Works

1.1.1 Local Consistency Algorithms for CSPs

Arc-consistency (AC) is the most used way of propagating constraints. AC ensures that, for any value a in the domain of a variable x, we can find a value b in the domain of any other variable y such that the pair (a, b) is allowed by the constraint between x and y. Mackworth is the first to clearly define the concept of arc-consistency for binary constraints [89]. He also extended definitions and algorithms to non-binary constraints [90] and analyzed the complexity [91]. The first version of AC algorithm, known as AC1, was due to Fikes [46], but the algorithm is rather inefficient because a single revision of an arc in a particular iteration causes all the arcs to be revised in the next iteration whereas in fact only a small fraction of them could possibly be affected. AC2 [125] and AC3 [89] both improve AC1 by using a priority queue to record affected arcs and only affected arcs in the queue will be further considered and revised in the next iteration. The difference is that AC2 chooses a particular ordering of arcs to revise whereas AC3 does not, and thus the former is just a special case of the latter. AC3 was extended to *generalized AC* (GAC) in arbitrary networks in [90]. Mohr and Henderson proposed AC4 to improve the time complexity [53, 100] and showed that AC4 has an optimal worst-case time complexity. However, AC3 is often better than AC4 in practice, as AC4 almost always reaches its worst-case. Wallace discussed this issue in [123]. Bessiere proposed AC6 which keeps the optimal worst-case time complexity of AC4 and also has a better practical performance than AC3 [8]. Bessiere also proposed AC7 [11], which exploits the bidirectionality of value supports on constraints, but AC7 is slower than AC6 in general. It was later shown that AC3 can be made optimal by storing the smallest support for each value on each constraint, like AC6. The new algorithm is called AC3.1 [12], which is practically comparable to AC6 and is easier

to be implemented and integrated within search due to its simple data structure requirements.

Path-consistency (PC) is a higher order consistency notion than AC. PC was proposed by Montanari [101] for binary constraint networks to ensure that any consistent solution to a two-variable subnetwork is extendible to any third variable. Normally, PC can prune more inconsistent tuples and thus can produce a tighter network than AC. However, PC requires much higher time and space complexities than AC, and PC can produce additional constraints that were not in the network. Moreover, PC is usually considered only in binary constraint networks as its definition prevents its use on non-binary constraint networks. As such, although many PC algorithms have been proposed, like PC1 [101], PC2 [89], PC4 [100, 53], PC8 [24] and PC3.1 [12], they are seldom used in CSP solvers.

To alleviate the time and space requirements of enforcing PC, a weaker consistency notion of PC, *directional PC* (DPC), has been proposed by Dechter and Pearl [39]. DPC considers a given variable ordering and can thus be enforced more efficiently than PC. On the other hand, traditional PC is enforced on complete networks, Bliek and Sam-Haroud proposed to enforce PC on triangulated networks instead, which is called *partial PC* (PPC) [13]. Enforcing PPC can be much faster than PC for sparse networks, as far fewer new edges will be created.

1.1.2 Tractable Subclasses of CSPs

van Beek and Dechter proposed the class of *row-convex* constraints and showed that if a constraint network is *path-consistent* (PC) and each of its constraints is row-convex, then the network is *globally consistent* [7], in the sense that we can find a solution backtrack-free following an arbitrary variable ordering. However, enforcing PC on a row-convex constraint network may destroy its row-convexity. In fact, it was pointed out that deciding the consistency of row-convex constraint networks is NP-

complete [37]. Deville et al. then restricted the class of row-convex constraints and obtained the class of *connected row-convex* (CRC) constraints [40]. They showed that CRC constraints are closed under composition, intersection, and transposition, the basic operations of PC algorithms. This establishes that PC over CRC constraints produces a globally consistent network and is thus a polynomial-time decision procedure for CRC networks. Zhang and Marisetti proposed an efficient variable elimination algorithm to solve CRC networks [135].

In [137], Zhang and Yap generalized the class of row-convex constraints defined over linear domains to the class of *tree-convex* constraints defined over tree domains. Likewise, they also restricted the class of tree-convex constraints to obtain a tractable class of *locally chain convex and strictly union closed constraints*. However, it turns out that a locally chain convex and strictly union closed constraint network is just the disjoint union of several independent CRC constraint networks [68].

Jeavons et al. utilized the algebraic property to show that *strong PC*, namely, both AC and PC, is sufficient to ensure global consistency if and only if the class of binary constraints is *majority-closed* [62]. Notably, the class of CRC constraints is also majority-closed. Chen et al. further showed that *singleton arc-consistency* (SAC) is sufficient to decide the consistency of majority-closed constraint networks [23]. A recent breakthrough characterizes all CSPs that are solvable by local consistency enforcing methods: it shows that local consistency enforcing methods can be used to decide the consistency of a problem if and only if the problem does not have the *ability to count* [5] and any problem that can be decided by local consistency enforcing methods also can be decided by enforcing strong PC [4]. Kozik further showed that enforcing SAC solves the same family of problems that are solvable by enforcing strong PC [71]. However, it remains unclear whether backtrack-free search can be used to extract a solution for such a problem after enforcing strong PC or SAC.

1.1.3 Local Consistency Algorithms for DisCSPs

Several distributed algorithms for solving CRC constraints and simple temporal constraints have been proposed in the literature. Kumar *et al.* [75] proposed a distributed algorithm, called D-CRC, for solving CRC constraints, which was shown to be more efficient than the state-of-the-art centralized algorithm for CRC constraints. There are, however, several drawbacks of D-CRC: (i) it is based on randomization and as such it does not guarantee to return a solution even when the input CSP is consistent; and (ii) it cannot determine whether the input is consistent; and (iii) it cannot assign more than one variable to each agent, which makes the algorithm unrealistic to solve large networks in real distributed systems. Boerkoel and Durfee provided in [14] the extension of simple temporal network (STN) to multiagent STN (MaSTN) as well as a distributed algorithm, called D△PPC, for computing the complete joint solution space. However, as D△PPC is based on the $\mathsf{P}^3\mathsf{C}$ algorithm [104], which triangulates the input constraint graph, it has the drawback of creating new constraints between agents that are possibly not directly connected. These new constraints are undesirable, as they introduce constraints between two previously not directly connected agents and thus present a threat to the privacy of the relevant agents.

There are also several distributed AC algorithms proposed in the literature, including DisAC3 [6], DisAC4 [103] and DisAC6 [6], which are, respectively, the distributed versions of AC3, AC4 and AC6. Another distributed algorithm DisAC9 [52], which is also a distributed version of AC6, is currently the state-of-the-art. Although privacy is one main motivation and a major concern of solving DisCSPs [44, 50, 124, 131], no distributed AC algorithms so far have considered the communication address privacy of individual agents. Indeed, the distributed AC algorithms mentioned above either assume a complete agent communication graph, which reveals the communication address, thus the identity, of every agent, or broadcast

deleted values of variable domains, revealing the existence of variables and their domains. More precisely, the termination procedure of DisAC3, DisAC4, DisAC6 and DisAC9 assumes that the agent communication graph is complete, i.e., any two agents know the communication address of each other, which implies that they know the existence of each other and can directly send messages to each other. Also, whenever an agent deletes a value from one of its local domains, the agent broadcasts this information to all other agents immediately. This setting has the following drawbacks: (i) the algorithm may need to send unnecessarily many messages; (ii) the identities of agents and deleted domain values are revealed to irrelevant agents.

1.2 Thesis Outline and Contributions

This thesis extends the related works in the literature and explores centralized and distributed local consistency algorithms for solving CSPs in several dimensions. On the one hand, we explore centralized local consistency algorithms to identify more general tractable constraint classes and to solve tractable constraint classes more efficiently. On the other hand, we design more efficient distributed local consistency algorithms to solve tractable constraint subclasses and to filter inconsistent tuples for DisCSP solvers. The following paragraphs elaborate.

The first part, including Chapters 2 and 3, focuses on exploring constraint propagation algorithms for solving CSPs.

1.2.1 Tree-Preserving Constraints

Tree-convex constraints are generalizations of the well-known row-convex constraints from linear domains to tree domains. Chapter 2 studies three tractable subclasses of tree-convex constraints, which are chain-, path- and tree-preserving constraints. The chain-preserving constraints subsume the well-know CRC constraints studied in [40]. We prove that enforcing strong PC decides the consistency of a

tree-preserving constraint network and, if no inconsistency is detected, transforms the network into a globally consistent constraint network. Actually, we prove this by two methods. The first method directly proves that enforcing strong PC transforms a tree-preserving constraint network into a path-consistent tree-preserving network, while the second method relies on the characterization of tree-preserving constraints by closure under majority operations. Since every arc-consistent chain- or path-preserving constraint is a tree-preserving constraint, we get a tractable subclass of CSPs that is genuinely larger than the subclass of CRC constraints. We further show that partial PC algorithms can be applied to solve tree-preserving constraint networks in a backtrack-free style, which is more efficient than using a standard PC algorithm. As an application, we show that a large tractable subclass of the trihedral scene labelling problem can be modelled by tree-preserving constraints, and thus can be solved by the techniques discussed in this chapter.

The works of Chapter 2 are published in the following papers:

Shufeng Kong, Sanjiang Li, Yongming Li and Zhiguo Long: On tree-preserving constraints. In *Proceeding of the 21st International Conference on Principles and Practice of Constraint Programming* (CP'15), pp. 244-261 (2015)

Shufeng Kong, Sanjiang Li, Yongming Li and Zhiguo Long: On tree-preserving constraints. *Annals of Mathematics and Artificial Intelligence*, 81(3-4): 241-271 (2017)

1.2.2 CSPs Solvable with Directional PC

Chapter 3 investigates which CSPs can be solved by enforcing directional PC, a weaker form of PC that can be enforced more efficiently. Given a *complete binary constraint language*[1] Γ, it turns out that the directional PC algorithm by Dechter and Pearl [39], **DPC**, can solve the problems defined over Γ, **CSP**(Γ), if

[1]The concept will become clear in Chapter 3.

Γ is defined over domains with less than three values. Furthermore, we presented the algorithm **DPC***, a simple variant of **DPC**, which can solve the CSP of any majority-closed constraint language, and is sufficient for guaranteeing backtrack-free search for majority-closed constraint networks. Note that the subclass of tree-preserving constrains studied in Chapter 2 is also majority-closed. Our evaluations also show that **DPC*** significantly outperforms the state-of-the-art algorithms for solving majority-closed constraint networks.

The works of Chapter 3 are published in the following paper:

Shufeng Kong, Sanjiang Li and Michale Sioutis: Exploring directional path-consistency for solving constraint networks. *The Computer Journal*, first online: 27 December 2017. doi: 10.1093/comjnl/bxx122

The second part, including Chapters 4, 5 and 6, focuses on exploring constraint propagation algorithms for solving distributed CSPs.

1.2.3 A Distributed Partial PC Algorithm for Solving CRC Constraint Networks

As mentioned in Chapter 1.3, the class of CRC constraints generalizes several classes of constraints such as 2SAT, binary integer linear constraints, and monotone constraints [40], and is useful in modelling problems in domains such as VLSI design [15], scene labelling [7] as well as logical filtering [76]. Chapter 4 proposes the first deterministic distributed algorithm, called D∆CRC, for solving CRC constraints. The algorithm can efficiently transform an input CRC constraint network into an equivalent constraint network, where all constraints are minimal, and can generate all solutions in a backtrack-free manner. D∆CRC does not suffer from the problems that the state-of-the-art algorithm D-CRC has: (i) it is sound and complete and (ii) it can assign more than one variable to each agent, allowing the algorithm to solve large networks in real distributed systems. Furthermore, our theoretical and

experimental comparisons showed that DΔCRC significantly outperforms D-CRC.

The works of Chapter 4 are published in the following paper:

Shufeng Kong, Jae Hee Lee and Sanjiang Li: A deterministic distributed algorithm for reasoning with connected row-convex constraints. In *Proceeding of the 16th International Conference on Autonoumous Agent and Multiagent Systems* (AAMAS'17), pp. 203-211 (2017)

1.2.4 A Distributed AC Algorithm for Solving Multiagent STPs

The simple temporal constraints are widely used in temporal planning and scheduling. In fact, discretized simple temporal constraints are CRC constraints studied in Chapter 4. Chapter 5 presentes a novel AC-based approach for solving the simple temporal problems (STP) and the multiagent STPs. We show that enforcing AC is sufficient for solving an STP. Considering that STPs are defined over infinite domains, this result is rather surprising. Our empirical evaluations showed that the AC-based algorithms are significantly more efficient than their PC-based counterparts. This is mainly due to the fact that PC-based algorithms add many redundant constraints in the process of triangulation. More importantly, since our AC-based approach does not impose new constraints between agents that are previously not directly connected, it respects as much privacy of these agents as possible.

The works of Chapter 5 are published in the following paper:

Shufeng Kong, Jae Hee Lee and Sanjiang Li: Multiagent simple temporal problem: the arc-consistency approach. In *Proceeding of the 32th AAAI Conference on Artificial Intelligence* (AAAI'18), New Orleans, Louisiana, USA, February 2-7, 2018.

1.2.5 A New Distributed Generalized AC Algorithm

Chapter 6 presentes new distributed algorithms DisAC3.1 and DisGAC3.1 for efficient AC and GAC propagations. These algorithms do not assume a complete agent communication graph and release less private information of individual agents when enforcing AC and GAC. More precisely, an agent i only shares information about its communication address, its domain D_u, and its external constraint R_{uv} with another agent j, if the variable v is owned by j. Our theoretical analysis shows that our algorithms are efficient in both time and space. For problems that can be modelled as distributed CSPs, we have shown that DisAC3.1 and DisGAC3.1 are efficient algorithms that can serve as good candidates for search space pruning. Methods developed in this paper also can be adapted to solve the multi-agent STNs studied in Chapter 5.

The works of Chapter 6 are published in the following paper:

Shufeng Kong, Jae Hee Lee and Sanjiang Li: A new distributed algorithm for efficient generalized arc-consistency propagation. *Autonomous Agent and Multi-Agent Systems*, first online: 10 May 2018. doi: 10.1007/s10458-018-9388-x

1.3 Preliminaries

The remainder of this chapter introduces necessary concepts and notations that this work builds upon. Additional definitions, pertaining to individual tasks solved, are given at the beginning of corresponding chapters.

1.3.1 Constraint Satisfaction Problem

Definition 1.1 (constraint satisfaction problem)**.** *A* constraint satisfaction problem (CSP) $\mathcal{N}$ *is a triple* $\langle V, \mathcal{D}, C\rangle$*, where:*

- V *is a non-empty finite set of variables;*

- $\mathcal{D} = \{D_v \mid v \in V\}$ *is a collection of finite sets of values. We call* $D_v \in \mathcal{D}$ *the* domain *of variable* $v \in V$*;*
- C *is a finite set of pairs* (s, R)*, called* constraints*, where*
  - s*, which is called the* scope *of* (s, R)*, is a tuple of variables from* V*;*
  - R *is a relation defined over the variables in* s*, i.e., if* $s = (v_1, \ldots, v_l)$ *then* $R \subseteq D_{v_1} \times \cdots \times D_{v_l}$.

An instance of CSP is also called a constraint network*. With a slight abuse of notation, we will use the terms constraint network and CSP interchangeably in this dissertation.*

The arity *of a constraint* (s, R) *is defined as the cardinality of its scope* s*. A constraint with arity* k *is called a* k-ary *constraint and, in particular, a 2-ary constraint is also called a* binary *constraint. A CSP is called* k*-ary if it has a* k*-ary constraint but has no constraint of arity greater than* k*.*

Let $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$ *and* $\mathcal{N}' = \langle V', \mathcal{D}', C'\rangle$ *be CSPs. We say that* $\mathcal{N}'$ *is a* subnetwork *of* $\mathcal{N}$*, if* $V' = V$*,* $C' \subseteq C$ *and* $D'_v \subseteq D_v$ *for each* $v \in V$*, where* D'_v *and* D_v *are the domains of* v *in* $\mathcal{N}'$ *and* $\mathcal{N}$*, respectively.*

A *partial solution* of $\mathcal{N}$ w.r.t. a subset V' of V is an assignment of values to variables in V' such that all of the constraints (s, R) with $s \subseteq V'$ are satisfied. A partial solution w.r.t. V is called a *solution* of $\mathcal{N}$. We write $sol(\mathcal{N})$ for the set of solutions of $\mathcal{N}$ and say two CSPs $\mathcal{N}$ and $\mathcal{N}'$ are equivalent, if $sol(\mathcal{N}) = sol(\mathcal{N}')$. We say that $\mathcal{N}$ is *consistent* if it has a solution, i.e., $sol(\mathcal{N}) \neq \varnothing$, and *inconsistent*, otherwise. We say that $\mathcal{N}$ is *empty* if at least one of its domains or relations is empty; if $\mathcal{N}$ is empty, then it is trivially inconsistent. Figure 1.1 illustrates a CSP, where the assignment $(v_1 = a, v_2 = e, v_3 = c)$ is a solution.

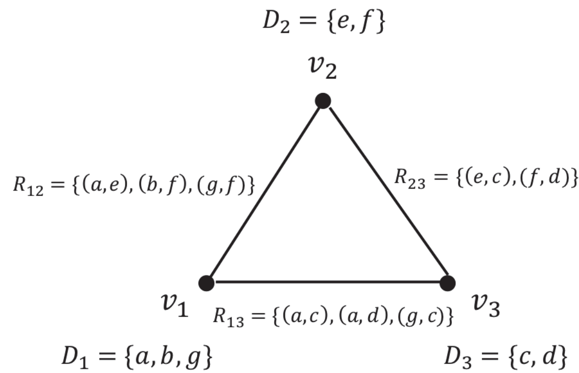


Figure 1.1 : A CSP example and its constraint graph.

Definition 1.2 (constraint graph). *Given a binary CSP $\mathcal{N} = \langle V, D, C \rangle$, the* constraint graph $G_{\mathcal{N}}$ *of $\mathcal{N}$ is a pair (V, E), where E is a set of undirected edges over V, such that there is an edge e_{ij} between v_i and v_j in E if and only if $((v_i, v_j), R_{ij}) \in C$. $G_{\mathcal{N}}$ can be make complete or triangulated by adding missing edges and for every newly added edge, say e_{kl} between v_k and v_l, we also add a related universal constraint to C, namely $((v_k, v_l), R_{kl} = D_k \times D_l)$.*

A binary CSP and its constraint graph are given in Figure 1.1.

Let $\mathcal{N} = \langle V, D, C \rangle$ be a binary CSP. We assume that for any pair of variables (v, w), there exists at most one constraint from v to w in C. We write this constraint as $((v, w), R_{vw})$ if it exists, and write

$$R_{vw}^{-1} = \{(b, a) \mid (a, b) \in R_{vw}\}.$$

for the *inverse* of R_{vw}. We also assume that $((w, v), R_{wv})$ with $R_{wv} = R_{vw}^{-1}$ is in C, if $((v, w), R_{vw})$ is in C. For brevity, we often write R_{vw} for the constraint $((v, w), R_{vw})$. The usual operations on relations, e.g., intersection ($\cap$), and composition ($\circ$), are applicable to constraints.

Given a binary CSP $\mathcal{N} = \langle V, D, C \rangle$ and two variables $v, w \in V$, the *image* of

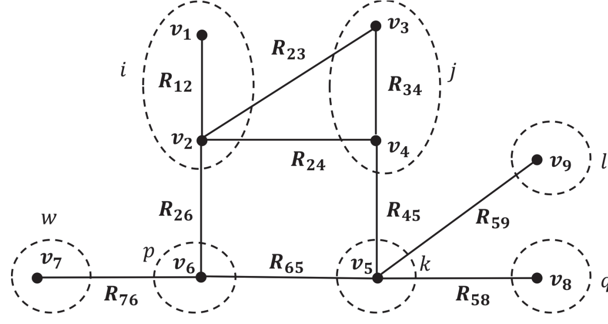


Figure 1.2 : A distributed binary CSP example: $V_i = \{v_1, v_2\}, C_i = \{R_{12}\}$, $V_j = \{v_3, v_4\}, C_j = \{R_{34}\}$, $V_w = \{v_7\}$, $V_p = \{v_6\}$, $V_k = \{v_5\}$, $V_q = \{v_8\}$, $V_l = \{v_9\}$, $C_{ij} = \{R_{23}, R_{24}\}$, $C_{ip} = \{R_{26}\}$, $C_{jk} = \{R_{45}\}$, $C_{wp} = \{R_{76}\}$, $C_{pk} = \{R_{65}\}$, $C_{kl} = \{R_{59}\}$ and $C_{kq} = \{R_{58}\}$.

$a \in D_v$ under R_{vw}, denoted as $R_{vw}(a)$, is the set $\{b \in D_w \mid (a, b) \in R_{vw}\}$. Each value b in $R_{vw}(a)$ is called a *support* of a on R_{vw}. We say a subset F of D_x is *unsupported* if every value in F is not supported. Given $A \subseteq D_v$, the *image* of A under R_{vw} is defined as $R_{vw}(A) = \{b \in D_w \mid \exists a \in A, (a, b) \in R_{vw}\}$. In Figure 1.1 the set of supports of $a \in D_1$ on R_{13} is $\{c, d\}$.

Definition 1.3 (distributed binary CSP)**.** *A* distributed binary CSP *is defined as a pair* $\langle \mathcal{P}, C^X \rangle$*, where*

- $\mathcal{P} = \{\mathcal{N}_1, \ldots, \mathcal{N}_p\}$ *is a set of binary CSPs with* $\mathcal{N}_i = \langle V_i, \mathcal{D}_i, C_i \rangle$ $(1 \leq i \leq p)$*;*
- $C^X = \bigcup\{C_{ij} \mid 1 \leq i, j \leq p, i \neq j\}$ *is a set of* external *constraints, where each* C_{ij} *is a set of constraints from some variable in* V_i *to some variable in* V_j*. We assume that* C_{ji} *consists of the inverses of the constraints in* C_{ij} *for all* $1 \leq i, j \leq p, i \neq j$*.*

We call $v \in V_i$ *a* shared *variable of agent* i *if it is connected to a variable* $w \in V_j$ $(j \neq i)$ *through an external constraint, and call* v *a* private *variable of agent* i *otherwise. A constraint* R_{vw} *is called a private constraint of agent* i *if* $R_{vw} \in C_i$*. We call* R_{vw} *a shared constraint of agent* i *if either* v *or* w *but not both belong to* V_i*.*

The constraint graph of a given distributed binary CSP is defined similarly to binary CSP. An example of a distributed binary CSP is given in Figure 1.2. We note that in Definition 1.3 each binary CSP $\mathcal{N}_i$ corresponds to an agent i and that C_{ij} is the set of constraints shared between agents i and j. We call agent j a *neighbor* of agent i if there is an external constraint between them, i.e., $C_{ij} \neq \varnothing$.

1.3.2 Local Consistency and Properties of Constraint Networks

Definition 1.4 (arc-consistency)**.** *Given a binary CSP $\mathcal{N} = \langle V, D, C\rangle$ and $v, w \in V$, we say that there is an* arc (v, w) *from v to w iff $R_{vw} \in C$. A constraint R_{vw} is* arc-consistent *(AC) iff for every $a \in D_v$, there is a support of a on R_{vw}, i.e., $R_{vw}(a) \neq \varnothing$. We say that $\mathcal{N}$ is AC iff every constraint in C is AC.*

In Figure 1.1 R_{13} is not AC because $b \in D_1$ has no support on R_{13}, i.e., $R_{13}(b) = \varnothing$. On the other hand, R_{12} is AC because $R_{12}(a)$, $R_{12}(b)$ and $R_{12}(g)$ are all nonempty.

Definition 1.5 (AC-closure)**.** *Given a binary CSP $\mathcal{N} = \langle V, D, C\rangle$, let $\mathcal{N}' = \langle V, \mathcal{D}', C'\rangle$ be a subnetwork of $\mathcal{N}$. We call $\mathcal{N}'$ an AC-subnetwork of $\mathcal{N}$, if $C' = C$ and $\mathcal{N}'$ is arc-consistent and not empty. We call $\mathcal{N}'$ the* AC-closure *of $\mathcal{N}$, if $\mathcal{N}'$ is the* largest *AC-subnetwork of $\mathcal{N}$ that is equivalent to $\mathcal{N}$, in the sense that every other AC-subnetwork $\mathcal{N}'' = \langle V, \mathcal{D}'', C\rangle$ of $\mathcal{N}$ that is equivalent to $\mathcal{N}$ is a subnetwork of $\mathcal{N}'$.*

For every binary CSP $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$, since the AC-subnetworks we considered in the above definition are different from $\mathcal{N}$ only in variable domains, we will identify the AC-subnetwork with its variable domains when the context is clear (e.g., $\mathcal{D}'$ for the AC-subnetwork $\mathcal{N}' = \langle V, \mathcal{D}', C\rangle$ of $\mathcal{N}$). Consider for example the binary CSP in Figure 1.1. We have that $\{D'_{v_1} = \{a\}, D'_{v_2} = \{e\}, D'_{v_3} = \{c\}\}$ is an AC-subnetwork of $\mathcal{N}$, while $\{D''_{v_1} = \{a, g\}, D''_{v_2} = \{e, f\}, D''_{v_3} = \{c, d\}\}$ is the AC-closure of $\mathcal{N}$.

The definition of arc-consistency, AC-subnetwork and AC-closure of binary CSPs naturally carry over to distributed binary CSPs. The AC-closure of a (distributed)

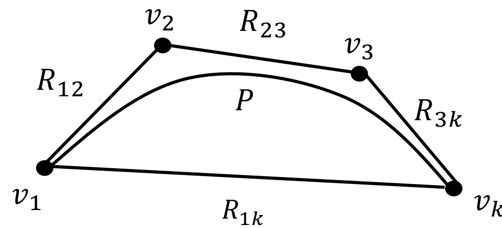


Figure 1.3 : Path-Consistency [13].

binary CSP $\mathcal{N}$ has the same solution set as $\mathcal{N}$, but its search space is smaller, which often significantly facilitates the search for a solution.

Definition 1.6 ((partial) path-consistency). *[13, 89] Let $\mathcal{N} = \langle V, D, C \rangle$ be a constraint network and $G_{\mathcal{N}}$ its constraint graph. Let $\pi_{ij} = (v_i = u_0, u_1, \ldots, u_k = v_j)$ be a path in $G_{\mathcal{N}}$ with $e_{ij} \in E(\mathcal{N})$. We say π_{ij} is* path-consistent *(PC) if, for every $\langle c_0, c_k \rangle \in R_{ij}$, we can find values for all intermediate variables u_x $(0 < x < k)$ s.t. all the constraints $R_{u_x,u_{x+1}}$ $(0 \leq x < k)$ are satisfied. We say $G_{\mathcal{N}}$ is partial path-consistent (PPC) if every path π_{ij} in $G_{\mathcal{N}}$ with $e_{ij} \in E(\mathcal{N})$ is PC. Moreover, we say $\mathcal{N}$ is PC (resp. PPC) if the completion of $G_{\mathcal{N}}$ (resp. $G_{\mathcal{N}}$) is PPC.*

Therefore, PC is a special case of PPC. We say a constraint network is *strong* PC (PPC, resp.) if it is both AC and PC (PPC, resp.). An illustration of PC is given in Figure 1.3.

Arc- and path-consistency are further generalized to k-consistency defined below.

Definition 1.7 (k-consistency). *[47, 48] A constraint network $\mathcal{N}$ over n variables is* k-consistent *if any consistent instantiation of any distinct $k-1$ variables can be consistently extended to any k-th variable. We say $\mathcal{N}$ is* strongly k-consistent *if it is j-consistent for all $j \leq k$; and say $\mathcal{N}$ is* globally consistent *if it is strongly n-consistent. 2- and 3-consistency are exactly* arc-consistency *(AC) and* path-consistency *(PC) respectively.*

Definition 1.8 (decomposability). *[37, 101] Let $\mathcal{N} = \langle V, D, C \rangle$ be a constraint*

network and $\prec = (v_1, \ldots, v_n)$ *an ordering of variables in* V*. Write* $V_{\leq i} = \{v_j \mid j \leq i\}$*. We say that* $\mathcal{N}$ *is* decomposable *or* backtrack-free *w.r.t.* $\prec$ *if for any* $i < n$*, any consistent assignment to* $V_{\leq i}$ *can be consistently extended to an assignment to* $V_{\leq i+1}$*. We say* $\mathcal{N}$ *is* decomposable *if it is decomposable w.r.t. every ordering of variables in* V*.*

It is easy to see that a network being decomposable and globally consistent are equivalent.

Definition 1.9 (minimality)**.** *[101] Let* $\mathcal{N} = \langle V, D, C \rangle$ *be a constraint network. We say a non-empty constraint* $R_{ij} \in C$ *is* minimal *if any assignment* $\langle a_i, a_j \rangle \in R_{ij}$ *to variables* v_i, v_j *can be extended to a solution of* $\mathcal{N}$*. We say* $\mathcal{N}$ *is minimal if it has a complete constraint graph and every constraint in* $\mathcal{N}$ *is minimal.*

Decomposable network is always minimal and a constraint network that admits a minimal constraint is always consistent.

1.3.3 Triangulated Constraint Networks

Triangulated graphs play a key role in efficiently solving large sparse constraint networks [13, 87, 104, 127]. An undirected graph $G = (V, E)$ is said to be *triangulated* or *chordal* if every cycle of length greater than 3 has a chord, i.e., an edge connecting two non-consecutive vertices of the cycle. A network $\mathcal{N}$ is said to be triangulated (resp. complete) if $G_{\mathcal{N}}$ is triangulated (resp. complete).

Triangulated constraint graphs have the following nice property.

Theorem 1.1. [13] *A triangulated constraint graph is PPC if every path of length 2 is PC.*

If a constraint graph is not triangulated, we may add new edges (labeled with universal constraints) to make it triangulated. In the following, we give a characterization of triangulated graphs.

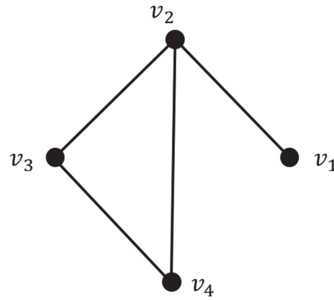


Figure 1.4 : A graph $G = (V, E)$.

Definition 1.10 (perfect vertex elimination ordering). *[67] An ordering $\prec$ in a graph $G = (V, E)$ is called a* perfect vertex elimination ordering *in G, if the set of successors of v*

$$F_v := \{w \mid (v, w) \in E, v \prec w\}$$

induces a complete subgraph of G for all $v \in V$.

For example, consider the graph $G = (V, E)$ in Figure 1.4, the ordering (v_1, v_2, v_3, v_4) is a perfect vertex elimination ordering in G, whereas the ordering (v_2, v_1, v_3, v_4) is not.

Proposition 1.1. [67] *A graph G is triangulated iff there exists a perfect vertex elimination ordering in G.*

1.3.4 Connected Row-Convex Constraint Networks

Suppose $R \subseteq D_1 \times D_2$ is a binary relation. The *(0,1)-matrix representation* of a relation R consists of $|D_1|$ rows and $|D_2|$ columns subject to orderings imposed on D_1, D_2. The entry in the i-th row and j-th column of the matrix is 1, if the corresponding pair of values from $D_1 \times D_2$ is in R, and is 0 otherwise.

Definition 1.11. *[40] A binary relation R, represented as a (0,1)-matrix, is* row-convex *if in each row all of the '1's are consecutive. We say R is a* connected row-convex *(CRC) relation if, after removing empty rows and columns in the matrix of*

R, (i) both R and its transpose are row-convex, and (ii) the positions of the '1's in any two consecutive rows or columns intersect, or are consecutive

Figure 1.5 shows two row-convex constraints, where the left one is CRC but the right one is not.

			1	1	1
			1	1	
		1			
	1				
1	1				
	1				

(a) A CRC constraint.

	1				
1	1	1			
	1	1	1		
		1			
		1	1	1	1
				1	1

(b) A non-CRC constraint.

Figure 1.5 : A CRC constraint and a non-CRC constraint.

Row convex constraints are not closed under intersection and composition [40]—the two operations to enforce PC, but CRC constraints are closed under these operations.

Lemma 1.1. [40] *CRC constraints are closed under intersection and composition.*

Row-convex constraint networks have the following property.

Lemma 1.2. [7] *A row-convex constraint network is decomposable and minimal if it is strong PC.*

The following theorem shows that CRC networks can be solved with PC, which follows directly from the above two lemmas.

Theorem 1.2. *Enforcing strong PC on a CRC constraint network transforms it into an equivalent decomposable and minimal constraint network, if no inconsistency is detected.*

The following lemma shows that enforcing PPC on a triangulation of a CRC network has the same effect as enforcing PC as far as only edges in the triangulation are concerned.

Lemma 1.3. [13] *Let $\mathcal{N}$ be a CRC constraint network. Suppose $G \supseteq G_{\mathcal{N}}$ is a triangulation of $G_{\mathcal{N}}$ and G^* is the completion of $G_{\mathcal{N}}$. Further, let $\mathcal{N}^{\triangle} = \langle V, D, C^{\triangle} \rangle$ and $\mathcal{N}^* = \langle V, D, C^* \rangle$ be the constraint networks obtained by enforcing PPC on $\mathcal{N}$ w.r.t G and G^*, respectively. Then $C^{\triangle} \subseteq C^*$.*

As a corollary of Lemmas 1.2 and 1.3, we have the following result, which shows that CRC networks can also solved with PPC.

Corollary 1.1. *Let $\mathcal{N}$ be a CRC constraint network and G a triangulation of $G_{\mathcal{N}}$. Suppose $\mathcal{N}' = \langle V, D, C' \rangle$ is the network obtained by enforcing PPC on $\mathcal{N}$ w.r.t. G. If no inconsistency is detected, then all constraints in C' are minimal.*

Proof. Suppose $\mathcal{N}$ is consistent. Let $\mathcal{N}^* = \langle V, D, C^* \rangle$ be the path-consistent constraint network that is equivalent to $\mathcal{N}$, i.e., $\mathcal{N}^*$ is obtained by enforcing PPC on $\mathcal{N}$ w.r.t. the completion of $G_{\mathcal{N}}$. By Lemma 1.2, $\mathcal{N}^*$ is decomposable and minimal. Further, by Lemma 1.3, we have $C' \subseteq C^*$, which finishes our proof. □

By Lemma 1.3 and Theorem 1.1, enforcing PPC on a CRC constraint network $\mathcal{N}$ can be done by the following two steps: (i) obtain a triangulation G of $G_{\mathcal{N}}$ by adding edges labeled with universal constraints, and (ii) make every path of length two in G path-consistent by using relational intersection and composition.

1.3.5 Algebraic Closure Properties of Constraints

This subsection recalls some notions and results about constraint closure properties from [17] and [61]. A constraint relation (or an operation) is called *one-sorted* if it is defined over a single domain and *multi-sorted* otherwise.

Definition 1.12. *Suppose $f^D : D^r \to D$ is a one-sorted operation on D. We say*

f^D *is* near-unanimity *if*

$$(\forall a, b \in D_i) \quad f^D(b, a, \ldots, a) = f^D(a, b, a, \ldots, a) = \ldots = f^D(a, \ldots, a, b) = a. \quad (1.1)$$

A ternary ($r = 3$) near-unanimity operation is called a *majority* operation.

Definition 1.13. *Given a collection of finite sets* $\mathcal{D} = \{D_1, \ldots, D_n\}$*, an* r-ary multi-sorted operation f *is a collection of one-sorted operations* $\{f^{D_1}, \ldots, f^{D_n}\}$*, where* $f^{D_i} : D_i^r \to D_i$*. Let* $R \subseteq D_{i_1} \times D_{i_2} \times \ldots \times D_{i_m}$ *be a multi-sorted relation over* $\mathcal{D}$*.* R *is* closed under f *if for any* r m*-tuples* $\langle d_{11}, d_{21}, \ldots, d_{m1}\rangle$, $\langle d_{12}, d_{22}, \ldots, d_{m2}\rangle$, …, $\langle d_{1r}, d_{2r}, \ldots, d_{mr}\rangle$ *in* R*, we have*

$$\langle f^{D_{i_1}}(d_{11}, d_{12}, \ldots, d_{1r}), \ldots, f^{D_{i_m}}(d_{m1}, d_{m2}, \ldots, d_{mr})\rangle \in R$$

Intuitively, a relation R is r-decomposable if it is representable by a r-ary constraint network, i.e., there exists a r-ary constraint network such that its set of solutions is exactly R. A formal definition is given below.

Definition 1.14. *Let* $\mathcal{D} = \{D_1, \ldots, D_n\}$ *be a set of domains. An* n*-ary relation* R *over* $\mathcal{D}$ *is a subset of* $D_1 \times \ldots \times D_n$*. For any tuple* $t \in R$ *and any* $1 \leq i \leq n$*, we denote by* $t[i]$ *the value in the* i*-th coordinate position of* t *and write* t *as* $\langle t[1], \ldots, t[n]\rangle$*.*

Definition 1.15. *Let* R *be an* m*-ary relation over domain* $\mathcal{D} = \{D_1, \ldots, D_n\}$*. We say* R *is* r-decomposable *if, for any* m*-tuples* t *and any* $I = (i_1, \ldots, i_k)$ *(a list of indices chosen from* $\{1, \ldots, m\}$*) with* $k \leq r$*, we have* $t \in R$ *if* $\pi_I(t) \in \pi_I(R)$*, where* $\pi_I(t) = \langle t[i_1], ..., t[i_k]\rangle$ *and* $\pi_I(R) = \{\pi_I(t') \mid t' \in R\}$*.*

Definition 1.16. *For a set of relations* Γ*, we write* Γ^+ *for the set of all relations which can be obtained from* Γ *by using some sequence of* Cartesian product *(for* $R_1, R_2 \in \Gamma, R_1 \times R_2 = \{\langle t_1, t_2\rangle \mid t_1 \in R_1, t_2 \in R_2\}$*),* equality selection *(for* $R \in$

$\Gamma, \sigma_{i=j}(R) = \{t \in R \mid t[i] = t[j]\}$ *), and* projection *(for* $R \in \Gamma, \pi_{i_1,\cdots,i_k}(R) = \{\langle t[i_1],\cdots,t[i_k]\rangle \mid t \in R\}$*).*

We denote by C_Γ the set of constraint networks all relations of which are taken from Γ. We have the following extension of [61, Theorem 3.5] from the one-sorted case to the multi-sorted case. The proof is similar to the one-sorted case and thus omitted.

Theorem 1.3. *Suppose* Γ *is a set of multi-sorted relations over a collection of finite sets* $\mathcal{D} = \{D_1, \ldots, D_n\}$*. For any* $r \geq 3$*, the following conditions are equivalent:*

1. *There exists an* r*-ary near unanimity operation* f^{D_i} *on* D_i *for each* $1 \leq i \leq n$ *such that every relation* R *in* Γ *is closed under the multi-sorted operation* $f = \{f^{D_1}, \ldots, f^{D_n}\}$*.*
2. *Every* R *in* Γ^+ *is* $(r-1)$*-decomposable.*
3. *For every constraint network* $\mathcal{N} \in C_\Gamma$*, establishing strong* r*-consistency ensures global consistency.*

Chapter 2

Tree-Preserving Constraints

2.1 Contribution

Tree-convex constraints are extensions of the well-known row-convex constraints. Just like the latter, every path-consistent tree-convex constraint network is globally consistent. However, it is NP-complete to decide whether a tree-convex constraint network has solutions. This chapter studies and compares three subclasses of tree-convex constraints, which are called chain-, path-, and tree-preserving constraints respectively.

The class of tree-preserving constraints strictly contains the subclasses of path-preserving and arc-consistent chain-preserving constraints. We prove that, when enforcing strong path-consistency on a tree-preserving constraint network, in each step, the network remains tree-preserving. This ensures global consistency of consistent tree-preserving networks after enforcing strong path-consistency, and also guarantees applicability of the partial path-consistency algorithms to tree-preserving constraint networks, which is usually much more efficient than the path-consistency algorithms for large sparse constraint networks. As an application, we show that the class of tree-preserving constraints is useful in solving the scene labelling problem.

2.2 Introduction and Chapter Outline

Since deciding the consistency of CSP instances is NP-complete in general, lots of efforts have been devoted to identifying tractable subclasses. There are two main approaches for constructing tractable subclasses. The first approach is *structural-based*,

in which tractable subclasses are obtained by restricting the topology of the underlying graph of the constraint network (being a tree or having treewidth bounded by a constant [37]); the second approach is *language-based*, in which tractable subclasses are obtained by by restricting the type of the allowed constraints between variables (cf. [7]). Recently, researchers also propose a hybrid approach for constructing tractable classes, see e.g., the subclass of CSP instances satisfying the *broken-triangle property* (BTP) [30, 31].

In this chapter, we are mainly interested in the language-based tractable subclasses. Montanari [101] showed that path-consistency is sufficient to guarantee that a constraint network is globally consistent if the relations are all *monotone*. Van Beek and Dechter [7] generalized monotone constraints to *row-convex* constraints, which are further generalized to *tree-convex* constraints by Zhang and Yap [137]. These constraints also have the nice property that every path-consistent constraint network is globally consistent.

However, neither row-convex constraints nor tree-convex constraints are closed under intersection, which is one of the main operations of *path-consistency* (PC) algorithms. This means that enforcing path-consistency may destroy row and tree-convexity. Deville et al. [40] proposed a tractable subclass of row-convex constraints, called *connected row-convex* (CRC) constraints, which are closed under composition and intersection. Zhang and Freuder [134] also identified a tractable subclass of tree-convex constraints, called *locally chain convex and strictly union closed* constraints. They also proposed the important notion of *consecutive* constraints. Kumar [74] showed that the subclass of arc-consistent consecutive tree-convex (ACCTC) constraints is tractable by providing a polynomial time randomised algorithm. Nevertheless, for the ACCTC problems, "*it is not known whether there are efficient deterministic algorithms, neither is it known whether strong path-consistency ensures global consistency on those problems.*"[134]

In this chapter, we study and compare three subclasses of tree-convex constraints, which are called, respectively, chain-, path- and tree-preserving constraints.

In Section 2.3, we start with basic notations and concepts that will be used throughout the chapter. Based on the concept of tree domains, we introduce chain-, path- and tree-preserving constraints. Chain-preserving constraints are exactly "locally chain convex and strictly union closed" constraints in the sense of [134], which include CRC constraints as a special case where tree domains are linear. Arc-consistent chain-preserving constraints, path-preserving constraints and AC-CTC constraints are all strictly contained in the class of tree-preserving constraints.

Therefore, the remainder of this chapter will focus on the more general tree-preserving constraints. We show in Section 2.4 that the class of tree-preserving constrains is closed under intersection and composition, which are operations of the path-consistency algorithm. This guarantees that a tree-preserving constraint network remains tree-preserving after enforcing path-consistency on it. Recall that every path-consistent tree-convex constraint network is globally consistent [137]. This shows that the class of tree-preserving constraints is tractable and can be solved by the path-consistency algorithm. We also prove in this section that our definitions and results for tree-preserving constraints can be extended to domains with acyclic graph structures, called *forest domains* in this chapter.

The above properties of tree-preserving constraints bear similarity to CRC constraints. Bliek and Sam-Haround [13] showed that enforcing *partial path-consistency* (PPC) is sufficient to solve sparse CRC constraint networks. PPC enforces PC on sparse constraint graphs by triangulating instead of completing them and thus can be enforced more efficiently than enforcing PC. As far as CRC constraints are concerned, the pruning capacity of path-consistency on triangulated graphs and their completion are identical on the common edges. In Section 2.5, we show that PPC

[13] is sufficient to decide the consistency of tree-preserving constraint networks. Moreover, we show that, after enforcing PPC, we can find a solution in a backtrack-free style if no inconsistency is detected.

Section 2.6 is concerned with the application of tree-preserving constraints in scene labelling. Solving the scene labelling problem is a crucial part of figuring out the possible 3D scenes of a 2D projection, which has applications in both vision and geometric modelling. Research in this field has centred on the trihedral scene labelling problem, i.e., scenes where no four planes share a point. The trihedral scene labelling problem has been shown to be NP-complete [66]. Based on the forest domains associated to each possible variable type by Zhang and Freuder [134] (see Figure 2.9), we show that all 39 possible types of the trihedral scene labelling problem instances are tree-convex, and 29 of them are tree-preserving. This means that a large subclass of the trihedral scene labelling problem can be modelled by tree-preserving constraint networks and thus can be efficiently solved by the techniques discussed in this chapter. As a byproduct, since every instance of the NP-complete trihedral scene labelling problem can be modelled by a tree-convex constraint network, we show that the class of tree-convex constraints is NP-complete.

It is interesting to compare our approach with another research line of studying tractable subclasses of CSPs, which focuses on the algebraic closure property of constraints [17, 45, 61]. In Section 2.7, we study the algebraic closure property of tree-preserving constraints and establish the equivalence between tree-preserving constraints and constraints that are closed under a "standard" majority operation. In this way, we provide an alternative way to prove the tractability of tree-preserving constraints.

Section 2.8 reports experimental evaluations on enforcing PPC and PC on tree-preserving constraint networks and Section 2.9 concludes the chapter.

2.3 Preliminaries

Necessary notations and results about CSP, AC, PC and PPC are provided in section 1.3.

Let D be the domain of a variable x. An undirected graph structure can often be associated to D such that there is a bijection between the vertices in the graph and the values in D. If the graph is connected and acyclic, i.e., a tree, then we say it is a *tree domain* of x. Tree domains arise naturally in e.g., scene labelling [134] and combinatorial auctions [27]. We note that, in this chapter, we fix a specific tree domain for each variable x.

In this chapter, we distinguish between trees and rooted trees. Standard notions from graph theory are assumed. In particular, the *degree* of a node a in a graph G, denoted by $\deg(a)$, is the number of neighbours of a in G. A node a is called a *leaf node* if it has only one neighbour, i.e., $\deg(a) = 1$.

Definition 2.1. *A* tree *is a connected graph without any cycle (cf. Figure 2.1(a)). A tree is* rooted *if it has a specified node r, called the* root *of the tree. Given a tree T, a subgraph I is called a* subtree *of T if I is connected. The empty subgraph is a subtree of any tree.*

Let T be a tree (rooted tree, resp.) and I a subtree of T. I is a path *(*chain*, resp.) in T if each node in I has at most two neighbours (at most one child, resp.) in I. Given two nodes p, q in T, the unique path that connects p to q is denoted by $\pi_{p,q}$.*

Suppose a is a node of a tree T. A branch *of a is a connected component of $T \setminus \{a\}$.*

Figure 2.1 gives illustrations of these notions.

Throughout this chapter, we always associate a tree structure $T_x = (D_x, E_x)$

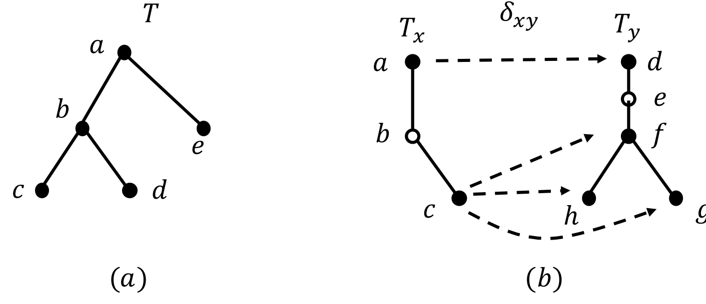


Figure 2.1 : (a) A tree T. Subgraphs with node sets $\{a, b, c, d\}$, $\{a, b, c\}$, and $\{c, b, d\}$ form, respectively, a subtree, a chain, and a path of T. The degree of node b is 3 and the three branches of b are the subtrees with node sets $\{c\}, \{d\}$ and $\{a, e\}$ respectively. (b) A binary constraint R_{xy} between T_x and T_y, where a dashed arrow from a node u in T_x to a node v in T_y indicates that (u, v) is in R_{xy}. Node c is supported under R_{xy} with image $\{f, g, h\}$, and b is unsupported under R_{xy}.

with a given domain D_x, where E_x is the set of tree edges connecting values in D_x. For convenience, we often use the notation T_x to denote the domain D_x and call T_x a tree domain. Also, $a \in T_x$ means that $a \in D_x$.

In this chapter, unless stated otherwise, *we assume that R_{xy} is the inverse of R_{yx}*, and if there is no constraint for (x, y), we assume that R_{xy} is the universal constraint (i.e., $D_x \times D_y$).

Definition 2.2. *Let x, y be two variables with finite tree domains $T_x = (D_x, E_x)$ and $T_y = (D_y, E_y)$, and R a constraint from x to y. We say R, w.r.t. T_x and T_y, is (cf. Figure 2.2)*

- tree-convex *if the image of every value a in D_x (i.e., $R(a)$) is a subtree of T_y;*
- consecutive *if the image of every edge in T_x is a subtree in T_y;*
- path-preserving *if the image of every path in T_x is a path in T_y;*
- tree-preserving *if the image of every subtree in T_x is a subtree in T_y.*

In case T_x and T_y are rooted, we say R, w.r.t. T_x and T_y, is

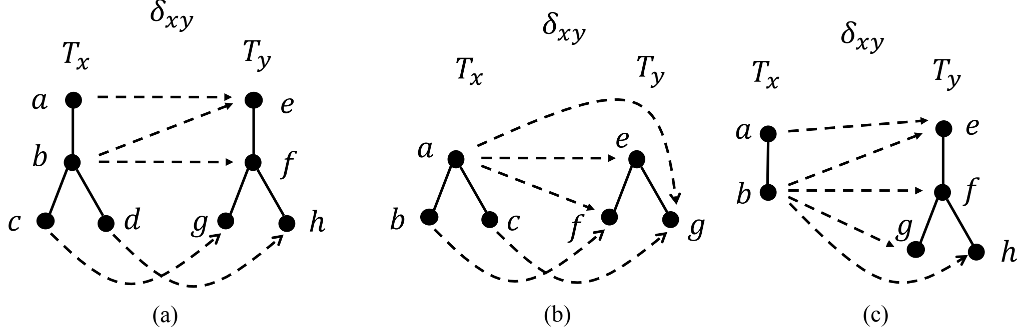


Figure 2.2 : (a) R_{xy} is a chain- but not path-preserving constraint as the the image of the path $\{c, b, d\}$ of T_x is $\{e, f, g, h\}$, which is not a path of T_y; (b) R_{xy} is a path- but not chain-preserving constraint as the image of the chain $\{a, c\}$ of T_x is $\{e, f, g\}$, which is not a chain of T_y; (c) R_{xy} is a tree-preserving but neither path- nor chain-preserving constraint as the image of the path $\{a, b\}$ of T_x, which is also a chain of T_x, is $\{e, f, g, h\}$ that is neither a path nor a chain of T_y.

- chain-preserving *if the image of every chain in T_x is a chain in T_y.*

We note that a subtree (a path or a chain) of T_x (or T_y) in the above definition is possibly empty. We also note that chain-preserving constraints are exactly those "locally chain convex and strictly union closed" constraints defined in [134].

CRC constraints are special chain-preserving constraints defined over chain domains. The following definition of CRC constraints is equivalent to the one given in Definition 1.11.

Definition 2.3. *Let x, y be two variables with finite tree domains T_x and T_y, where T_x and T_y are chains. A constraint R from x to y is* connected row-convex *(CRC), w.r.t. T_x and T_y, if both R and R^{-1} are chain-preserving.*

Definition 2.4. *A binary constraint network $\mathcal{N}$ over variables in V and tree domains T_x ($x \in V$) is called* tree-convex, chain-, path-, *or* tree-preserving *if every constraint $R \in \mathcal{N}$ is tree-convex, chain-, path-, or tree-preserving, respectively. A CRC constraint network is defined similarly.*

The following proposition summarizes relations between these tree-convex con-

straints.

Proposition 2.1. *Every chain-, path-, or tree-preserving constraint (network) is consecutive and every path-preserving constraint (network) is tree-preserving. Moreover, every arc-consistent consecutive tree-convex (ACCTC) constraint (network) is tree-preserving.*

Proof. First, we notice that an edge is a chain, a path and a subtree, and a chain or a path is a subtree. Then the claim that every chain-, path-, or tree-preserving constraint is consecutive directly follows from Definition 2.2.

Second, we show that if a constraint R_{xy} is path-preserving, then it is also tree-preserving. Let R_{xy} be a path-preserving constraint over tree domains T_x and T_y. Suppose that R_{xy} is not tree-preserving. We know that there exists a subtree t_x of T_x such that $R_{xy}(t_x)$ is not a subtree of T_y. Therefore, t_x can be divided into two parts, say t_x^1 and t_x^2, such that $R_{xy}(t_x^1)$ is disconnected from $R_{xy}(t_x^2)$. Let $v_1 \in t_x^1$, $v_2 \in t_x^2$ such that $R_{xy}(v_1) \neq \varnothing$ and $R_{xy}(v_2) \neq \varnothing$. Let π_{v_1,v_2} be the path from v_1 to v_2 in T_x. Let $\pi_1 = \pi_{v_1,v_2} \cap t_x^1$ and $\pi_2 = \pi_{v_1,v_2} \cap t_x^2$. Then $R_{xy}(\pi_1)$ is disconnected from $R_{xy}(\pi_2)$. Therefore, $R_{xy}(\pi_{v_1,v_2})$ cannot be a path of T_y which contradicts that R_{xy} is path-preserving.

Finally, if a constraint R_{xy} over tree domains T_x and T_y is arc-consistent consecutive tree-convex, we show that R_{xy} is also tree-preserving. Let t_x be an arbitrary subtree of T_x. We show that $R_{xy}(t_x)$ is a subtree of T_y. Let t_0 be a subtree of t_x such that $|t_0| = 1$. Because R_{xy} is tree-convex, $R_{xy}(t_0)$ is a subtree of T_y. Let t be a subtree of t_x and $e_{v_1v_2}$ be an edge of T_x with $v_1 \in t$, $v_2 \notin t$ and $v_2 \in t_x$. Suppose $R_{xy}(t)$ is a subtree of T_y. Because R_{xy} is arc-consistent consecutive tree-convex, $R_{xy}(e_{v_1v_2})$ is also a subtree of T_y, and $R_{xy}(v_1) \neq \varnothing$. Because $\varnothing \neq R_{xy}(v_1) \subseteq R_{xy}(t) \cap R_{xy}(e_{v_1v_2})$, $R_{xy}(t)$ and $R_{xy}(e_{v_1v_2})$ are connected. Thus, $R_{xy}(t \cup e_{v_1v_2})$ is also subtree of T_y. Therefore, by induction, we can add edges to t_0 one by one until $t_0 = t_x$, and in each step,

$R_{xy}(t_0)$ is a subtree of T_y. $\square$

Although every arc-consistent chain- or path-preserving constraint is tree-preserving, Figure 2.2(c) shows that the other direction is not always true. Furthermore, Figure 2.1(b) shows that not every chain-preserving (or consecutive tree-convex) constraint is tree-preserving and Figure 2.2(a,b) show that chain-preserving constraints and path-preserving constraints are incomparable.

The following results of trees will be used in the proof of some results in this chapter.

Lemma 2.1. *[137] Let T be a tree and suppose t_i $(i = 1, \ldots, m)$ are subtrees of T. Then $\bigcap_{i=1}^{m} t_i$ is nonempty iff $t_i \cap t_j$ is nonempty for every $1 \leq i \neq j \leq m$.*

Lemma 2.2. *Let T be a tree and t, t' subtrees of T. Suppose $\{u, v\}$ is an edge in T. If $u \in t$ and $v \in t'$, then $t \cup t'$ is a subtree of T; if, in addition, $u \notin t'$ and $v \notin t$, then $t \cap t' = \varnothing$.*

Proof. The first part is clear as the edge $\{u, v\}$ connects t and t'. Suppose $u \notin t'$, $v \notin t$. We show $t \cap t' = \varnothing$. Suppose this is not the case and there exists $w \in t \cap t'$. Then we have $\pi_{w,u} \subseteq t$ and $\pi_{w,v} \subseteq t'$. Since u is a neighbour of v, we have either $u \in \pi_{w,v}$ or $v \in \pi_{w,u}$, i.e., either $u \in t'$ or $v \in t$. Both contradict our assumption that $u \notin t'$, $v \notin t$. Therefore, we must have $t \cap t' = \varnothing$. $\square$

Using Lemma 2.1, Zhang and Yap [137] proved the following result:

Theorem 2.1. *A tree-convex constraint network is globally consistent if it is path-consistent.*

In the following, we will focus on the class of tree-preserving constraints.

2.4 Tree-Preserving Constraints

In this section, we show that the class of tree-preserving constraints is tractable. Given a tree-preserving constraint network $\mathcal{N}$, we show that, when enforcing strong path-consistency on $\mathcal{N}$, in each step, the network remains tree-preserving. Hence, by Theorem 2.1, we know that, if no inconsistency is detected, a tree-preserving constraint network will be transformed into an equivalent globally consistent network after enforcing strong path-consistency.

Firstly, we show that tree-preserving constraint networks are closed under arc-consistency.

Lemma 2.3. *Suppose R_{xy} and R_{yx} are tree-preserving (tree-convex) w.r.t. tree domains T_x and T_y. Let t be a subtree of T_x and $R'_{xy} = \{\langle a, b\rangle \in R_{xy} : a \in t\}$ and $R'_{yx} = \{\langle b, a\rangle \in R_{yx} : a \in t\}$ the restrictions of R_{xy} and R_{yx} to t. Then both R'_{xy} and R'_{yx} are tree-preserving (tree-convex).*

Proof. Note that a path or subtree of t is also a path or subtree of T_x. The conclusion then follows directly from the definitions of tree-preserving and tree-convex constraints. □

As a corollary, we have

Corollary 2.1. *Let $\mathcal{N}$ be a tree-preserving (tree-convex) constraint network over tree domains T_x ($x \in V$). Assume that t is a nonempty subtree of T_x. When restricted to t, $\mathcal{N}$ remains tree-preserving (tree-convex).*

The following lemma examines unsupported values of a tree-preserving constraint.

Lemma 2.4. *Suppose R_{xy} is nonempty and tree-preserving w.r.t. tree domains T_x and T_y. If $v \in T_y$ has no support in T_x under R_{yx}, then all supported nodes of T_y*

are in the same branch of v. That is, every node in any other branch of v is not supported under R_{yx}.

Proof. Suppose a, b are two supported nodes in T_y. There exist u_1, u_2 in T_x s.t. $u_1 \in R_{yx}(a)$ and $u_2 \in R_{yx}(b)$. By $R_{yx} = R_{xy}^{-1}$, we have $a \in R_{xy}(u_1)$ and $b \in R_{xy}(u_2)$. Hence $a, b \in R_{xy}(\pi_{u_1,u_2})$. Since R_{xy} is tree-preserving, $R_{xy}(\pi_{u_1,u_2})$ is a subtree in T_y. If a, b are in two different branches of v, then $\pi_{a,b}$ must pass v and hence we must have $v \in R_{xy}(\pi_{u_1,u_2})$. This is impossible as v has no support. □

It is worth noting that this lemma does not require R_{yx} to be tree-preserving.

The following result then follows directly.

Proposition 2.2. *Let $\mathcal{N}$ be a tree-preserving constraint network over tree domains T_x $(x \in V)$. If no inconsistency is detected, then $\mathcal{N}$ remains tree-preserving after enforcing arc-consistency.*

Proof. Enforcing arc-consistency on $\mathcal{N}$ only removes values which have no support under some constraints. For any $y \in V$, if v is an unsupported value in T_y, then, by Lemma 2.4, every supported value of T_y is located in the same branch of v. Deleting all these unsupported values from T_y, we get a subtree t of T_y. Applying Corollary 2.1, the restricted constraint network to t remains tree-preserving. □

Secondly, we consider the intersection and composition of tree-preserving constraints.

When doing relational intersection, we may need to remove some unsupported values from domains. Unlike CRC [40, Lemma 13] and chain-preserving constraints [134, Proposition 5], removing a value from a domain may change the tree-preserving property of a network. Instead, we need to remove a 'trunk' from the tree domain or just keep one branch.

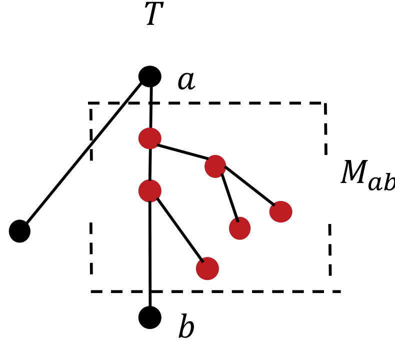


Figure 2.3 : M_{ab} is a trunk of tree T, i.e., the subtree induced by vertices with red colour.

Definition 2.5. *Suppose $a \neq b$ are two nodes of a tree T that are not neighbours. The* trunk *between a, b, written as $M_{a,b}$, is defined as the connected component of $T \setminus \{a, b\}$ which contains all the internal nodes of $\pi_{a,b}$ (see Figure 2.3). The* M-contraction *of T by $M_{a,b}$, denoted by $T \ominus M_{a,b}$, is the tree obtained by removing the nodes with associated edges in $M_{a,b}$ and adding an edge $\{a, b\}$ to T.*

To improve readability, we defer the proofs of Lemmas 2.5-2.7 to Appendix.

Lemma 2.5. *Let $\mathcal{N}$ be an arc-consistent and tree-preserving constraint network over tree domains T_x $(x \in V)$. Suppose $x \in V$ and $M_{a,b}$ is a trunk in T_x. When restricted to $T_x \ominus M_{a,b}$ and enforcing arc-consistency, $\mathcal{N}$ remains tree-preserving if no inconsistency is detected.*

The following two lemmas consider the intersection of two tree-preserving constraints.

Lemma 2.6. *Assume R_{xy} and R'_{xy} are two arc-consistent and tree-preserving constraints w.r.t. trees T_x and T_y. Let $R^*_{xy} = R_{xy} \cap R'_{xy}$. Let $W = \{w \in T_x \mid R^*_{xy}(w) \neq \varnothing\}$ be the set of supported values of R^*_{xy}. Suppose $u \in T_x$ and $u \notin W$. Then there exist at most two values w_1, w_2 in W s.t. no value in W other than w_i is on the path $\pi_{w_i,u}$ for $i = 1, 2$.*

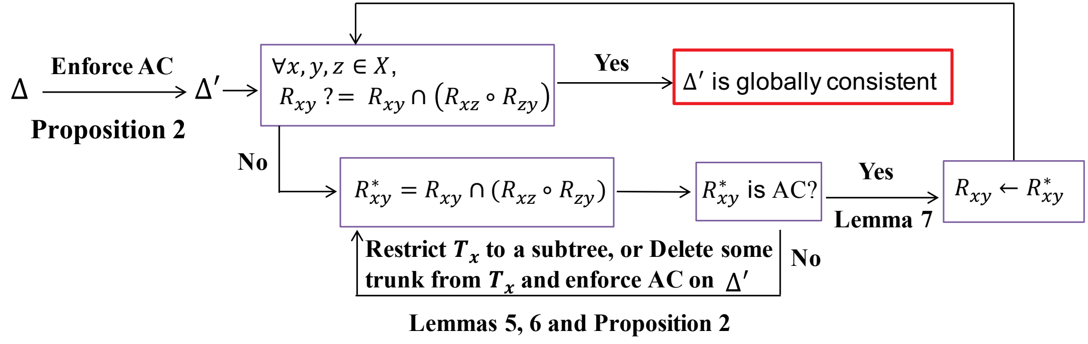


Figure 2.4 : The flow diagram of proof of Theorem 2.2.

Remark 2.1. *If there is only one value w_1 in W that satisfies the condition in Lemma 2.6, regarding u as the root of T_x, then W is contained in the subtree rooted at w_1; if there are two values w_1, w_2 in W that satisfy the condition in the Lemma 2.6, then the trunk M_{w_1,w_2} should be contracted from T_x to make W connected.*

Lemma 2.7. *Suppose R_{xy} and R'_{xy} are arc-consistent and tree-preserving constraints w.r.t. trees T_x and T_y and so are R_{yx} and R'_{yx}. Let $R^*_{xy} = R_{xy} \cap R'_{xy}$. Assume $\{u, v\}$ is an edge in T_x s.t. $R^*_{xy}(u) \neq \varnothing$, $R^*_{xy}(v) \neq \varnothing$, and $R^*_{xy}(u) \cup R^*_{xy}(v)$ is disconnected in T_y. Then there exist unique $r \in R^*_{xy}(u)$ and $s \in R^*_{xy}(v)$ s.t. every node in $M_{r,s}$ is unsupported under R^*_{yx}.*

The following result follows from the definition of tree-preserving constraints.

Proposition 2.3. *Assume that R_{xz} and R_{zy} are two tree-preserving constraints w.r.t. trees T_x, T_y, and T_z. Then their composition $R_{xz} \circ R_{zy}$ is tree-preserving.*

Proof. Let $R'_{xy} = R_{xz} \circ R_{zy}$ and t_x be an arbitrary subtree of T_x. Then we have that $R'_{xy}(t_x) = R_{zy}(R_{xz}(t_x))$. Because R_{xz} is tree-preserving, we have that $R_{xz}(t_x)$ is a subtree of T_z. Similarly, $R_{zy}(R_{xz}(t_x))$ is a subtree of T_y. Thus, R'_{xy} is tree-preserving. □

Finally, we give the main result of this section.

Theorem 2.2. *Let $\mathcal{N}$ be a tree-preserving constraint network. If no inconsistency is detected, then enforcing strong path-consistency determines the consistency of $\mathcal{N}$ and transforms $\mathcal{N}$ into a globally consistent network.*

Proof. Figure 2.4 is the flow diagram of the proof.

If we can show that $\mathcal{N}$ is still tree-preserving after enforcing strong path-consistency, then by Theorem 2.1 the new network is globally consistent if no inconsistency is detected.

By Proposition 2.2, $\mathcal{N}$ remains tree-preserving after enforcing arc-consistency. To enforce path-consistency on $\mathcal{N}$, we need to call the following updating rule:

$$R_{xy} \leftarrow R_{xy} \cap (R_{xz} \circ R_{zy}) \tag{2.1}$$

$$R_{yx} \leftarrow R_{xy}^{-1} \tag{2.2}$$

for $x, y, z \in V$ until the network is stable.

Suppose $\mathcal{N}$ is arc-consistent and tree-preserving w.r.t. trees T_x for $x \in V$ before applying (2.1). Note that if $R^*_{xy} = R_{xy} \cap (R_{xz} \circ R_{zy})$ (as well as its inverse R^*_{yx}) is arc-consistent, then $R^*_{xy}(u)$ is nonempty for any node u in T_x. By Lemma 2.7, $R^*_{xy}(u) \cup R^*_{xy}(v)$ is connected for every edge $\{u, v\}$ in T_x as otherwise there will exist unsupported nodes in T_y under the inverse of R^*_{xy}. Therefore R^*_{xy} is arc-consistent and consecutive, and hence, tree-preserving. Since $R^*_{yx} = R^{*-1}_{xy} = R_{yx} \cap (R_{yz} \circ R_{zx})$, analogously, we have R^*_{yx} is tree-preserving.

If R^*_{xy} is not arc-consistent, then there exists $u \in T_x$ s.t. $R^*_{xy}(u)$ is empty. By Lemma 2.6 and Remark 2.1, we should restrict the domain to a subtree or contract some trunk from T_x and enforce arc-consistency. If R^*_{yx} is not arc-consistent, then we do analogously. By Lemma 2.5 and Proposition 2.2, if no inconsistency is detected, then we have an updated arc-consistent and tree-preserving network. Still write $\mathcal{N}$

for this network and recompute R^*_{xy}, R^*_{yx} and repeat the above procedure until either an inconsistency is detected or both R^*_{xy} and R^*_{yx} are arc-consistent. Note that, after enforcing arc-consistency, the composition $R_{xz} \circ R_{zy}$ may have changed.

Once arc-consistency of R^*_{xy} and R^*_{yx} is achieved, we update R_{xy} with R^*_{xy} and R_{yx} with R^*_{yx} and continue the process of enforcing path-consistency until $\mathcal{N}$ is path-consistent or an inconsistency is detected. □

In above, we assume that each domain is associated to a tree structure. Actually, our definitions and results of tree-preserving constraints can be straightforwardly extended to domains with acyclic graph structures (which are connected or not). We call such a structure a *forest* domain.

Proposition 2.4. *The consistency of a tree-preserving constraint network over forest domains can be reduced to the consistency of several parallel tree-preserving networks over tree domains.*

Proof. Given a tree-preserving constraint network $\mathcal{N}$ over forest domains $F_1, ..., F_n$ of variables $v_1, ..., v_n$, suppose that F_i consists of trees (i.e., maximally connected components) $t_{i,1}, \ldots, t_{i,k_i}$. Note that the image of each tree, say $t_{i,1}$, of F_i under constraint R_{ij} is a subtree t of F_j. Assume t is contained in the tree $t_{j,s}$ of forest F_j. Then the image of $t_{j,s}$ under constraint R_{ji} is a subtree of $t_{i,1}$. This establishes, for any $1 \leq i \neq j \leq n$, a 1-1 correspondence between trees in F_i and trees in F_j if the image of each tree is nonempty. In this way, the consistency of $\mathcal{N}$ is reduced to the consistency of several parallel tree-preserving networks over tree domains. □

Figure 2.5 shows a tree-preserving network over forest domains. We note that $\mathcal{N}$ cannot be modelled as tree-preserving over tree domains. For example, if we modify F_1 as a tree T_1 by adding edges $\{a, b\}$ and $\{b, c\}$. Then, in order to make R_{13} tree-preserving, edges $\{g, h\}$ and $\{h, i\}$ should be added to F_3. Write T_3 for the new tree.

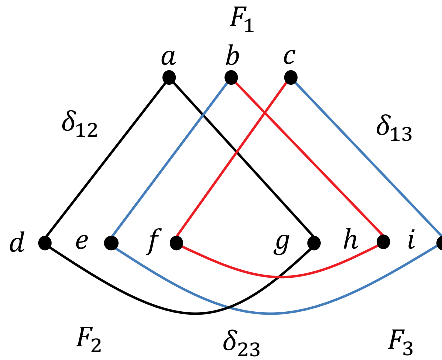


Figure 2.5 : A tree-preserving network $\mathcal{N} = \{R_{12}, R_{13}, R_{23}\}$ over forest domains F_1, F_2, and F_3, where each forest domain consists of three trees which contain only one node.

Likewise, in order to make R_{12} tree-preserving, edges $\{d, e\}$ and $\{e, f\}$ should be added to F_2. Write T_2 for the new tree. However, R_{23} is not tree-preserving w.r.t. T_2 and T_3.

Recall that when enforcing path-consistency, we transform a constraint network into a complete constraint graph despite the number of non-trivial constraints it has. In the following section, we consider a more efficient path-consistency algorithm that respects the density of non-trivial constraints in the network.

2.5 Partial PC for Tree-Preserving Constraints

Partial PC (PPC) is a more general consistency condition than PC and can be enforced more efficiently for constraint networks with sparse constraint graphs. The idea of PPC is to enforce path-consistency on sparse constraint graphs by triangulating instead of completing them. Bliek and Sam-Haroud demonstrated that, as far as CRC constraints are concerned, the pruning capacity of path-consistency on triangulated graphs and their completions are identical on the common edges. In this section, we show that a similar result applies to tree-preserving constraints. Moreover, we show that, after enforcing strong PPC (i.e., both AC and PPC), we can find a solution in a backtrack-free style if no inconsistency is detected.

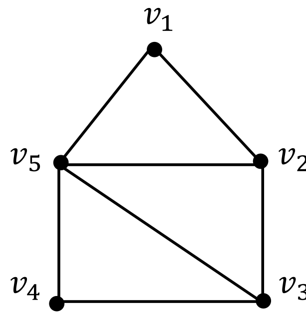


Figure 2.6 : A triangulated graph G, where $\langle v_1, v_2, v_3, v_4, v_5 \rangle$ is a perfect vertex elimination ordering of G.

Basic definitions and results related to graph triangulation and PPC have been given in subsections 1.3.2 and 1.3.3. Some extra graph notations and results are given below.

Definition 2.6 ([13])**.** *Suppose $\prec = \langle v_1, v_2, ..., v_n \rangle$ is a perfect vertex elimination ordering of graph G. For $1 \leq i \leq n$, we denote by G_i the subgraph of G induced by $S_i = \{v_{n-i+1}, ..., v_n\}$ and write $F_i = \{v_k \in N(v_{n-i}) | v_{n-i} \prec v_k\}$.*

Since the vertex elimination ordering is perfect, the subgraph induced by F_i is complete. An example is given in Figure 2.6, where G is a triangulated constraint graph and $\langle v_1, \ldots, v_5 \rangle$ is a perfect vertex elimination ordering of G. By Definition 2.6, we can see that S_i just denotes the last i vertices w.r.t. the ordering. Therefore, we have $S_1 = \{v_5\}, S_2 = \{v_4, v_5\}, S_3 = \{v_4, v_4, v_5\}, S_4 = \{v_2, v_3, v_4, v_5\}$ and $S_5 = \{v_1, v_2, v_3, v_4, v_5\}$. Finally, F_i denotes the set of vertices that are adjacent to v_{n-i} and after it w.r.t. the ordering. Therefore, we have $F_1 = \{v_5\}, F_2 = \{v_4, v_5\}, F_3 = \{v_3, v_5\}$ and $F_4 = \{v_2, v_5\}$.

Proposition 2.5 ([13])**.** *A triangulated constraint graph G is path-consistent iff every path of length 2 is path-consistent.*

The following result is a straightforward extension from CRC constraints [13] to tree-preserving constraints. For the purpose of being self-contained, we provide a

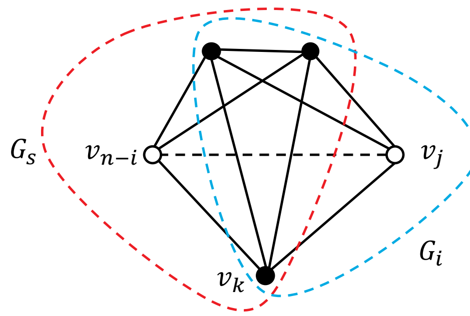


Figure 2.7 : Illustration of proof of Lemma 2.8. The figure is taken from [13].

complete proof below.

Lemma 2.8. *Suppose $\mathcal{N}$ is a strongly partial path-consistent tree-preserving constraint network with a triangulated constraint graph $G = (V, E)$. Assume that $\prec = \langle v_1, v_2, ..., v_n \rangle$ is a perfect vertex elimination ordering of G. Let $G_i, S_i, F_i (1 \leq i \leq n)$ be defined as in Definition 2.6. Suppose i is the largest index such that G_i is complete. Assume that v_j is a node in S_i that is not a neighbour of v_{n-i} in G. Let $R_{v_{n-i},v_j} = \bigcap_{x \in F_i} \left(R_{v_{n-i},x} \circ R_{x,v_j} \right)$. Then*

(i) The constraint graph $G' = (V, E \cup \{(v_{n-i}, v_j)\})$ is triangulated and $\prec$ is also a perfect vertex elimination ordering of G' and F_i is exactly the set $\{x \mid (v_{n-i}, x), (x, v_j) \in E\}$;

(ii) G' is strongly path-consistent;

(iii) The constraint R_{v_{n-i},v_j} and its inverse are tree-preserving.

Proof. (i) This result directly follows from the proof of [13, Lemma 1].

(ii) First, we show that G' is PC. To this end, because G' is triangulated, it is sufficient to show that every path of length 2 of G' is PC by Proposition 2.5. Because G is PC, paths of length 2 of G' that do not go through v_{n-i} and v_j are PC. So, let us consider paths of length 2 of G' that go through v_{n-i} and v_j.

Let G_s be the subgraph of G that is induced by $F_i \cup \{v_{n-i}\}$. Note that G_i is the subgraph of G that is induced by $F_i \cup \{v_j\}$. See Figure 2.7 for an illustration where the vertices in F_i are colored black. We claim that G_s and G_i are globally consistent. By assumption, we know that G_i is complete. Because $\prec$ is a perfect vertex elimination ordering of G, we know that G_s is also complete. Now, because G_i and G_s are complete, strongly path-consistent and tree-preserving, they are globally consistent by Theorem 2.1.

Let π be a path of length 2 of G' that goes through v_{n-i} and v_j. We now show that π is PC. To this end, we have to consider the following two cases:

Case 1: $\pi = \langle v_{n-i}, v_k, v_j \rangle$ with some $v_k \in V(v_k \neq v_{n-i}, v_j)$. By (i), we know that F_i is exactly the set $\{x \mid (v_{n-i}, x), (x, v_j) \in E\}$. Therefore, $v_k \in F_i$. Because G_s is globally consistent, it admits at lease one solution, say α. Because G_i is also globally consistent, $\alpha|_{F_i}$ can be consistently extended to v_j such that all constraints in G_i are satisfied. Therefore, α can be extended to a solution of $G_s \cup G_i$, say β. Then we have that, for all $x \in F_i$, $\langle \beta|_{v_{n-i}}, \beta|_x \rangle \in R_{v_{n-i},x}$ and $\langle \beta|_x, \beta|_{v_j} \rangle \in R_{x,v_j}$. Therefore, $\langle \beta|_{v_{n-i}}, \beta|_{v_j} \rangle \in \bigcap_{x \in F_i} \left(R_{v_{n-i},x} \circ R_{x,v_j} \right)$. Thus, $R_{v_{n-i},v_j} = \bigcap_{x \in F_i} \left(R_{v_{n-i},x} \circ R_{x,v_j} \right) \neq \varnothing$. Further, because $v_k \in F_i$, for any $\langle a, b \rangle \in R_{v_{n-i},v_j}$, there is some $c \in D_k$ such that $\langle a, c \rangle \in R_{v_{n-i},v_k}$ and $\langle c, b \rangle \in R_{v_k,v_j}$, and thus $\pi = \langle v_{n-i}, v_k, v_j \rangle$ is PC.

Case 2: $\pi = \langle v_{n-i}, v_j, v_k \rangle$ with some $v_k \in V(v_k \neq v_{n-i}, v_j)$. Since G' is triangulated by (i), there is an edge $\{v_{n-i}, v_k\}$ of G'. Therefore, $v_k \in F_i$. Surely, R_{v_{n-i},v_k} is not empty, because it is a constraint in G and G is PC. Now, we show that for any $\langle a, b \rangle \in R_{v_{n-i},v_k}$, there is some $c \in D_j$ such that $\langle a, c \rangle \in R_{v_{n-i},v_j}$ and $\langle c, b \rangle \in R_{v_j v_k}$. Because G_s is globally consistent, $\langle a, b \rangle$ can be extended to a solution of G_s. Similar to case 1, we know that the solution of G_s can be extended to a solution of $G_s \cup G_i$. We write such a solution of $G_s \cup G_i$ as ψ. Let value c be the restriction of ψ to variable v_j. Certainly, $\langle c, b \rangle \in R_{v_j v_k}$. Further, for any $x \in F_i$, let c' be the

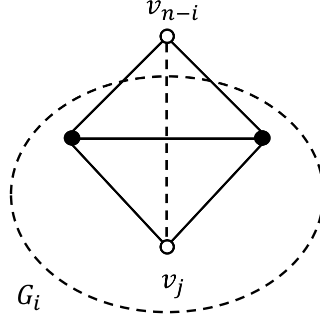


Figure 2.8 : Illustration of proof of Theorem 2.3.

restriction of ψ to x. We have that $\langle a, c' \rangle \in R_{v_{n-i},x}$ and $\langle c', c \rangle \in R_{x,v_j}$, and thus $\langle a, c \rangle \in R_{v_{n-i},v_j} = \bigcap_{x \in F_i} \left(R_{v_{n-i},x} \circ R_{x,v_j} \right)$. Therefore, we have that $\pi = \langle v_{n-i}, v_j, v_k \rangle$ is also PC.

Second, we show that G' is AC. To this end, it is sufficient to prove that R_{v_{n-i},v_j} is AC. Because $R_{v_{n-i},v_k}(v_k \in F_i)$ is AC, for any $a \in D_{n-i}$, there is a $v \in D_k$ such that $\langle a, b \rangle \in R_{v_{n-i},v_k}$. Further, because $\pi = \langle v_{n-i}, v_j, v_k \rangle$ is PC, there is a $c \in D_j$ such that $\langle a, c \rangle \in R_{v_{n-i},v_j}$ and $\langle c, b \rangle \in R_{v_j,v_k}$. Similarly, for any $c \in D_j$, there is a $a \in D_{n-i}$ such that $\langle c, a \rangle \in R_{v_j,v_{n-i}}$. So, R_{v_{n-i},v_j} is AC.

(iii) By (i), we have $F_i = \{x \mid \{v_{n-i}, x\}, \{x, v_j\} \in E\}$. According to Proposition 2.3, we know that $R_{v_{n-i},x} \circ R_{x,v_j}$ is tree-preserving for any $x \in F_i$. Furthermore, because R_{v_{n-i},v_j} is arc-consistent by (ii), according to Lemma 2.7, we know that $R_{v_{n-i},v_j} = \bigcap_{x \in F_i} \left(R_{v_{n-i},x} \circ R_{x,v_j} \right)$ is tree-preserving. Similarly, since

$$R^{-1}_{v_{n-i},v_j} = \bigcap_{x \in F_i} \left(R_{v_{n-i},x} \circ R_{x,v_j} \right)^{-1} = \bigcap_{x \in F_i} \left(R^{-1}_{x,v_j} \circ R^{-1}_{v_{n-i},x} \right) = \bigcap_{x \in F_i} \left(R_{v_j,x} \circ R_{x,v_{n-i}} \right),$$

we know that the inverse of R_{v_{n-i},v_j} is also tree-preserving. □

By the above lemma, we now prove that the result obtained for CRC constraints in [13] also applies to tree-preserving constraints.

Theorem 2.3. *For a tree-preserving constraint network $\mathcal{N}$ with a triangulated con-*

straint graph G, strong PC on G is equivalent to strong PC on the completion of G in the sense that the relations computed for the constraints in G are identical.

Proof. The proof is analogous to that for CRC constraints in [13]. Suppose we have a triangulated constraint graph $G = (V, E)$ that is strongly PC. We will add to G the missing edges one by one until the graph is complete. To prove the theorem, we show that the relations of the constraints can be computed from the existing ones so that each intermediate graph, including the completed graph, is strongly PC. Let i be the largest index such that G_i is complete. Since G_{i+1} is not complete, we add one by one all missing edges $\{v_{n-i}, v_j\}$ into G_{i+1} and compute their corresponding constraints as

$$R_{v_{n-i},v_j} = \bigcap_{x \in F_i} \left(R_{v_{n-i},x} \circ R_{x,v_j}\right)$$

(see Figure 2.8 for an illustration, where vertices in F_i are denoted in black). By Lemma 2.8, we know R_{v_{n-i},v_j} and its inverse are tree-preserving and the revised graph G'_{i+1} remains triangulated. Continuing in this way, we will transform G_{i+1} into a complete graph. Applying the above procedure to G_{i+1}, and so on, until the whole graph is complete, we will have the desired result. □

Then, we show that enforcing PPC on a consistent tree-preserving constraint network transforms it into an equivalent constraint network that is backtrack-free in the following sense.

Definition 2.7 ([37]). *A constraint network is* backtrack-free *relative to a given ordering $\prec = \langle x_1, \cdots, x_n \rangle$ if for every $i \leq n-1$, every partial solution of $\{x_1, \ldots, x_i\}$ can be consistently extended to x_{i+1}.*

We now have the main result of this section.

Theorem 2.4. *Suppose $\mathcal{N}$ is a tree-preserving constraint network with triangulated constraint graph G. If no inconsistency is detected, then enforcing strong PPC on*

G transforms it into an equivalent consistent network that is backtrack-free relative to the reverse ordering of any perfect elimination ordering $\prec = \langle v_1, \cdots, v_n\rangle$ of G.

Proof. Suppose $\prec = \langle v_1, \cdots, v_n\rangle$ is a perfect elimination ordering of G. In the following, we use notations S_i, F_i and G_i that are defined in Definition 2.6.

Assume that no inconsistency is detected. Write $\mathcal{N}^*$ for the equivalent network obtained from enforcing strong PPC on $\mathcal{N}$. We show that $\mathcal{N}^*$ is backtrack-free w.r.t. the ordering $\prec^{-1} = \langle v_n, v_{n-1}, ..., v_1\rangle$. To this end, we need to show that, for any $2 \leq i \leq n$, any consistent instantiation of vertices in $S_{i-1} = \{v_n, v_{n-1}, \cdots, v_{n-i+2}\}$ can be consistently extended to v_{n-i+1}.

Suppose the above statement holds for any $2 \leq i \leq j$. We show that it holds for $i = j + 1$. Because the elimination ordering is perfect, $F_j \cup \{v_{n-j}\}$ is complete. So, the restriction of $\mathcal{N}$ into $F_j \cup \{v_{n-j}\}$ is strongly PC and tree-preserving and thus, by Theorem 2.2, globally consistent. Therefore, any consistent instantiation to vertices in F_j could be consistently extended to v_{n-j}. Also, because there are no edges (i.e., no constraints) between v_{n-j} and vertices in $G_j \setminus F_j$, any consistent instantiation to vertices in G_j could be consistently extended to v_{n-j} such that all constraints in G_{j+1} are satisfied. In this way, we have shown that $\mathcal{N}^*$ is backtrack-free w.r.t. $\prec^{-1}$. □

According to Theorem 2.4, enforcing PPC is sufficient to solve tree-preserving constraint networks. In the following, we will show that *conservative dual-consistency* (CDC) [81] is equal to PPC for tree-preserving constraint networks with triangulated constraint graphs.

Given a binary constraint network $\mathcal{N}$, $\mathcal{N}|_{v_i=a_i}$ represents the network obtained from $\mathcal{N}$ by restricting the domain of v_i to the singleton $\{a_i\}$ and $AC(\mathcal{N}|_{v_i=a_i})$ denotes the network obtained by enforcing AC on $\mathcal{N}|_{v_i=a_i}$.

Definition 2.8. *[81] Let $\mathcal{N}$ be a binary constraint network over variable set V. $\mathcal{N}$ is called* conservative dual-consistent *(CDC) if for any $R_{v_iv_j} \in \mathcal{N}$ and any $\langle a_i, a_j \rangle \in R_{v_iv_j}$, we have $a_j \in D_j^*$ where D_j^* is the domain of v_j w.r.t. $AC(\mathcal{N}|_{v_i=a_i})$. $\mathcal{N}$ is called strongly CDC if it is both AC and CDC.*

Proposition 2.6. *Strong partial path-consistency is equivalent to strong conservative dual-consistency for tree-preserving constraint networks with triangulated constraint graphs.*

Proof. Let $\mathcal{N}$ be a tree-preserving constraint network over variable set V. Suppose that the constraint graph $G = (V, E)$ of $\mathcal{N}$ is triangulated. If $\mathcal{N}$ is strongly CDC, then $\mathcal{N}$ is also strongly PPC, because conservative dual-consistency is a stronger consistency condition than partial path-consistency [83]. Now, suppose that $\mathcal{N}$ is strongly PPC, we show that $\mathcal{N}$ is also strongly CDC. We first obtain a new network $\mathcal{N}'$ by adding all the missing edges to G to make it complete. Constraints of newly added edges are all set to be universal. Then we enforce strong PC on $\mathcal{N}'$. Now, $\mathcal{N}'$ is strongly PC and $\mathcal{N}'$ is equivalent to $\mathcal{N}$. By Theorem 2.2, $\mathcal{N}'$ is globally consistent. Let $R_{v_iv_j}$ be an arbitrary constraint of $\mathcal{N}$. By Theorem 2.3, $R_{v_iv_j}$ is also a constraint of $\mathcal{N}'$. Then, for any tuple $\langle a_i, a_j \rangle \in R_{v_iv_j}$, it can be extended to a solution ψ of $\mathcal{N}'$ which is also a solution of $\mathcal{N}$. Therefore, we have $a_j \in D_j^*$ where D_j^* is the domain of v_j w.r.t. $AC(\mathcal{N}|_{v_i=a_i})$, and we know that $\mathcal{N}$ is CDC. Because $\mathcal{N}$ is also AC, it is strongly CDC. □

Consequently, we can adopt efficient strong CDC enforcing algorithms, such as sCDC1 [83], to enforce PPC for tree-preserving constraint networks.

Finally, we give Algorithm 2.1 below to find solutions for tree-preserving constraint networks.

We first explain how Algorithm 2.1 works, and then prove its correctness and

Algorithm 2.1: Solving tree-preserving constraint network using PPC.

input : A tree-preserving constraint network $\mathcal{N}$.
output: A solution or inconsistency.

```
1  Triangulate the constraint graph G(N);
2  Find a perfect elimination order of G(N), {v_1, v_2, ..., v_n};
3  Enforce PPC on G(N);
4  if no inconsistency is detected then
5  |   Choose values a_n and a_{n-1} for v_n and v_{n-1} respectively s.t. (a_n, a_{n-1})
   |   satisfies R_{n,n-1};
6  |   for i <- n - 2 to 1 do
7  |   |   S = ∩_{v_k ∈ F_{n-i}} R_{ki}(v_k);
8  |   |   Pick a value a_i from S for v_i;
9  |   end
10 end
11 else return inconsistency;
```

analyse its time complexity in Theorem 2.5. Lines 1-3 are self-explanatory. Line 5 instantiates $G_2 = \{v_n, v_{n-1}\}$, and then Lines 6-9 consistently instantiate G_3 to G_n in a sequential way. Line 8 extends the consistent instantiation of G_i to G_{i+1} by finding a consistent value for v_{n-i}.

Take the constraint graph in Figure 2.6 as an example. The algorithm first assigns consistent values a_5 and a_4 to the variables v_5 and v_4 of $G_2 = \{v_5, v_4\}$ respectively, and then extends the instantiation of G_2 to G_3 by finding a consistent value for v_3, i.e., to find a value a_3 for vertex v_3 such that $(a_i, a_3) \in R_{v_i,v_3}$ for all $i \in F_2$. To achieve this, Algorithm 2.1 computes the intersection of $R_{v_i,v_3}(a_i)$ for all $i \in F_2$. By Theorem 2.3, the intersection S is nonempty. Then Algorithm 2.1 picks any value from S for v_3. Similarly, the algorithm extends the instantiation of G_3 to G_4, and then extends the instantiation of G_4 to G_5.

Theorem 2.5. *Algorithm 2.1 is correct and its time complexity is* $O(n(e+f)+\alpha(e+f)d^3)$*, where* α *is the maximum degree of vertices in* $G(\mathcal{N})$*, the constraint graph of the input tree-preserving constraint network,* f *is the number of added edges to make* $G(\mathcal{N})$ *triangulated and* d *is the maximal domain size.*

Proof. The correctness of Algorithm 2.1 follows directly from Theorem 2.3. Now, we analyze time complexity of the algorithm. Finding a minimum triangulation for $G(\mathcal{N})$ in Line 1 could be done in $O(n(e+f))$[67], where f is the number of added edges. In Line 2, a perfect elimination ordering for the minimum triangulation of G can be found in $O(n+e+f)$ [67]. Also, enforcing PPC in Line 3 can be done in $O(\alpha(e+f)d^3)$[13], and Lines 6-9 take time $O(\alpha nd)$. Therefore, the overall time complexity of the algorithm is $O(n(e+f)+\alpha(e+f)d^3)$. □

In the next section, we consider a particular application of tree-preserving constraints.

2.6 The Scene Labelling Problem

The scene labelling problem [58] is a classification problem where every edge in a line-drawing picture has to be associated with a label describing it. The scene labelling problem is NP-complete in general and this is true even in the case of the trihedral scenes, i.e., scenes where no four planes share a point [66]. Several tractable subclasses of scene labelling problem have been identified (cf. [29, 65]).

Labels used in the scene labelling problem are listed as follows:

‘+’ The edge is *convex*, *i.e.,*, the edge can be touched by a ball;

‘−’ The edge is *concave*, *i.e.,*, the edge cannot be touched by a ball;

‘→’ Only one plane associated with the edge is visible, and when one moves in the direction indicated by the arrow, the pair of associated planes is *to the right*.

In the case of trihedral scenes, there are only four basic ways in which three plane surfaces can come together at a vertex [25, 58]. A vertex projects into a ‘V’, ‘W’, ‘Y’ or ‘T’-junction in the picture (each of these junction-types may appear with an arbitrary rotation in a given picture). A complete list of the labelled line

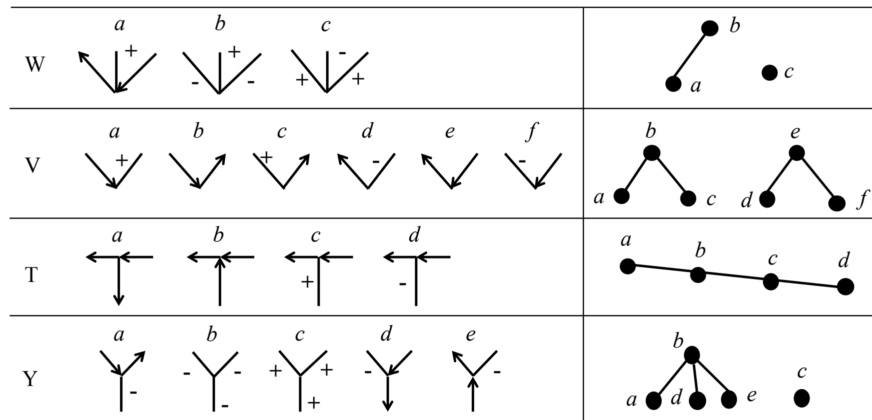


Figure 2.9 : Possible labelled line configurations of a junction in a picture and their corresponding forest structures.

configurations that are possible in the vicinity of a node in a picture is given in Figure 2.9.

In this section, we show that (i) every instance of the trihedral scene labelling problem can be modelled by a tree-convex constraint network over forest domains; (ii) a large subclass of the trihedral scene labelling problem can be modelled by tree-preserving constraints.

A CSP for the scene labelling problem can be formulated as follows. Each junction in the line-drawing picture is a variable. The domains of the variables are the possible configurations as shown in Figure 2.9. The constraints between variables are simply that, if two variables share an edge, then the edge must be labelled the same at both ends.

Proposition 2.7. *Every instance of the trihedral scene labelling problem can be modelled by a tree-convex constraint network. Furthermore, there are only 39 possible configurations of two neighbouring nodes in 2D projected pictures of 3D trihedral scenes, and 29 out of these can be modelled by tree-preserving constraints.*

Proof. The complete list of these configurations and their corresponding tree-convex

or tree-preserving constraints can be found in the online appendix[1]. Because '—' must be labelled by an arrow from right to left and '|' can be labelled by any labels, the T-junctions decompose into unary constraints. For this reason, we do not consider T-junctions in line drawing pictures. □

As a consequence, the consistency of any constraint network whose relations are taken from these 29 relations [2] can be decided by enforcing strong path-consistency. Moreover, because it is NP-hard to decide if a trihedral scene labelling instance is consistent, we have the following corollary.

Corollary 2.2. *The consistency problem of tree-convex constraint networks is NP-complete.*

A scene labelling instance and its corresponding constraint network are shown in Figure 2.10; the network is tree-preserving but neither chain-preserving nor CRC. Consider the line drawing on the left of Figure 2.10 and the constraints for the drawing listed on the right. One can easily verify that all constraints are tree-preserving w.r.t. the forest structures listed in Figure 2.9, but, for example, R_{21} is not chain-preserving w.r.t. the forest structures illustrated in Figure 2.9 and R_{25} is not CRC.

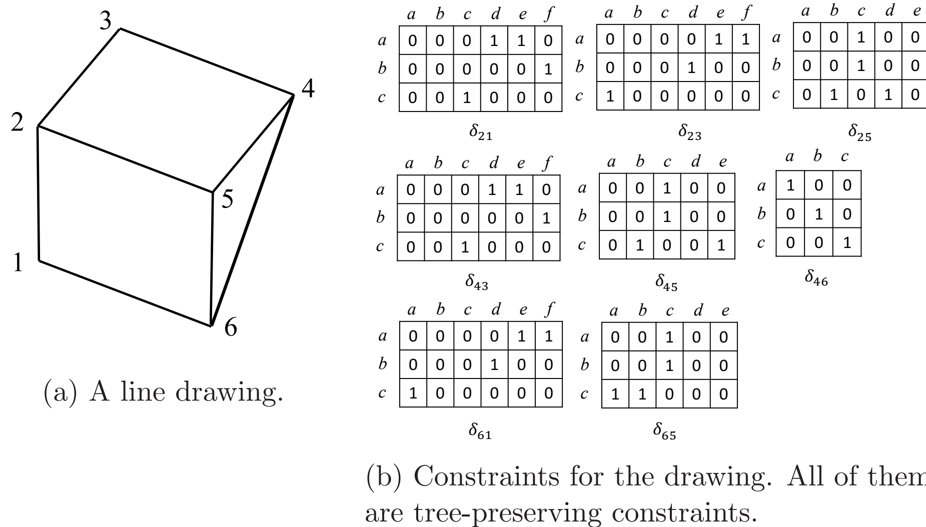
In the following section we give another method for proving the tractability of the class of tree-preserving constraints.

2.7 Algebraic Closure Properties of Tree-Preserving Constraints

We have shown that strong path-consistency ensures global consistency for the classes of chain-, path-, and tree-preserving constraints and thus identified three

[1] https://www.researchgate.net/publication/301815699_Appendix

[2] Among those 29 relations, 14 relations are not chain- or path-preserving.



(a) A line drawing.

(b) Constraints for the drawing. All of them are tree-preserving constraints.

Figure 2.10 : A scene labelling instance and its corresponding constraint network.

tractable classes of binary relations. Our approach follows the research line initiated by Dechter [36] and continued in e.g. [7, 40, 133, 137], which are based on the idea of achieving global consistency by enforcing local consistency.

An alternative approach to the study of tractable classes of relations focuses on certain algebraic closure properties of constraints [17, 45, 62]. In this section, we show that, under mild restrictions, a relation is tree-preserving *if and only if* it satisfies some algebraic closure property, which has been introduced in subsection 1.3.5.

For each tree domain, we introduce a natural majority operation.

Definition 2.9. *Let T_x be a nonempty tree domain for a variable x. We define a majority operation m_x as:*

$$(\forall a, b, c \in T_x) \qquad m_x(a, b, c) = \pi_{a,b} \cap \pi_{b,c} \cap \pi_{a,c}, \tag{2.3}$$

where a, b, c are not necessarily distinct and $\pi_{u,v}$ denotes the unique path from u to v in T_x. We call m_x the standard *majority operation on T_x.*

Even for one-sorted relations over a tree domain T, the class of tree-preserving

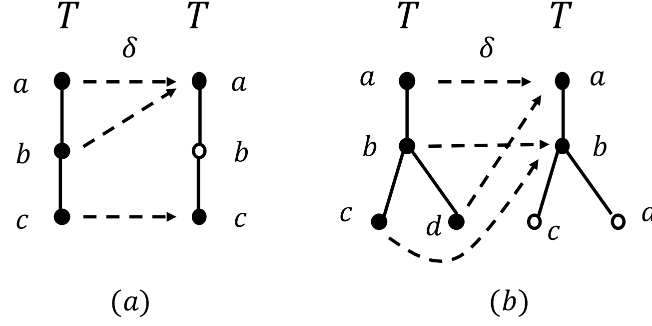


Figure 2.11 : (a) R is a relation from tree domain T to T, and $R = \{(a, a), (b, a), (c, c)\}$. R is closed under the standard majority operation on T, but it is not tree-preserving w.r.t. T. (b) R is a relation from tree domain T to T, and $R = \{(a, a), (b, b), (c, b), (d, a)\}$. R is tree-preserving w.r.t. T, but it is not closed under the standard majority operation on T.

relations on T is not comparable to the class of relations that are closed under the standard majority operation on T (see Figure 2.11 for an illustration).

The following lemma gives two important properties of relations closed under standard majority operations.

Lemma 2.9. *Let T_x and T_y be two nonempty tree domains and m_x and m_y their standard majority operations. Suppose $R \subseteq T_x \times T_y$ is a nonempty relation that is closed under $\{m_x, m_y\}$. Then*

- *R^{-1}, the inverse of R, is closed under $\{m_y, m_x\}$;*
- *if $a', b' \in T_y$ are the only supported nodes in the path from a' to b', then $R^{-1}(M_{a',b'}) = \varnothing$, where $M_{a',b'}$ is the trunk between a', b' in T_y.*

Proof. The first result follows directly from the definition.

To prove the second result, suppose $\langle a, a' \rangle$ and $\langle b, b' \rangle \in R$ and no other nodes in $\pi_{a',b'}$ are supported (see Figure 2.12 (a) for an illustration). If there exist $c' \in M_{a',b'}$ and $c \in T_x$ such that $\langle c, c' \rangle \in R$, then we have $\langle m_x(a, b, c), m_y(a', b', c') \rangle \in R$. It is clear that $m_y(a', b', c')$ is a node in $\pi_{a',b'}$ which is different from a' and b'. This is a

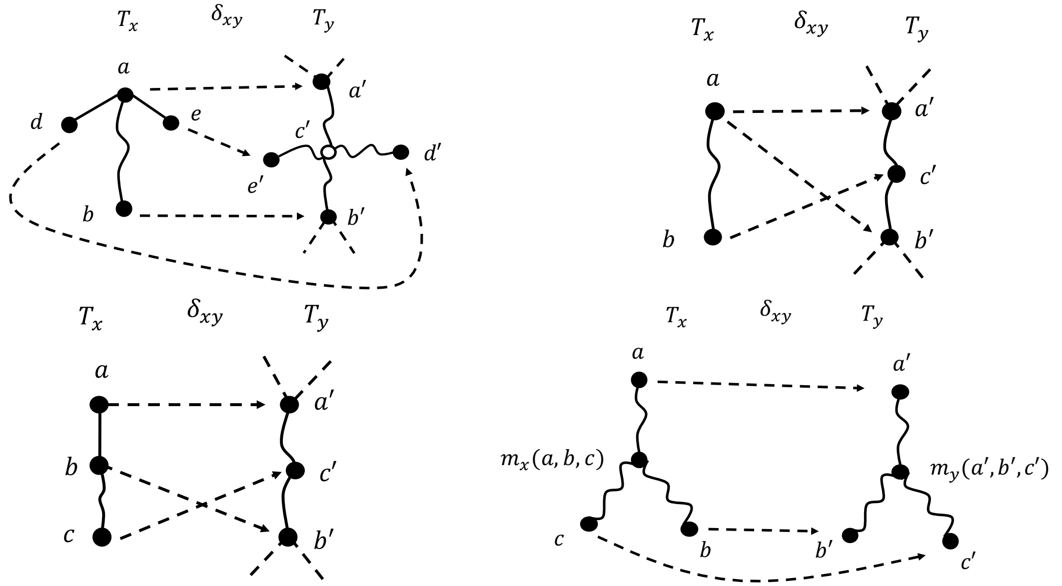


Figure 2.12 : Illustration of proof of proposition 2.8. Broken arrow lines indicate images of the node and empty circles indicates empty support nodes.

contradiction and hence the statement is correct. □

Let $T'_y = T_y \ominus M_{a',b'}$ and m'_y the standard majority operation on tree T'_y. Based on Lemma 2.9, it is easy to see that, when restricted to T'_y, R and its inverse are also closed under $\{m_x, m'_y\}$. If we continue contracting and revising T_y and T_x in this way, then, in finite steps, we will reach a state in which every node is supported. Write the revised tree domains in this state as T^*_x and T^*_y and let m^*_x and m^*_y be their corresponding standard majority operations. Then, when restricted to T^*_x and T^*_y, R and R^{-1} are closed under $\{m^*_x, m^*_y\}$. This suggests that it is reasonable to consider relations that are arc-consistent.

Based upon this observation, we have the following characterisation.

Proposition 2.8. *Let T_x and T_y be two nonempty tree domains and m_x and m_y their standard majority operations. Suppose $R \subseteq T_x \times T_y$ is a nonempty relation such that both R and its inverse, R^{-1}, are arc-consistent. Then R is closed under $\{m_x, m_y\}$ iff both R and R^{-1} are tree-preserving w.r.t. T_x and T_y.*

Proof. Suppose R is closed under $\{m_x, m_y\}$. We first prove that R_{xy} is tree-convex. Suppose not. Then we have $a \in T_x$, $a', b', c' \in T_y$ such that $\langle a, a'\rangle, \langle a, b'\rangle \in R$, $c' \in \pi_{a',b'}$ but $\langle a, c'\rangle \notin R$ (see Figure 2.12 (b) for an illustration). Because R^{-1} is arc-consistent, there exists $c \in T_x$ such that $\langle c, c'\rangle \in R$. Consider the three tuples $\langle a, a'\rangle, \langle a, b'\rangle, \langle c, c'\rangle$ in R. We have $\langle m_x(a, a, c), m_y(a', b', c')\rangle \in R$ as R is closed under $\{m_x, m_y\}$. Since $m_x(a, a, c) = a$ and $m_y(a', b', c') = c'$, we have $\langle a, c'\rangle \in R$, which is a contradiction. Therefore, R_{xy} is tree-convex. Next, we prove that R is consecutive. Suppose not. Then there exist two neighbouring nodes $a, b \in T_x$ such that $R(a) \cup R(b)$ is not a subtree of T_y. This means that there are $a' \in R(a)$, $b' \in R(b)$, and $c' \in T_y$ such that c' is in $\pi_{a'b'}$ but not in either $R(a)$ or $R(b)$. Because R^{-1} is arc-consistent, there exists c in T_x such that $\langle c, c'\rangle \in R$ (see Figure 2.12 (c) for an illustration). Consider the three tuples $\langle a, a'\rangle, \langle b, b'\rangle, \langle c, c'\rangle$ in R. We have $\langle m_x(a, b, c), m_y(a', b', c')\rangle \in R_{xy}$. Because a is a neighbour of b, $m_x(a, b, c) = \pi_{ab} \cap \pi_{bc} \cap \pi_{ac}$ is either a or b. Therefore, we have either $\langle a, c'\rangle \in R$ or $\langle b, c'\rangle \in R$, which is a contradiction. Therefore, R_{xy} is arc-consistent and consecutive tree-convex and, hence, tree-preserving. Similarly, R^{-1} is tree-preserving since it is also closed under $\{m_x, m_y\}$.

On the other hand, suppose both R and R^{-1} are tree-preserving. We prove that R is closed under $\{m_x, m_y\}$. Take three arbitrary tuples $\langle a, a'\rangle, \langle b, b'\rangle, \langle c, c'\rangle$ from R. We need to prove $\langle m_x(a, b, c), m_y(a', b', c')\rangle \in R$. For convenience, we denote m, m' for $m_x(a, b, c)$ and $m_y(a', b', c')$ respectively. Because R is tree-preserving, from $a', b' \in R(\pi_{a,b})$, we know $\pi_{a',b'}$ is contained in $R(\pi_{a,b})$. In particular, $m' \in R(\pi_{a,b})$. Similarly, we also have $m' \in R(\pi_{a,c})$ and $m' \in R(\pi_{b,c})$. Note that $\pi_{a,b} = \pi_{a,m} \cup \pi_{b,m}$, $\pi_{b,c} = \pi_{b,m} \cup \pi_{c,m}$, and $\pi_{a,c} = \pi_{a,m} \cup \pi_{c,m}$. We know m' belongs to at least two of $R(\pi_{a,m})$, $R(\pi_{b,m})$, and $R(\pi_{c,m})$. Suppose $m' \notin R(\pi_{c,m})$. Then $\langle m, m'\rangle \notin R$ and $m' \in R(\pi_{a,m})$ and $m' \in R(\pi_{b,m})$. There are $a_1 \in \pi_{a,m}$ and $b_1 \in \pi_{b,m}$ such that $\langle a_1, m'\rangle \in R$ and $\langle b_1, m'\rangle \in R$. Since R^{-1} is tree-preserving, $R^{-1}(m')$ is a subtree of

T_x. From $\pi_{a_1,b_1} \subseteq R^{-1}(m')$ and $m \in \pi_{a_1,b_1}$, we know $m \in R^{-1}(m')$, i.e., $\langle m, m' \rangle \in R$, which is a contradiction. Therefore, the assumption that $m' \notin R(\pi_{c,m})$ is incorrect. This implies that m' belongs to each of $R(\pi_{a,m})$, $R(\pi_{b,m})$, and $R(\pi_{c,m})$. We therefore have $a_1 \in \pi_{a,m}$, $b_1 \in \pi_{b,m}$, and $c_1 \in \pi_{c,m}$ such that $\langle a_1, m' \rangle$, $\langle b_1, m' \rangle$, and $\langle c_1, m' \rangle$ are all in R. Let t be the subtree spanned by a_1, b_1, c_1 in T_x. Then t is contained in $R^{-1}(m')$ since R^{-1} is tree-preserving. From $m \in t$, we have the desired result that $\langle m, m' \rangle \in R$. □

Remark 2.2. *Proposition 2.8 establishes the connection between tree-preserving constraints and those binary constraints that are closed under the standard majority operation. Using this result, it is natural to extend the definition of tree-preserving constraints to non-binary tree-preserving constraints. For a non-binary relation R, assuming that it is arc-consistent in each variable, we may call R a tree-preserving constraint if it is closed under the standard majority operation induced by the relevant tree domains.*

Using Proposition 2.8, we now give an alternative proof for Theorem 2.2.

Proof of Theorem 2.2: Let $\mathcal{N} = \{x_i R_{ij} x_j \mid 1 \leq i, j \leq n\}$ be a tree-preserving constraint network. We note that if no inconsistency is detected after enforcing arc-consistency, then $\mathcal{N}$ remains tree-preserving (see Proposition 2.2). Without loss of generality, we suppose $\mathcal{N}$ is arc-consistent. Write T_i for the tree-domain of variable x_i. Let Γ be the set of binary relations over $\mathcal{D} = \{T_i \mid 1 \leq i \leq n\}$ that are closed under the multi-sorted operation $f = (m_{x_i} \mid 1 \leq i \leq n)$, where each m_{x_i} is the standard majority operation over T_i. By Proposition 2.8, every relation R_{ij} in $\mathcal{N}$ is closed under $\{m_{x_i}, m_{x_j}\}$. That is, each R_{ij} is a relation in Γ and thus $\mathcal{N}$ is an instance of C_Γ. By Theorem 1.3, we have the result that establishing strong path-consistency ensures global consistency for $\mathcal{N}$.

Proposition 2.8 considers only tree domains. We can also generalize it to forest

domains.

Definition 2.10. *Given a forest domain* F_x *with trees* $\{t_1, ..., t_n\}$*, the* standard majority operation m_x *for* F_x *is defined as:*

$$m_x(a,b,c) = \begin{cases} \pi_{ab} \cap \pi_{bc} \cap \pi_{ac} & \text{if } a,b,c \in t_i \ (1 \leq i \leq n), \\ a & \text{if } a,b \in t_i,\ c \in t_j;\ \text{or } a,c \in t_i,\ b \in t_j \ (i \neq j), \\ b & \text{if } a \in t_i,\ b,c \in t_j \ (i \neq j), \\ a & \text{if } a \in t_i,\ b \in t_j,\ c \in t_k \ (i \neq j \neq k). \end{cases}$$

We note that the order of a, b, c matters. That is, for example, $m_x(a, b, c)$ may not be the same as $m_x(b, c, a)$.

Proposition 2.9. *If R and its inverse R^{-1} are both arc-consistent and tree-preserving over forest domains F_x and F_y, then R and R^{-1} are closed under $\{m_x, m_y\}$ and $\{m_y, m_x\}$ respectively.*

Proof. Because R and R^{-1} are arc-consistent and tree-preserving, there is a bijection between trees in $F_x = \{t_1, t_2, ..., t_n\}$ and trees in $F_y = \{t'_1, t'_2, ..., t'_n\}$ such that $R(t_i) = t'_i$ and $R^{-1}(t'_i) = t_i$ for every $1 \leq i \leq n$. For any $\langle a, a' \rangle, \langle b, b' \rangle, \langle c, c' \rangle \in R$, consider the following cases separately.

(1) Suppose $a, b, c \in t_i$ for some $t_i \in F_x$. Because R and R^{-1} are tree-preserving, a', b', c' are all in t'_i. Then, by Proposition 2.8, $\langle m_x(a, b, c), m_y(a', b', c') \rangle \in R$.

(2) Suppose $a, b \in t_i$ and $c \in t_j$, or $a, c \in t_i$ and $b \in t_j$ for some $i \neq j$. We have $a', b' \in t'_i$ and $c' \in t'_j$, or $a', c' \in t'_i$ and $b' \in t'_j$. Thus $\langle m_x(a, b, c), m_y(a', b', c') \rangle = \langle a, a' \rangle \in R$.

(3) Suppose $a \in t_i$ and $b, c \in t_j$ for some $i \neq j$. We have $a' \in t'_i$ and $b', c' \in t'_j$. Thus $\langle m_x(a, b, c), m_y(a', b', c') \rangle = \langle b, b' \rangle \in R$.

(4) Suppose $a \in t_i, b \in t_j$ and $c \in t_k$ for some pairwise different i, j, k. We have $a' \in t'_i, b' \in t'_j$ and $c' \in t'_k$. Thus $\langle m_x(a,b,c), m_y(a',b',c')\rangle = \langle a, a'\rangle \in R$. □

Remark 2.3. *For the 39 different relations in the trihedral scene labelling problem, 29 of them are tree-preserving. These 29 relations are all closed under the standard majority operations defined above for forest domains. As we have expected, the other ten relations are not closed under the standard majority operations.*

2.8 Evaluations

In this section, we report experimental evaluations of local consistency enforcing algorithms for *sparse* tree-preserving constraint networks.

As we have shown in Section 4, enforcing strong PPC is sufficient to solve tree-preserving constraint networks. Although enforcing PPC should be generally faster than enforcing PC for sparse constraint networks [13], for practical interests, we conduct experimental comparisons between PPC algorithms and their counterparts, PC algorithms, for solving sparse tree-preserving constraint networks. We consider two competitive strong PC algorithms, sDC1 [82, 83] and PC2001 [12], for comparisons. It is worthwhile to note that sDC1 is a strong *dual consistency* (DC) enforcing algorithm and DC has been shown to be equal to PC for binary constraint networks [82, 83]. Just like PC, DC considers every distinct pair of variables of binary constraint networks. However, *conservative* DC (CDC) only considers a pair $\{v_i, v_j\}$ if there is a constraint $R_{v_i v_j}$ imposed on it. As Proposition 2.6 suggests strong CDC is equivalent to strong PPC for tree-preserving constraint networks with triangulated constraint graphs, we adopt the efficient strong CDC enforcing algorithm sCDC1 to enforce strong PPC on such networks. We also devise another strong PPC algorithm, called PPC2001, upon the algorithm PC2001. We do it by modifying PC2001 to enforce PC on a triangulation instead of the completion of the input constraint graph.

By Proposition 2.8, the class of tree-preserving constraints is closed under the standard majority operation, and thus tree-preserving constraint networks can also be solved by enforcing *singleton arc-consistency* (SAC) (see e.g., [23]). We also include two state-of-the-art SAC algorithms, SAC3-SDS [9] and SAC-opt [10], for comparisons. However, it is worthwhile to note that it is unknown whether enforcing SAC would enable backtrack-free search for tree-preserving constraint networks [1].

By Proposition 2.6, a random tree-preserving constraint network can be generated as follows:

(1) For every domain D_x of the network, generate a random tree T_x for with vertex set D_x.

(2) For each pair of variables x and y, generate an arc-consistent relation $R_{xy} \subseteq D_x \times D_y$ s.t. R_{xy} is closed under $\{m_x, m_y\}$.

Four parameters are used to generate a random tree-preserving network: (1) n - the number of variables, (2) d - the size of the domains, (3) ρ - the density of the constraint graph (i.e., the ratio of non-universal constraints to n^2), (4) l - the looseness of the constraints (i.e., the ratio of the number of allowed tuples to d^2).

All constraints are represented as Boolean matrices. Experimentation was carried out on a computer with an Inter Core i5 processor with a frequency of 2.9 GHz per CPU core, 8 GB of RAM, and the MAC OSX. The experimental platform is Eclipse with JDK 8.

Experimental results are presented in Figure 2.13, where each test is averaged over 20 instances. We can observe that lower densities of constraint graphs do not benefit the performances of sCDC1 and PC2001 much. On the other hand,

[1] We compare our algorithm with SAC algorithms in a sense that when only consistency checking task is considered, whether our algorithm still outperforms the state-of-the-art SAC algorithms on tree-preserving constraint networks

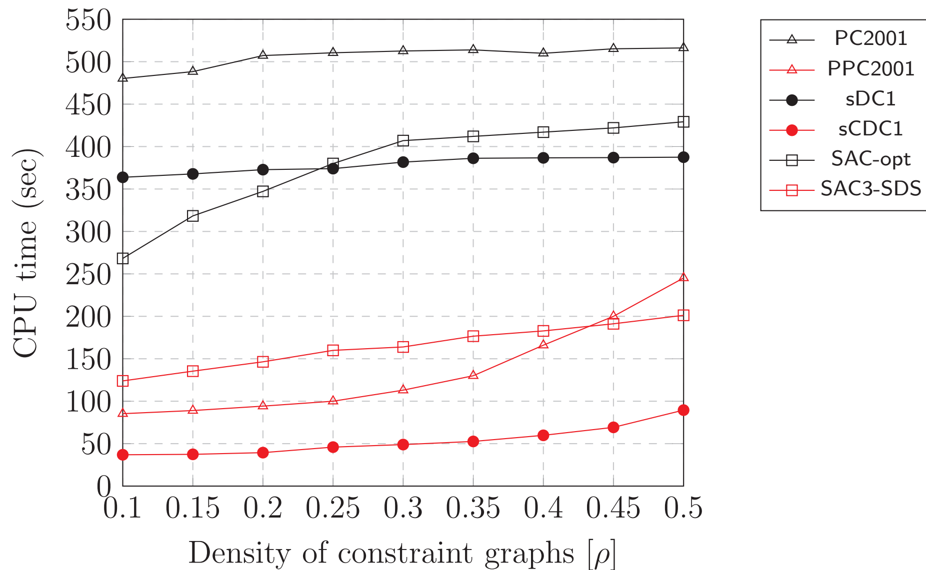


Figure 2.13 : Performance evaluation of different local consistency algorithms for solving tree-preserving constraint networks with different densities. We set $n = 100$, $d = 30$ and $l = 0.5$.

SAC3-SDS, SAC-opt, sCDC1 and PPC2001 are all exploiting the sparsity of constraint networks. In particular, they all perform better when constraint networks are sparser. The PPC algorithms sCDC1 and PPC2001 can outperform their counterparts, the PC algorithms sCD1 and PC2001, by up to a factor of 7 and 3.5 respectively. Moreover, sCDC1 beats all the other considered algorithms.

SAC3-SDS performs reasonably well for sparse tree-preserving constraint networks and is comparable to PPC2001. It also outperforms SAC-opt roughly by a factor of 2, but its performance is about twice worse than that of sCDC1. Unexpectedly, sDC1 is comparable to SAC-opt for sparse tree-preserving constraint networks, because SAC is a weaker consistency condition than PC and thus SAC algorithms are usually expected to be more efficient than PC algorithms.

2.9 Conclusion

The study of tractable subclasses of constraint satisfaction problems is one of the most important research problems in artificial intelligence. In this chapter, we studied three tractable subclasses of tree-convex constraints, which are generalizations of the well-known row-convex constraint. We proved that enforcing strong path-consistency decides the consistency of a tree-preserving constraint network and, if no inconsistency is detected, transforms the network into a globally consistent constraint network. Actually, we proved this by two methods. The first method directly proved that enforcing strong path-consistency transforms a tree-preserving constraint network into a path-consistent tree-preserving network, while the second method relied on the characterisation of tree-preserving constraints by closure under majority operations. Since every arc-consistent chain- or path-preserving constraint is a tree-preserving constraint, we got a tractable subclass of CSPs that is genuinely larger than the subclass of CRC constraints. We further showed that PPC algorithms can be applied to solve tree-preserving constraint networks in a backtrack-free style, which is more efficient than using a standard path-consistency algorithm. As an application, we proved that every relation used in the trihedral scene labelling problem can be modelled by a tree-convex constraint, and, among these different relations (39 in total), 29 are tree-preserving constraints. This means that a large tractable subclass of the NP-hard trihedral scene labelling problem can be solved by the techniques discussed in this chapter.

Appendix to This Chapter: Proofs

Lemma 5 *Let $\mathcal{N}$ be an arc-consistent and tree-preserving constraint network over tree domains T_x ($x \in V$). Suppose $x \in V$ and $M_{a,b}$ is a trunk in T_x. When restricted to $T_x \ominus M_{a,b}$ and enforcing arc-consistency, $\mathcal{N}$ remains tree-preserving if no inconsistency is detected.*

Proof. The proof of this result heavily uses Lemma 2.13 and Proposition 2.2.

First, we consider how trunks propagate in the network. Starting from the trunk $M_{a,b}$ in T_x, we get, as in Lemma 2.13, a unique trunk M_{a_y,b_y} in T_y for each $y \neq x$ in V if $R_{xy}(a) \cup R_{xy}(b)$ is not connected in T_y. Furthermore, each trunk M_{a_y,b_y} (in T_y) will also propagate in the network, obtaining a (possibly new) trunk $M_{a_{yz},b_{yz}}$ in T_z for each $z \neq y$. Continuing this way, we stop until no new trunks are generated. Since there are finitely many different trunks in each tree domain, the process will stop in a finite number of steps. Write $\mathcal{T}_y$ for the set of trunks obtained for variable y. We note that nodes in these trunks have to be deleted from the corresponding tree domain to maintain arc-consistency.

We next amalgamate these trunks in each $\mathcal{T}_y$. By Lemma 2.11, the union of two connected trunks can be the whole tree, a branch, or a larger trunk. Similarly, we can prove that if a trunk and a branch are connected, then their union is a branch, a trunk, or the whole tree. If the union of all trunks in a $\mathcal{T}_y$ is the whole tree, viz. T_y, then the network is inconsistent. In the following, we assume that this is not the case. This implies that the trunks in $\mathcal{T}_y$ can be merged into a set of pairwise disconnected maximal trunks and maximal branches. Let t_y be the subtree of T_y obtained after removing all these maximal branches. We now restrict the constraint network to subtrees t_y $(y \in V)$ and enforce arc-consistency. By Corollary 2.1 and Proposition 2.2, we get a new arc-consistent tree-preserving network $\mathcal{N}'$ over smaller tree domains, say T'_y $(y \in V)$, if no inconsistency is detected.

Consider the original trunk $M_{a,b}$ in T_x. If $M_{a,b} \cap T'_x = \varnothing$ or $T'_x \subseteq M_{a,b}$, then we need do nothing as the network is either tree-preserving or trivially inconsistent after contracting $M_{a,b}$. If $M_{a,b} \cap T'_x$ is a branch of T'_x, then we use Corollary 2.1 again and transform $\mathcal{N}'$ into a new arc-consistent and tree-preserving network if no inconsistency is detected. If neither of the above happens, then $M_{a,b} \cap T'_x$ is a

trunk in T'_x. We repeat the above process again and again until no new branches are generated.

From now on, we suppose that no branches are obtained by merging trunks in any $\mathcal{T}_y$. Furthermore, for each variable y, we denote by $\mathcal{MT}_y$ the set of maximal trunks of T_y after amalgamation. We contract the maximal trunks in $\mathcal{MT}_y$ one by one and write T^*_y for the contracted tree.

For any two variables $y \neq z$, we show that R_{yz}, when restricted to T^*_y and T^*_z, remains tree-preserving. Suppose $\mathcal{MT}_y = \{M_{a_1,b_1}, ..., M_{a_k,b_k}\}$ and assume, without loss of generality, that the first $m \leq k$ trunks satisfy the precondition of Lemma 2.13, i.e., $R_{yz}(a_i) \cup R_{yz}(b_i)$ is disconnected for $1 \leq i \leq m$. By Lemma 2.13, there exists a unique trunk $M_{a'_i,b'_i}$ in T_z such that $a'_i \in R_{yz}(a_i)$, $b'_i \in R_{yz}(b_i)$ and $R_{zy}(M_{a'_i,b'_i}) \subseteq M_{a_i,b_i}$ for each $1 \leq i \leq m$.

Let $\mathcal{MN}_y = \{M_{a_1,b_1}, ..., M_{a_m,b_m}\}$ and $\mathcal{N}_z = \{M_{a'_1,b'_1}, ..., M_{a'_m,b'_m}\}$. Note that trunks in $\mathcal{N}_z$ are not necessarily maximal. Let $M_{a''_i,b''_i}$ be the maximal trunk in $\mathcal{MT}_z$ which contains $M_{a'_i,b'_i}$.[1] We write $\mathcal{MN}_z = \{M_{a''_1,b''_1}, ..., M_{a''_m,b''_m}\}$. We now show how to contract these trunks so that we get T^*_y and T^*_z while preserving the tree-preserving property.

First, we contract all maximal trunks in $\mathcal{MT}_y$ that are not in $\mathcal{MN}_y$. Because the images of the two nodes a_i, b_i under R_{yz} are connected in T_z, the constraint R_{yz} remains tree-preserving after the contraction. Let T'_y be the resultant tree domain of y.

Second, we contract all maximal trunks in $\mathcal{MN}_y$ from T'_y and contract all corresponding trunks in $\mathcal{N}_z$ from T_z. Clearly, the resultant tree domain of y is exactly T^*_y. Denote by T'_z the resultant tree domain of z. By Lemma 2.13, R_{yz} is tree-preserving

[1] Since no branches can be obtained by merging trunks in $\mathcal{T}_z$, we know that $M_{a'_i,b'_i}$ is contained in a maximal trunk.

when restricted to T_y^* and T_z'. Note that, after the contraction of $M_{a_i',b_i'}$ from T_z, $M_{a_i'',b_i''}$ becomes a trunk in T_z'. We then contract all these trunks together with all other maximal trunks in $\mathcal{MT}_z$ from T_z'. The resultant tree domain is exactly T_z^*.

We show that R_{yz} is still tree-preserving when restricted to T_y^* and T_z^*. Given any subtree t of T_y^*, because R_{yz} is tree-preserving w.r.t. T_y^* and T_z', we know $t' \equiv R_{yz}(t)$ is a nonempty subtree of T_z'. By Lemma 2.12 we know t' is a (possibly empty) subtree of T_z^*. This proves that R_{yz} is still tree-preserving when restricted to T_y^* and T_z^*.

Because the arbitrariness of y, z above, we know every R_{uw}, in particular R_{zy}, is tree-preserving w.r.t. T_u^* and T_w^*. Note that these constraints are not necessarily arc-consistent. By Proposition 2.2 again, we transform these constraints into arc-consistent constraints while remaining tree-preserving. In summary, we know that, when restricted to $T_x \ominus M_{a,b}$ and enforcing arc-consistency, $\mathcal{N}$ remains tree-preserving if no inconsistency is detected. □

Lemma 6 *Assume R_{xy} and R_{xy}' are two arc-consistent and tree-preserving constraints w.r.t. trees T_x and T_y. Let $R_{xy}^* = R_{xy} \cap R_{xy}'$. Let $W = \{w \in T_x \mid R_{xy}^*(w) \neq \varnothing\}$ be the set of supported values of R_{xy}^*. Suppose $u \in T_x$ and $u \notin W$. Then there exist at most two values w_1, w_2 in W s.t. no value in W other than w_i is on the path $\pi_{w_i,u}$ for $i = 1, 2$.*

Proof. Suppose w_1, w_2, w_3 are three supported values of R_{xy}^* in T_x s.t. no value in W other than w_j is on the path $\pi_{w_j,u}$ for $1 \leq j \leq 3$. Take $w_i' \in R_{xy}^*(w_i)$ $(i = 1, 2, 3)$. Let $\{u_1\} = \pi_{w_1,w_2} \cap \pi_{w_1,w_3} \cap \pi_{w_2,w_3}$ and $\{u_1'\} = \pi_{w_1',w_2'} \cap \pi_{w_1',w_3'} \cap \pi_{w_2',w_3'}$. By the choice of w_1, w_2, w_3, they cannot be on a same path. In particular, u_1 is different from w_1, w_2, w_3. Furthermore, since u_1 is on π_{w_1,w_2}, it must be on either π_{u,w_1} or π_{u,w_2}. In either case, we have u_1 is not in W.

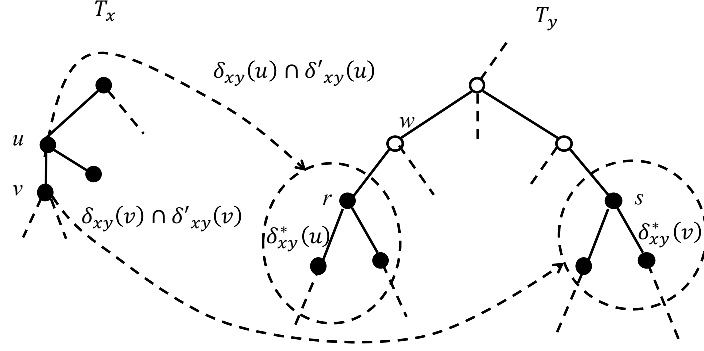


Figure 2.14 : Illustration of proof of Lemma 2.7.

Because R_{xy} is tree-preserving, from $w_i, w_j \in R_{yx}(\pi_{w'_i,w'_j})$, we know π_{w_i,w_j} is contained in $R_{yx}(\pi_{w'_i,w'_j})$ for any $1 \leq i \neq j \leq 3$. In particular, u_1 is in $R_{yx}(\pi_{w'_i,w'_j})$. In other words, $R_{xy}(u_1) \cap \pi_{w'_i,w'_j} \neq \varnothing$. By Lemma 2.1, we know $R_{xy}(u_1) \cap \pi_{w'_1,w'_2} \cap \pi_{w'_1,w'_3} \cap \pi_{w'_2,w'_3} \neq \varnothing$. Because $\{u'_1\} = \pi_{w'_1,w'_2} \cap \pi_{w'_1,w'_3} \cap \pi_{w'_2,w'_3}$, we know $u'_1 \in R_{xy}(u_1)$. Analogously, we have $u'_1 \in R'_{xy}(u_1)$ and, hence, u_1 is a value in W. This is a contradiction. Therefore, there exist at most two values w_1, w_2 in W s.t. no value in W other than w_j is on the path $\pi_{w_j,u}$ for $i = 1, 2$. □

Lemma 7 *Suppose R_{xy} and R'_{xy} are arc-consistent and tree-preserving constraints w.r.t. trees T_x and T_y and so are R_{yx} and R'_{yx}. Let $R^*_{xy} = R_{xy} \cap R'_{xy}$. Assume $\{u, v\}$ is an edge in T_x s.t. $R^*_{xy}(u) \neq \varnothing$, $R^*_{xy}(v) \neq \varnothing$, and $R^*_{xy}(u) \cup R^*_{xy}(v)$ is disconnected in T_y. Then there exist unique $r \in R^*_{xy}(u)$ and $s \in R^*_{xy}(v)$ s.t. every node in $M_{r,s}$ is unsupported under R^*_{yx}.*

Proof. Write $T_r = R^*_{xy}(u)$ and $T_s = R^*_{xy}(v)$. Clearly, T_r and T_s are nonempty subtrees of T_y. Since they are disconnected, there exist (unique) $r \in T_r$, $s \in T_s$ s.t. $\pi_{r,s} \cap (T_r \cup T_s) = \{r, s\}$ (see Figure 2.14 for an illustration). Write $A = R_{xy}(u)$, $B = R_{xy}(v)$, $C = R'_{xy}(u)$ and $D = R'_{xy}(v)$. We show every node in $M_{r,s}$ is not supported under R^*_{yx}.

Suppose w is an arbitrary internal node on $\pi_{r,s}$. We first show w is not supported

under R^*_{yx}. Note $w \in A \cup B$, $w \in C \cup D$, $w \notin A \cap C$, and $w \notin B \cap D$. There are two cases according to whether $w \in A$. If $w \in A$, then we have $w \notin C$, $w \in D$, and $w \notin B$. If $w \notin A$, then we have $w \in B$, $w \notin D$, and $w \in C$. Suppose w.l.o.g. $w \in A$. By $w \in A = R_{xy}(u)$, we have $u \in R_{yx}(w)$; by $w \notin B = R_{xy}(v)$, we have $v \notin R_{yx}(w)$. Similarly, we have $u \notin R'_{yx}(w)$ and $v \in R'_{yx}(w)$. Thus subtree $R'_{yx}(w)$ is disjoint from subtree $R_{yx}(w)$. This shows $R^*_{yx}(w) = \varnothing$ and hence w is not supported under R^*_{yx}.

Second, suppose w_1 is an arbitrary node in $M_{r,s}$ s.t. w_1 is in a different branch of w to r and s, i.e., $\pi_{w,w_1} \cap (T_r \cup T_s) = \varnothing$. We show w_1 is not supported under R^*_{yx} either.

Again, we assume $w \in A$. In this case, we have $u \in R_{yx}(w) \subseteq R_{yx}(\pi_{w,w_1})$ and $v \in R'_{yx}(w) \subseteq R'_{yx}(\pi_{w,w_1})$. As $\pi_{w,w_1} \cap (T_r \cup T_s) = \varnothing$, we have $\pi_{w,w_1} \cap T_r = \pi_{w,w_1} \cap A \cap C = \varnothing$. As $\pi_{w,w_1} \cap A \neq \varnothing$ and $A \cap C \neq \varnothing$, by Lemma 2.1, we must have $\pi_{w,w_1} \cap R'_{xy}(u) = \varnothing$. This shows $u \notin R'_{yx}(\pi_{w,w_1})$. Similarly, we can show $v \notin R_{yx}(\pi_{w,w_1})$. Thus subtree $R'_{yx}(\pi_{w,w_1})$ is disjoint from subtree $R_{yx}(\pi_{w,w_1})$ and, hence, $R^*_{yx}(\pi_{w,w_1}) = \varnothing$. This proves that w_1 is unsupported under R^*_{yx} either.

In summary, every node in $M_{r,s}$ is unsupported. □

The following simple properties are used to assist the proofs of Lemmas 2.5-2.7.

Lemma 2.10. *Suppose $M_{a,b}$ and u are, respectively, a trunk and a node of tree T. Let $t_a = \{w \in T \mid a \in \pi_{w,b}\}$ and $t_b = \{w \in T \mid b \in \pi_{w,a}\}$. Then*

(i) $a \in t_a$, $b \in t_b$, $a, b \notin M_{a,b}$. Moreover, t_a and t_b are subtrees of T separated by $M_{a,b}$ and $\{t_a, t_b, M_{a,b}\}$ is a partition of T.

(ii) $u \notin M_{a,b}$ iff $a \in \pi_{u,b}$ or $b \in \pi_{u,a}$; if $u \in M_{a,b}$, then $\pi_{u,a} \subseteq M_{a,b} \cup \{a\}$ and $\pi_{u,b} \subseteq M_{a,b} \cup \{b\}$.

Proof. Take a as the root of T. Let a_1 be the child of a that is on the path $\pi_{a,b}$. Then t_a is the subtree obtained by removing the subtree rooted at a_1, and t_b is the subtree rooted at b. The results are then clear. □

The next lemma considers the union of two connected trunks.

Lemma 2.11. *Suppose $M_{a,b}$ and $M_{c,d}$ are two trunks of tree $T = (V, E)$. Then $M_{a,b} \cup M_{c,d}$ is connected if and only if either (i) $M_{a,b} \cap M_{c,d} \neq \varnothing$ or (ii) there exist $x \in \{a, b\}$, $y \in \{c, d\}$ s.t. $\{x, y\}$ is an edge in T and $x \in M_{c,d}, y \in M_{a,b}$. Moreover, if $M_{a,b} \cup M_{c,d}$ is connected, then it is T, or a trunk or branch of T.*

Proof. If either (i) or (ii) holds, then clearly $M_{a,b} \cup M_{c,d}$ is connected. Suppose $M_{a,b} \cup M_{c,d}$ is connected but $M_{a,b} \cap M_{c,d} = \varnothing$, we show (ii) holds. Because $M_{a,b} \cup M_{c,d}$ is connected, there exist $u \in M_{a,b}$, $v \in M_{c,d}$ such that $\{u, v\}$ is an edge in T. In addition, because $M_{a,b} \cap M_{c,d} = \varnothing$, we have $v \notin M_{a,b}$, $u \notin M_{c,d}$. Then by Lemma 2.10, we know $a \in \pi_{v,b}$ or $b \in \pi_{v,a}$, and $c \in \pi_{u,d}$ or $d \in \pi_{u,c}$. Without loss of generality, suppose $a \in \pi_{v,b}$ and $c \in \pi_{u,d}$. From $a \in \pi_{v,b}$ and $u \in M_{a,b}$, we have $u \in \pi_{v,b}$. Because u is a neighbour of v, this is possible only if $v = a$ and $u \in \pi_{a,b}$. Analogously, we also have $u = c$ and $v \in \pi_{c,d}$. Therefore, we have $\{a, c\} \in E$, $c \in M_{a,b}$ and $a \in M_{c,d}$. Note that in this case $M_{a,b} \cup M_{c,d} = M_{b,d}$ is another trunk.

We next suppose $M_{a,b} \cap M_{c,d} \neq \varnothing$ and $c, d \notin M_{a,b}$. We show that $a \in M_{c,d}$ iff $b \in M_{c,d}$. Suppose otherwise $a \in M_{c,d}$ but $b \notin M_{c,d}$. Then by $b \notin M_{c,d}$ and Lemma 2.10 we have either $c \in \pi_{b,d}$ or $d \in \pi_{b,c}$. In either case, by $a \in M_{c,d}$ and Lemma 2.10, we have $c \in \pi_{a,b}$ or $d \in \pi_{a,b}$, which contradicts the assumption that $c, d \notin M_{a,b}$. Thus, we have $a \in M_{c,d}$ iff $b \in M_{c,d}$.

We now consider the following possible cases (symmetric cases are omitted):

(1) Suppose $c, d \notin M_{a,b}$ and $a, b \notin M_{c,d}$. By Lemma 2.10, we have $a \in \pi_{c,b}$ or $b \in \pi_{c,a}$, $a \in \pi_{d,b}$ or $b \in \pi_{d,a}$, $c \in \pi_{a,d}$ or $d \in \pi_{a,c}$, and $c \in \pi_{b,d}$ or $d \in \pi_{b,c}$. Since

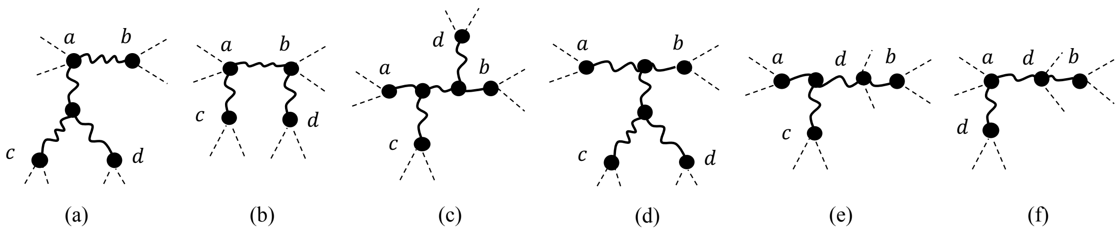


Figure 2.15 : Possible configurations of two connected trunks $M_{a,b}$ and $M_{c,d}$, where $M_{a,b} \cup M_{c,d}$ is the whole tree in (c) and (d), a trunk in (a), (b) and (f), and a branch in (e).

$M_{a,b} \cap M_{c,d} \neq \varnothing$, it is straightforward to show that $\{a, b\} = \{c, d\}$ and $M_{a,b} = M_{c,d}$.

(2) Suppose $c, d \notin M_{a,b}$ and $a, b \in M_{c,d}$. Because $c, d \notin M_{c,d}$, we know a, b, c, d are pairwise different. Moreover, we have $a \in \pi_{c,b}$ or $b \in \pi_{c,a}$, and $a \in \pi_{d,b}$ or $b \in \pi_{d,a}$. We can prove that $M_{a,b} \cup M_{c,d} = M_{c,d}$ is a trunk. There are two possible configurations (see Figure 2.15 (a) and (b)).

(3) Suppose $c, d \in M_{a,b}$ and $a, b \in M_{c,d}$. Then, a, b, c, d are pairwise different. We can prove that $M_{a,b} \cup M_{c,d}$ is the whole tree. There are two possible configurations (see Figure 2.15(c) and (d)).

(4) Suppose $c, d \in M_{a,b}$, $a \in M_{c,d}$, but $b \notin M_{c,d}$. Then a, b, c, d are pairwise different. We can prove that $M_{a,b} \cup M_{c,d}$ is a branch of b. There is only one possible graph (see Figure 2.15(e)).

(5) Suppose $c \in M_{a,b}, d \notin M_{a,b}$, $a \in M_{c,d}$, and $b \notin M_{c,d}$. Then $c \neq a, b$ and $a \neq c, d$, but it is possible that $b = d$. In this case, we can prove that $a, c \in \pi_{d,b}$, $a \in \pi_{d,c}$, and $c \in \pi_{a,b}$ (see Figure 2.15(f)). Moreover, we have $M_{a,b} \cup M_{c,d} = M_{d,b}$ is a trunk. □

The next lemma considers what happens to a subtree when we contract a trunk from the tree domain.

Lemma 2.12. *Suppose $M_{a,b}$ and t are, respectively, a trunk and a subtree of tree T.*

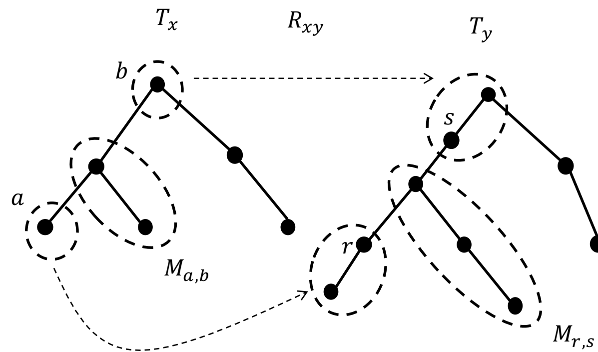


Figure 2.16 : Illustration of proof of Lemma 2.13.

If t is not contained in $M_{a,b}$, then t, when restricted to $T \ominus M_{a,b}$, is also a subtree of $T_x \ominus M_{a,b}$.

Proof. Let $t_a = \{u \in T \mid a \in \pi_{u,b}\}$ and $t_b = \{u \in T \mid b \in \pi_{u,a}\}$. By Lemma 2.10, $a \in t_a$, $b \in t_b$, t_a and t_b are two subtrees separated by $M_{a,b}$, and $\{t_a, t_b, M_{a,b}\}$ is a partition of T.

If $t \subseteq M_{a,b}$, then all nodes in t are deleted after the contraction of T by $M_{a,b}$; if $t \cap M_{a,b} = \varnothing$, then no nodes in t are deleted after the contraction of T by $M_{a,b}$; if $t \nsubseteq M_{a,b}$ and $t \cap M_{a,b} \neq \varnothing$, there are three subcases. First, if $t \cap t_a \neq \varnothing$ and $t \cap t_b \neq \varnothing$, then both a, b are in t. After contraction, t is the union of two subtrees $t \cap t_a$ and $t \cap t_b$, which are connected by the new edge $\{a, b\}$. Hence, t is still a subtree. Second, if $t \cap t_a \neq \varnothing$ but $t \cap t_b = \varnothing$, then $a \in t$ but $b \notin t$. After contraction, t will be replaced by $t \cap t_a$. Third, if $t \cap t_a = \varnothing$ and $t \cap t_b \neq \varnothing$, then, after contraction, t will be replaced by $t \cap t_b$. □

Given a tree-preserving constraint R_{xy} w.r.t. tree domains T_x and T_y. Suppose a, b are two nodes in T_x s.t. $R_{xy}(a) \cup R_{xy}(b)$ is not connected in T_y. We now consider how to modify T_y so that R_{xy} remains tree-preserving after contracting trunk $M_{a,b}$ from T_x.

Lemma 2.13. *Suppose R_{xy} and R_{yx} are arc-consistent and tree-preserving w.r.t.*

tree domains T_x and T_y. Let a, b be two nodes in T_x s.t. $R_{xy}(a) \cup R_{xy}(b)$ is not connected in T_y. Then there exist unique $r, s \in T_y$ s.t. $r \in R_{xy}(a)$, $s \in R_{xy}(b)$, and $R_{yx}(M_{r,s}) \subseteq M_{a,b}$. Moreover, R_{xy} and R_{yx} are tree-preserving when restricted to $T_x \ominus M_{a,b}$ and $T_y \ominus M_{r,s}$.

Proof. Choose $r \in R_{xy}(a)$ and $s \in R_{xy}(b)$ such that the path $\pi_{r,s}$ from r to s in T_y is a shortest one among $\{\pi_{r',s'} : r' \in R_{xy}(a), s' \in R_{xy}(b)\}$ (see Figure 2.16 for an illustration). In particular, we have $\pi_{r,s} \cap (R_{xy}(a) \cup R_{xy}(b)) = \{r, s\}$. We assert that the image of every node v in $M_{r,s}$ under R_{yx} is contained in $M_{a,b}$. Suppose otherwise and there exists u in $T_x \setminus M_{a,b}$ s.t. $(u, v) \in R_{xy}$. Assume that u is in the same connected component as a. Since the subtree $R_{yx}(\pi_{v,s})$ contains u and b, it also contains a. This implies that there is a node v' on $\pi_{v,s}$ which is in $R_{xy}(a)$. This is impossible as $v \in M_{r,s}$ and $R_{xy}(a) \cap \pi_{r,s} = \{r\}$. Therefore $R_{yx}(v) \subseteq M_{a,b}$ for any $v \in M_{r,s}$. Hence $R_{yx}(M_{r,s}) \subseteq M_{a,b}$ holds.

It is clear that, when restricted to $T_x \ominus M_{a,b}$ and $T_y \ominus M_{r,s}$, $R_{xy}(\{a, b\})$ is connected and so is $R_{yx}(\{r, s\})$. For any other edge $\{a', b'\}$ in $T_x \ominus M_{a,b}$, by $R_{yx}(M_{r,s}) \subseteq M_{a,b}$, $R_{xy}(\{a', b'\}) \cap M_{r,s} = \varnothing$ and the image of $\{a', b'\}$ is unchanged (hence connected) after the M-contraction of T_y. This shows that R_{xy} is consecutive when restricted to $T_x \ominus M_{a,b}$. Furthermore, for every node c in $T_x \ominus M_{a,b}$, since c is supported by a node in $T_y \ominus M_{r,s}$, we know that $R_{xy}(c)$ is a nonempty subtree in T_y. By Lemma 2.12 and $R_{xy}(c) \cap M_{r,s} = \varnothing$, we know $R_{xy}(c) \cap (T_y \ominus M_{r,s})$ is also a nonempty subtree in $T_y \ominus M_{r,s}$. This shows that R_{xy} is tree-preserving when restricted to $T_x \ominus M_{a,b}$ and $T_y \ominus M_{r,s}$. On the other hand, for any subtree t in $T_y \ominus M_{r,s}$, w.l.o.g., assume that $r, s \in t$. Then $R_{yx}(t) = R_{yx}(t \cap t_r) \cup R_{yx}(t \cap t_s)$, where t_r and t_s are the two connected components of T_y separated by $M_{r,s}$. Because both $R_{yx}(t \cap t_r)$ and $R_{yx}(t \cap t_s)$ are subtrees in T_x and, hence, subtrees in $T_x \ominus M_{a,b}$ by Lemma 2.12. By $a \in R_{yx}(t \cap t_r)$ and $b \in R_{yx}(t \cap t_s)$, we know $R_{yx}(t)$ is a subtree of $T_y \ominus M_{r,s}$. This shows that R_{yx}

is tree-preserving when restricted to $T_x \ominus M_{a,b}$ and $T_y \ominus M_{r,s}$. $\square$

It is possible that there is a node $v \in T_y \ominus M_{r,s}$ s.t. $R_{yx}(v) \subseteq M_{a,b}$, but the image of $T_x \ominus M_{a,b}$ under the restricted R_{xy} is a subtree of $T_y \ominus M_{r,s}$.

Chapter 3

CSPs Solvable with Directional PC

3.1 Contribution

Among the local consistency techniques used for solving constraint networks, path-consistency (PC) has received a great deal of attention. However, enforcing PC is computationally expensive and sometimes unnecessary. Directional path-consistency (DPC) is a weaker notion of PC that considers a given variable ordering and can thus be enforced more efficiently than PC. This chapter shows that **DPC** (the DPC enforcing algorithm of Dechter and Pearl [39]) decides the constraint satisfaction problem (CSP) of a constraint language if it is complete and has the variable elimination property (VEP). However, we also show that no complete VEP constraint language can have a domain with more than 2 values. We then present a simple variant of the **DPC** algorithm, called **DPC***, and show that the CSP of a constraint language can be decided by **DPC*** if it is closed under a majority operation. In fact, **DPC*** is sufficient for guaranteeing backtrack-free search for such constraint networks. Examples of majority-closed constraint classes include the classes of connected row-convex (CRC) constraints and tree-preserving constraints, which have found applications in various domains, such as scene labeling, temporal reasoning, geometric reasoning, and logical filtering. Our experimental evaluations show that **DPC*** significantly outperforms the state-of-the-art algorithms for solving majority-closed constraints.

3.2 Introduction and Chapter Outline

Many Artificial Intelligence tasks can be formulated as *constraint networks* [101], such as natural language parsing [93], temporal reasoning [38, 104], and spatial reasoning [84]. A constraint network comprises a set of variables ranging over some domain of possible values, and a set of constraints that specify allowed value combinations for these variables. Solving a constraint network amounts to assigning values to its variables such that its constraints are satisfied. *Backtracking search* is the principal mechanism for solving a constraint network; it assigns values to variables in a depth-first manner, and backtracks to the previous variable assignment if there are no consistent values for the variable at hand. *Local consistency* techniques are commonly used to reduce the size of the search space before commencing search. However, searching for a complete solution for a constraint network is still hard. In fact, even deciding whether the constraint network has a solution is NP-complete in general. Therefore, given a particular form of local consistency, a crucial task is to determine problems that can be solved by *backtrack-free* search using that local consistency [48].

This chapter considers a particular form of local consistency, called *path-consistency* (PC), which is one of the most important and heavily studied local consistencies in the literature (see e.g. [89, 53, 113, 13, 24]). Recently, it was shown that strong PC can be used to decide the satisfiability of a problem if and only if the problem does not have the *ability to count* [5, 4]; however, it remains unclear whether backtrack-free search can be used to extract a solution for such a problem after enforcing PC.

Directional path-consistency (DPC) [39] is a weaker notion of PC that considers a given variable ordering and can thus be enforced more efficiently than PC. The DPC enforcing algorithm of Dechter and Pearl [39], denoted by **DPC**, has been used

to efficiently solve reasoning problems in temporal reasoning [38, 104] and spatial reasoning [114]. It is then natural to ask *what binary constraint networks with finite domains can be solved by* **DPC**. Dechter and Pearl [39] showed that **DPC** is sufficient for enabling backtrack-free search for a network with a constraint graph of *regular width* 2, i.e., there exists a width 2 ordering of the constraint graph which remains width 2 after applying **DPC**. We consider the aforementioned question in the context of *constraint languages*, which is a widely adopted approach in the study of tractability of constraint satisfaction problems [18]. Specifically, we are interested in finding all binary constraint languages Γ such that the consistency of any constraint network defined over Γ can be decided by **DPC**.

To this end, we first exploit the close connection between DPC and *variable elimination* by defining constraint languages that have the (weak) variable elimination property (VEP) (which will become clear in Definition 3.4). We call a constraint language Γ *complete* if it contains all relations that are definable in Γ (in the sense of Definition 3.1). Then, we show that the CSP of a constraint language Γ can be decided by **DPC** if it is complete and has VEP, which is shown to be equivalent to the Helly property. However, we also show that no complete VEP constraint language can have a domain with more than 2 values.

We then present a simple variant of the algorithm **DPC**, called **DPC***, and show that the consistency of a constraint network can be decided by **DPC*** if it is defined over any majority-closed constraint language. In fact, we show that **DPC*** is sufficient for guaranteeing backtrack-free search for such constraint networks. Several important constraint classes have been found to be majority-closed. The most well-known one is the class of *connected row-convex (CRC)* constraints [40], which is further generalized to a larger class of *tree-preserving* constraints in Chapter 2. The class of CRC constraints has been successfully applied to temporal reasoning [73], logical filtering [76], and geometric reasoning [72], and the class of tree-preserving

constraints can model a large subclass of the scene labeling problem (cf. Chapter 2). We also conduct experimental evaluations to compare **DPC*** to the state-of-the-art algorithms for solving majority-closed constraints, and show that **DPC*** significantly outperforms the latter algorithms.

The remainder of this chapter is organized as follows. In Section 3.3 we introduce basic notions and results that will be used throughout the chapter. In Section 3.4 we present the **DPC** algorithm, and in Section 3.5 we discuss the connection between **DPC** and variable elimination. In Section 3.6 we prove that a complete constraint language Γ has weak VEP if and only if Γ is majority-closed. We then present in Section 3.7 our variable elimination algorithm **DPC***, and empirically evaluate **DPC*** in Section 3.8. Finally, Section 3.9 concludes the chapter.

3.3 Preliminaries

In this chapter we are concerned with binary constraint networks (BCNs) defined over a particular *constraint language* and we use *constraint languages*, *constraint classes* and *sets of relations* interchangeably. This chapter heavily uses closure properties of constraint relations which have been introduced in subsection 1.3.5. Other necessary notations and results about CSP and DPC are also provided in section 1.3.

Definition 3.1. *A relation R is said to be* definable *in Γ if $R \in \Gamma^+$, and a set of binary relations Γ is said to be* complete *if every binary relation definable in Γ is also contained in Γ. The* completion *of Γ, written as Γ^c, is the set of all binary relations contained in Γ^+.*

The following lemma asserts that a complete set of binary relations is closed under the operations that are used to achieve PC.

Lemma 3.1. *[26] Let Γ be a complete set of binary relations over a domain D.*

Algorithm 3.1: DPC

Input : A binary constraint network $\mathcal{N} = \langle V, \mathcal{D}, C \rangle$; an ordering $\prec = (v_1, \ldots, v_n)$ on V.

Output: An equivalent subnetwork that is strongly DPC relative to $\prec$, or "Inconsistent".

1 **for** $k \leftarrow n$ **to** 1 **do**
2   **for** $i < k$ with $R_{ik} \in C$ **do**
3     $D_i \leftarrow D_i \cap R_{ki}(D_k)$;
4   **if** $D_i = \varnothing$ **then**
5   **return** *"Inconsistent"*
6   **for** $i, j < k$ *s.t.* $i \neq j$ and $R_{ik}, R_{jk} \in C$ **do**
7   **if** $R_{ij} \notin C$ **then**
8       $R_{ij} \leftarrow D_i \times D_j$;
9       $C \leftarrow C \cup \{R_{ij}\}$;
10     $R_{ij} \leftarrow R_{ij} \cap (R_{ik} \circ R_{kj})$;
11   **if** $R_{ij} = \varnothing$ **then**
12   **return** *"Inconsistent"*;
13 **return** $\mathcal{N}$.

Suppose R, S are binary relations in Γ. Then $R \cap S$ and $R \circ S$ are also in Γ.

Let Γ be a set of binary relations. A BCN $\mathcal{N} = \langle V, \mathcal{D}, C \rangle$ is *defined over* (or, simply, *over*) Γ if $R \in \Gamma$ for every constraint (s, R) in C. The constraint satisfaction problem (CSP) of Γ, denoted by $\mathbf{CSP}(\Gamma)$, is the problem of deciding the consistency of BCNs defined over Γ. Note that $\mathbf{CSP}(\Gamma^+)$ is log-space reducible to $\mathbf{CSP}(\Gamma)$ [26].

A set of binary relations Γ is *weakly closed under singletons*, if $\{\langle a, b \rangle\} \in \Gamma^+$ for any $R \in \Gamma$ and any $\langle a, b \rangle \in R$.

In this chapter, we often assume that the constraint languages are *complete* and *weakly closed under singletons*. We will see that this is not very restrictive as, for any set Γ of binary relations that is closed under a majority operation φ, the completion Γ^c of Γ is also closed under φ [61] and weakly closed under singletons (cf. Proposition 3.4).

3.4 The Strong Directional PC Algorithm

This section recalls the notions of *directional arc-consistency* (DAC) and *directional path-consistency* (DPC), and the *strong DPC* enforcing algorithm of Dechter and Pearl [37].

Definition 3.2. *[37] Let $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$ be a BCN and $\prec = (v_1, \ldots, v_n)$ an ordering of the variables in V. We say that $\mathcal{N}$ is* directionally arc-consistent (DAC) *relative to $\prec$ if v_i is arc-consistent relative to v_k for all $k > i$ with $R_{ik} \in C$. Similarly, $\mathcal{N}$ is* directionally path-consistent (DPC) *relative to $\prec$ if, for any $i \neq j$, (v_i, v_j) is path-consistent relative to v_k for all $k > i, j$ whenever $R_{ik}, R_{jk} \in C$. Meanwhile, $\mathcal{N}$ is* strongly DPC *relative to $\prec$ if it is both DAC and DPC relative to $\prec$.*

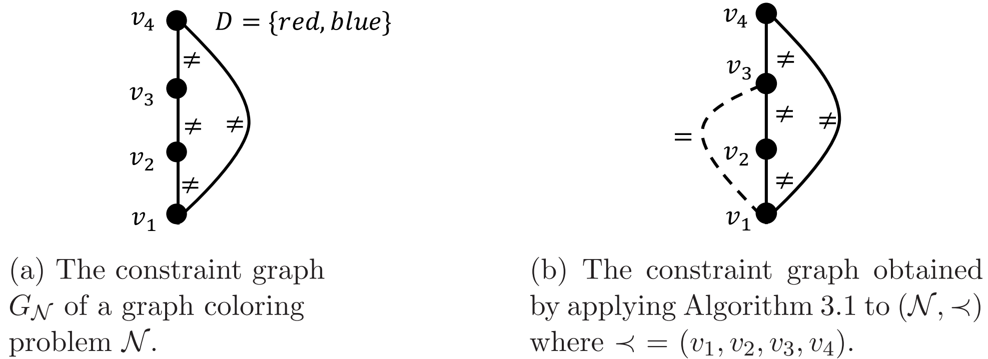
The strong DPC algorithm is presented as Algorithm 3.1. In comparison with traditional PC algorithms [24], a novelty of this single pass algorithm is its explicit reference to the constraint graph of the input constraint network. As only Line 8 may require extra working space, Algorithm 3.1 has a very low space complexity in practice. Further, Algorithm 3.1 runs in $O(w^*(\prec) \cdot e \cdot (\alpha + \beta))$ time [37], where e is the number of edges of the output constraint graph, $w^*(\prec)$ is the *induced width* [37] of the ordered graph along $\prec$, and α, β are the runtimes of relational intersection and composition respectively. Note that $w^*(\prec) \leq n$ and α, β are bounded by $O(d^2)$ and $O(d^3)$, respectively, where d is the largest domain size.

Proposition 3.1. *[37] Let $(\mathcal{N}, \prec)$ be an input to Algorithm 3.1, where $\prec = (v_1, \ldots, v_n)$. Suppose $\mathcal{N}' = \langle V, \mathcal{D}', C'\rangle$ is the output. Then*

(i) $G_{\mathcal{N}'}$ is triangulated and $\prec^{-1}$, the inverse of $\prec$, is a PEO of $G_{\mathcal{N}'}$;

(ii) $\mathcal{N}'$ is equivalent to $\mathcal{N}$ and strongly DPC relative to $\prec$.

Let Γ be a set of binary relations. We say that Algorithm 3.1 *decides* $\mathbf{CSP}(\Gamma)$ if,



(a) The constraint graph $G_{\mathcal{N}}$ of a graph coloring problem $\mathcal{N}$.

(b) The constraint graph obtained by applying Algorithm 3.1 to $(\mathcal{N}, \prec)$ where $\prec = (v_1, v_2, v_3, v_4)$.

Figure 3.1 : A graph coloring problem with domain $D_i = \{red, blue\}$ for $i = 1, 2, 3, 4$ [37].

for any given BCN $\mathcal{N}$ in **CSP**(Γ) and any ordering $\prec$ of variables of $\mathcal{N}$, Algorithm 3.1 returns "Inconsistent" iff $\mathcal{N}$ is inconsistent.

The following corollary follows directly from Proposition 3.1.

Corollary 3.1. *Let Γ be a complete set of binary relations. Then the following two conditions are equivalent:*

(i) Algorithm 3.1 decides **CSP**(Γ).

(ii) Let $\mathcal{N}$ be any not trivially inconsistent BCN in **CSP**(Γ). *Suppose $\mathcal{N}$'s constraint graph $G_{\mathcal{N}}$ is triangulated and let $\prec^{-1} = (v_n, \ldots, v_1)$ be a PEO of it. Then $\mathcal{N}$ is consistent if $\mathcal{N}$ is strongly DPC relative to $\prec$.*

Example 3.1. *The graph coloring problem $\mathcal{N}$ with domains $\{red, blue\}$ depicted in Figure 3.1 is taken from [37] and can be decided by Algorithm 3.1. After applying Algorithm 3.1 to $(\mathcal{N}, \prec)$, where $\prec = (v_1, v_2, v_3, v_4)$, a solution can be obtained along $\prec$ in a backtrack-free manner (see Figure 3.1b).*

3.5 Directional PC and Variable Elimination

This section characterizes the binary constraint languages Γ such that **CSP**(Γ) can be decided by **DPC**. We observe that **DPC** achieves (strong) DPC using the

idea of *variable elimination* [37]: it iterates variables along the ordering $\prec^{-1}$, and propagates the constraints of a variable v_k to subsequent variables in the ordering with the update rule $R_{ij} \leftarrow R_{ij} \cap (R_{ik} \circ R_{kj})$, as if v_k is '*eliminated*'.

The following definition formalizes the process of elimination.

Definition 3.3. *Let $\mathcal{N} = \langle V, \mathcal{D}, C \rangle$ be a BCN with $V = \{v_1, ..., v_n\}$ and $\mathcal{D} = \{D_1, ..., D_n\}$. For a variable v_x in V, let $E_x = \{R_{ix} \mid R_{ix} \in C\}$. The network obtained after v_x is eliminated from $\mathcal{N}$, written as*

$$\mathcal{N}_{-x} = \langle V \setminus \{v_x\}, \{D'_1, ..., D'_{x-1}, D'_{x+1}, ..., D'_n\}, C' \rangle,$$

is defined as follows:

- *If $E_x = \{R_{ix}\}$, we set $C' = C \setminus E_x$ and let*

$$D'_j = \begin{cases} D_i \cap R_{xi}(D_x), & \text{if } j = i \\ D_j, & \text{otherwise} \end{cases} \tag{3.1}$$

- *If $|E_x| \neq 1$, we set $D'_j = D_j$ for all $j \neq x$, and let*

$$C' = (C \setminus E_x) \cup \{(R_{ix} \circ R_{xj}) \cap R_{ij} \mid R_{jx}, R_{ix} \in E_x, i \neq j\}.$$

R_{ij} is assumed to be $D_i \times D_j$ if $R_{ij} \notin C$.

Figure 3.2 illustrates the elimination process.

Definition 3.4. *A BCN $\mathcal{N} = \langle V, \mathcal{D}, C \rangle$ is said to have the* variable elimination property *(VEP), if, for any v_x in V, every solution of $\mathcal{N}_{-x}$ can be extended to a solution of $\mathcal{N}$.*

$\mathcal{N}$ is said to have weak *VEP, if, for any v_x in V such that v_x is AC relative to all relations in E_x, every solution of $\mathcal{N}_{-x}$ can be extended to a solution of $\mathcal{N}$.*

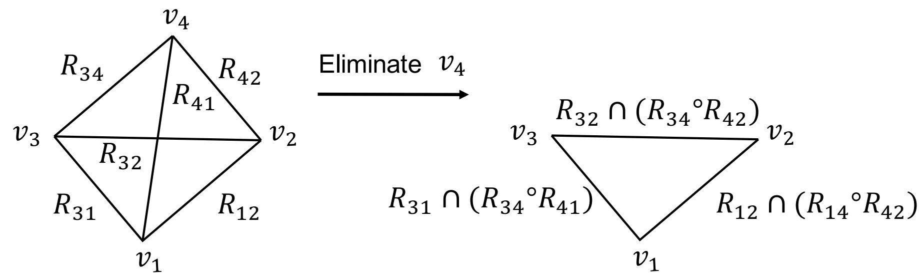


Figure 3.2 : Two binary constraint networks $\mathcal{N}$ and $\mathcal{N}_{-4}$.

A set of binary relations Γ *is said to have (weak) VEP if every BCN in* $\mathbf{CSP}(\Gamma)$ *has (weak) VEP. Such a set of binary relations* Γ *is also called a (weak) VEP class.*

It is easy to see that, if a BCN (a set of binary relations) has VEP, then it also has weak VEP. The following example explains why we should take special care when eliminating variables with only one successor in Eq. (3.1).

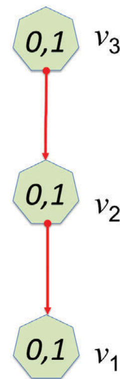


Figure 3.3 : A constraint graph that is a chain.

Example 3.2. *Let* $\mathcal{N} = \langle V, \mathcal{D}, C \rangle$ *be a BCN defined by* $V = \{v_1, v_2, v_3\}$, $D_1 = D_2 = D_3 = \{0, 1\}$, *and* $C = \{((v_3, v_2), R), ((v_2, v_1), R)\}$ *with* $R = \{(1, 0)\}$ *(see Figure 3.3). Suppose we do not have the operation specified in* (3.1) *and* $\prec = (v_3, v_2, v_1)$ *is the variable elimination ordering. Let* $\mathcal{N}_{-3}$ *be the restriction of* $\mathcal{N}$ *to* $\{v_1, v_2\}$. *Then* $\mathcal{N}_{-3}$ *has a unique solution* σ *but it cannot be extended to a solution of* $\mathcal{N}$.

Proposition 3.2. *Let* Γ *be a complete set of binary relations that is weakly closed under singletons. Then* **DPC** *decides* $\mathbf{CSP}(\Gamma)$ *iff* Γ *has VEP.*

Proof. We address the 'if' part first. Assume that Γ has VEP, and let $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$ be a network in $\mathbf{CSP}(\Gamma)$ that is not trivially inconsistent and strongly DPC relative to $\prec = (v_1, \ldots, v_n)$, where $G_\mathcal{N}$ is triangulated and $\prec^{-1}$ is a PEO of it. We show that $\mathcal{N}$ is consistent. Let $V_i = \{v_1, \ldots, v_i\}$ and $\mathcal{N}|_{V_i}$ be the restriction of $\mathcal{N}$ to V_i. We claim that $\mathcal{N}|_{V_i}$ is consistent for $k = 1, \ldots, n$ and prove the claim by induction on k. First, since $\mathcal{N}$ is not trivially inconsistent, D_1 is not empty and there is an $a_1 \in D_1$. Then, $\mathcal{N}|_{V_1}$ is consistent and has a solution $\sigma_1 = \langle a_1\rangle$. Further, suppose that $\mathcal{N}|_{V_i}$ is consistent and $\sigma_i = \langle a_1, a_2, ..., a_i\rangle$ is a solution of $\mathcal{N}|_{V_i}$. We show that σ_i can be extended to a solution $\sigma_{i+1} = \langle a_1, \ldots, a_i, a_{i+1}\rangle$ of $\mathcal{N}|_{V_{i+1}}$. Since Γ has VEP and $\mathcal{N}|_{V_i}$ is indeed the same network as the one obtained by eliminating v_{i+1} from $\mathcal{N}|_{V_{i+1}}$, by Definition 3.4, σ_i can be extended to a solution σ_{i+1} of $\mathcal{N}|_{V_{i+1}}$. Thus, by induction, $\mathcal{N}$ is consistent. By Corollary 3.1, **DPC** decides $\mathbf{CSP}(\Gamma)$.

Next, we address the 'only if' part. Assume that **DPC** decides $\mathbf{CSP}(\Gamma)$. We show that Γ has VEP. Let $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$ be a not trivially inconsistent network in $\mathbf{CSP}(\Gamma)$. Given $v_x \in V$, we show that every solution of $\mathcal{N}_{-x}$ can be extended to $\mathcal{N}$. Without loss of generality, we assume that $x = n$. Let $\sigma = \langle a_1, \ldots, a_{n-1}\rangle$ be a solution of $\mathcal{N}_{-n}$, and $E_n = \{R_{in} \mid R_{in} \in C\}$. By the definition of $\mathcal{N}_{-n}$, for any $R_{in}, R_{jn} \in E_n (i \neq j)$, we have $\langle a_i, a_j\rangle \in (R_{in} \circ R_{nj}) \cap R_{ij}$. We then construct a new problem $\mathcal{N}' = \langle V, \mathcal{D}', C'\rangle$ in $\mathbf{CSP}(\Gamma)$, where $\mathcal{D}' = \{D'_1, ..., D'_{n-1}, D_n\}$ with $D'_i = \{a_i\}$ for $1 \leq i < n$ and $C' = \{\{\langle a_i, a_j\rangle\} \mid 1 \leq i \neq j < n\} \cup E_n$. Clearly, σ is also a solution of $\mathcal{N}'_{-n}$ and $\mathcal{N}'_{-n}$ is strongly PC and, hence, strongly DPC relative to the ordering $(v_1, \ldots, v_{n-1})$. Further, since $\langle a_i, a_j\rangle \in (R_{in} \circ R_{nj}) \cap R_{ij}$ for any $R_{in}, R_{jn} \in E_n (i \neq j)$, we have that $\mathcal{N}'$ is strong DPC relative to $\prec = (v_1, \ldots, v_n)$. As $G_{\mathcal{N}'_{-n}}$ is a complete graph, $G_{\mathcal{N}'}$ is triangulated with $\prec^{-1}$ being a PEO of it. As

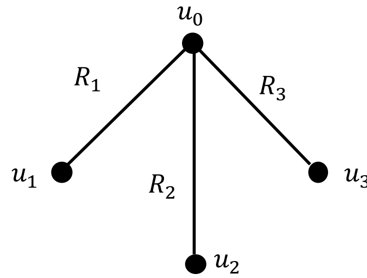


Figure 3.4 : An illustration of Example 3.3.

DPC decides **CSP**(Γ) and $\mathcal{N}' \in$ **CSP**(Γ), by Corollary 3.1, $\mathcal{N}'$ is consistent and has a solution that extends σ and is also a solution of $\mathcal{N}$. This shows that $\mathcal{N}$ is consistent and, hence, Γ has VEP. □

Therefore, if $\mathcal{N} = \langle V, \mathcal{D}, C \rangle$ is defined over a complete VEP class, then **DPC** can decide it. Note that in the above proposition we require Γ to be complete. This is important; for example, every *row-convex constraint* [7] network has VEP (cf. the proof of [135, Theorem 1]) and, hence, the class of row-convex constraints has VEP. However, **DPC** does not decide the consistency problem over the row-convex constraint class because it was shown to be NP-complete (cf. e.g. [63]).

VEP is closely related to the *Helly property*. For example, any set of intervals on the real line satisfies the Helly property, because the intersection of all intervals is not empty if every pair of intervals intersects. We give the formal definition as follows.

Definition 3.5. *A set* Γ *of binary relations over* $\mathcal{D} = \{D_1, ..., D_n\}$ *is said to have the* Helly property *if for any* $k > 2$ *binary relations,* $R_i \subseteq D_{u_i} \times D_{u_0} (1 \leq i \leq k, 1 \leq u_i \neq u_0 \leq n)$*, in* Γ*, and for any* k *values,* $a_i \in D_{u_i} (1 \leq i \leq k)$*, such that* $R_i(a_i) = \{b \in D_{u_0} \mid \langle a_i, b \rangle \in R_i\}$ *is nonempty, we have* $\bigcap_{i=1}^{k} R_i(a_i) \neq \varnothing$ *iff* $R_i(a_i) \cap R_j(a_j) \neq \varnothing$ *for any* $1 \leq i \neq j \leq k$*.*

Example 3.3. *Let* $D_{u_0} = \{a, b, c, d\}, D_{u_1} = \{e\}, D_{u_2} = \{f\}, D_{u_3} = \{g\}$ *and* $R_1 = \{\langle e, a\rangle, \langle e, b\rangle, \langle e, c\rangle\}, R_2 = \{\langle f, b\rangle, \langle f, c\rangle, \langle f, d\rangle\}, R_3 = \{\langle g, c\rangle, \langle g, d\rangle, \langle g, a\rangle\}$. *See Figure 3.4 for an illustration. Then* $\Gamma = \{R_1, R_2, R_3\}$ *over* $\mathcal{D} = \{D_{u_0}, D_{u_1}, D_{u_2}, D_{u_3}\}$ *has the Helly property.*

Theorem 3.1. *A set of binary relations* Γ *over* $\mathcal{D} = \{D_1, ..., D_n\}$ *has VEP iff it has the Helly property.*

Proof. Suppose Γ has VEP. We show that Γ has the Helly property. Let $\mathcal{D} = \{D_1, ..., D_n\}$ be the set of domains related to relations in Γ. Suppose $R_i \subseteq D_{u_i} \times D_{u_0}$ $(1 \leq i \leq k,\ 1 \leq u_i \neq u_0 \leq n)$ are $k > 2$ binary relations in Γ and $a_i \in D_{u_i}$ $(1 \leq i \leq k)$ are values such that $\varnothing \neq R_i(a_i) \subseteq D_{u_0}$. Suppose $R_i(a_i) \cap R_j(a_j)$ is nonempty for any i, j with $1 \leq i \neq j \leq k$. We show that $\bigcap_{i=1}^{k} R_i(a_i)$ is nonempty. To this end, we construct a BCN $\mathcal{N} = \langle V, \mathcal{D}', C\rangle$ over Γ with $V = \{v_1, ..., v_k, v_{k+1}\}$, $\mathcal{D}' = \{D_{u_1}, ..., D_{u_k}, D_{u_0}\}$, and $C = \{R_{i,k+1} \mid 1 \leq i \leq k\}$, where $R_{i,k+1} = R_i$. Consider $\mathcal{N}_{-(k+1)}$. As $R_i(a_i) \cap R_j(a_j) \neq \varnothing$, we have $\langle a_i, a_j\rangle \in R_{i,k+1} \circ R_{k+1,j}$. This shows that $\sigma = \langle a_1, \ldots, a_k\rangle$ is a solution of $\mathcal{N}_{-(k+1)}$. Since Γ and, hence, $\mathcal{N}$ have VEP, $\mathcal{N}$ has a solution that extends σ. Hence there exists $a \in D_{u_0}$ such that $a \in R_{i,k+1}(a_i)$ for every $1 \leq i \leq k$. Thus $\bigcap_{i=1}^{k} R_i(a_i) \neq \varnothing$. This proves that Γ has the Helly property.

Suppose Γ over $\mathcal{D} = \{D_1, ..., D_n\}$ has the Helly property. We show that Γ has VEP. Let $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$ be a not trivially inconsistent BCN defined over Γ with $V = \{v_1, v_2, ..., v_n\}$ and C is a set of binary constraints $((v_i, v_j), R)$ with $R \in \Gamma$. Let $E_n = \{R_{in} \mid R_{in} \in C\}$. Assume $\sigma = \langle a_1, a_2, ..., a_{n-1}\rangle$ is a solution of, say, $\mathcal{N}_{-n}$. We show that there exists $a_n \in D_n$ such that $\langle a_1, ..., a_{n-1}, a_n\rangle$ is a solution of $\mathcal{N}$. If E_n is empty, we can take any a_n from D_n which is nonempty since $\mathcal{N}$ is not trivially inconsistent; if E_n contains only one constraint, say, $((v_i, v_n), R_{in})$, by $a_i \in D'_i = D_i \cap R_{ni}(D_n)$, there exists $a_n \in D_n$ such that $\langle a_i, a_n\rangle \in R_{in}$; if

E_n contains $k \geq 2$ constraints and let them be $((v_{u_i}, v_n), R_{u_i n})$ $(1 \leq i \leq k)$, we have $\langle a_{u_i}, a_{u_j} \rangle \in R_{u_i u_j} \cap (R_{u_i n} \circ R_{n u_j})$ for $1 \leq i \neq j \leq k$. Therefore, we have $R_{u_i n}(a_i) \cap R_{u_j n}(a_j) \neq \varnothing$ for $1 \leq i \neq j \leq k$. By the Helly property of Γ, we have $\bigcap_{i=1}^k R_{u_i n}(a_i) \neq \varnothing$. So we can take any $a_n \in \bigcap_{i=1}^k R_{u_i n}(a_i)$ so that $\langle a_1, \ldots, a_{n-1}, a_n \rangle$ is a solution of $\mathcal{N}$. Therefore, Γ has VEP. □

The class of row-convex constraints and the class of tree-convex constraints have the Helly property and, thus, they have VEP by Theorem 3.1.

Proposition 3.2 only concerns a complete set of binary relations that has VEP. The following proposition, viz., Proposition 3.3, does not require a set of binary relations to be complete. However, we also note that path-consistency operations can destroy the Helly property; this suggests that Proposition 3.3 is only useful when N happens to have the Helly property after enforcing DPC.

Proposition 3.3. *Suppose Γ is a set of binary relations that has the Helly property. Let $\mathcal{N} \in \mathbf{CSP}(\Gamma)$. Suppose $\mathcal{N}$ is not trivially inconsistent and $G_{\mathcal{N}}$ is triangulated with $\prec^{-1} = (v_n, \ldots, v_1)$ as a PEO of it. Then $\mathcal{N}$ is consistent if it is strongly DPC relative to $\prec$.*

Proof. Let $\mathcal{N} \in \mathbf{CSP}(\Gamma)$. Suppose $\mathcal{N} = \langle V, \mathcal{D}, C \rangle$ is not trivially inconsistent and $G_{\mathcal{N}}$ is triangulated with $\prec^{-1} = (v_n, \ldots, v_1)$ being a PEO of it. Suppose $\mathcal{N}$ is strongly DPC relative to $\prec$. We show that $\mathcal{N}$ is consistent. Let $V_k = \{v_1, \ldots, v_k\}$ and $\mathcal{N}_k$ be the restriction of $\mathcal{N}$ to V_k. Since $\mathcal{N}$ is not trivially inconsistent, we have that $\mathcal{N}_1$ is consistent. Suppose $\mathcal{N}_k$ is consistent, we show that $\mathcal{N}_{k+1}$ is consistent. Let $\sigma = \langle a_1, \ldots, a_k \rangle$ be a solution of $\mathcal{N}_k$. Let $E_{k+1} = \{R_{i,k+1} \mid R_{i,k+1} \in C, i \leq k\}$. Since $G_{\mathcal{N}}$ is triangulated and $\prec^{-1} = (v_n, \ldots, v_1)$ is a PEO of it, for any two different constraints $R_{i,k+1}, R_{j,k+1} \in E_{k+1}$, we have $R_{ij} \in C$. Further, since $\mathcal{N}$ is strongly DPC relative to $\prec$, we have $\langle a_i, a_j \rangle \in (R_{i,k+1} \circ R_{k+1,j}) \cap R_{ij}$. Thus, we have $R_{i,k+1}(a_i) \cap R_{j,k+1}(a_j) \neq \varnothing$ for any two different constraints $R_{i,k+1}, R_{j,k+1} \in E_{k+1}$.

Since Γ has the Helly property, we have $\bigcap_{R_{i,k+1} \in E_{k+1}} R_{i,k+1}(a_i) \neq \varnothing$. Therefore, σ can be extended to a solution of $\mathcal{N}_{k+1}$ and $\mathcal{N}_{k+1}$ is consistent. By induction on k, we have that $\mathcal{N}$ is consistent. □

3.6 Weak VEP Classes and Majority-Closed Classes

In this section we characterize weak VEP classes. We show that a complete set of binary relations Γ has weak VEP iff all relations in Γ are closed under a majority operation.

Proposition 3.4. *Let Γ be the set of binary relations that is closed under a multi-sorted majority operation $\varphi = \{\varphi_1, ..., \varphi_n\}$ on $\mathcal{D} = \{D_1, ..., D_n\}$. Then Γ is weakly closed under singletons.*

Proof. Suppose R is a relation in Γ and $\langle a, b\rangle \in R \subseteq D_i \times D_j$. We show that $\{\langle a, b\rangle\}$ is closed under φ. For any $t_1, t_2, t_3 \in \{\langle a, b\rangle\}$, we have $t_1 = t_2 = t_3 = \langle a, b\rangle$, and, hence, $\varphi(t_1, t_2, t_3) = \langle \varphi_i(a, a, a), \varphi_j(b, b, b)\rangle = \langle a, b\rangle$. This shows that $\{\langle a, b\rangle\}$ is closed under φ and, hence, a relation in Γ. □

Next, we show that complete weak VEP classes are majority-closed classes.

Theorem 3.2. *Let Γ be a complete set of binary relations over a set of finite domains $\mathcal{D} = \{D_1, ..., D_n\}$. Then Γ has weak VEP iff it is a majority-closed class.*

Proof. We first deal with the 'only if' part. Suppose that Γ is a complete set of binary relations that has weak VEP. By Theorem 1.3, we only need to show that for every BCN in $\mathbf{CSP}(\Gamma)$, establishing strong PC ensures global consistency. Let $\mathcal{N}^0$ be a network in $\mathbf{CSP}(\Gamma)$ and suppose $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$ is the network obtained by enforcing strong PC on $\mathcal{N}^0$. Since Γ is complete and hence closed under operations for achieving PC by Lemma 3.1, $\mathcal{N}$ is also a problem in $\mathbf{CSP}(\Gamma)$. Suppose $\mathcal{N}$ is not

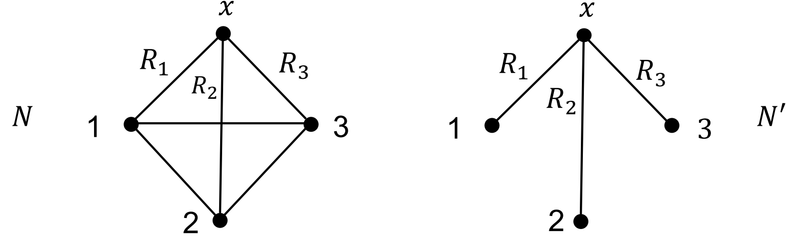


Figure 3.5 : Illustration of proof of Theorem 3.2.

trivially inconsistent. We show that any partial solution of $\mathcal{N}$ can be extended to a solution of $\mathcal{N}$.

Suppose $V' = \{v_1, \ldots, v_{m-1}\} \subset V$ and $\sigma = \langle a_1, \ldots, a_{m-1}\rangle$ is a solution of $\mathcal{N}|_{V'}$, which is the restriction of $\mathcal{N}$ to V'. Assume further that $v_m \notin V'$ is a new variable and let $V'' = V' \cup \{v_m\}$. We show that σ can be consistently extended to $\mathcal{N}|_{V''}$, the restriction of $\mathcal{N}$ to V''. Because $\mathcal{N}$ is strongly PC, $\mathcal{N}|_{V''}$ is strongly PC as well. In particular, v_i is AC relative to v_m for any R_{im} in C, and R_{ij} is PC relative to v_m (*i.e.*, $R_{ij} \subseteq R_{im} \circ R_{mj}$) for any $i \neq j$ such that both R_{im} and R_{jm} are in C. By Definition 3.3, $\mathcal{N}|_{V'}$ is the same as $(\mathcal{N}|_{V''})_{-m}$, viz., the network obtained by eliminating v_m from $\mathcal{N}|_{V''}$. Moreover, since $\mathcal{N}$ and, hence, $\mathcal{N}|_{V''}$ are AC, v_m is AC relative to all constraints R_{im} that are in C. By the assumption that Γ has weak VEP, σ can be consistently extended to v_m. Following this reasoning, we will find a solution of $\mathcal{N}$ that extends σ.

Next, we consider the 'if' part. Suppose that Γ is a complete set of binary relations that is closed under some multi-sorted majority operation $\varphi = \{\varphi_1, \ldots, \varphi_n\}$ on $\mathcal{D}$. Let $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$ be a problem in $\mathbf{CSP}(\Gamma)$ and v_x a variable in V. Let $E_x = \{R_{ix} \mid R_{ix} \in C\}$, and $\mathcal{N}_{-x} = \langle V \setminus \{v_x\}, \mathcal{D}, C'\rangle$, where $C' = (C \cup \{(R_{ix} \circ R_{xj}) \cap R_{ij} \mid R_{jx}, R_{ix} \in E_x\}) \setminus E_x$. Suppose that v_x is AC relative to all relations in E_x. We only need to show that any solution of $\mathcal{N}_{-x}$ can be extended to a solution of $\mathcal{N}$. We prove this by contradiction.

Let σ be a solution of $\mathcal{N}_{-x}$. Assume that σ cannot be extended to a solution of $\mathcal{N}$. Therefore, E_x cannot be empty, otherwise σ can be trivially extended to a solution of $\mathcal{N}$. The case where $E_x = \{R_{ix}\}$ is a singleton is also impossible, as by (3.1), v_i is AC relative to R_{ix} and we could extend σ to a solution of $\mathcal{N}$ by assigning any valid value to v_x. Suppose that E_x has $q \geq 2$ constraints and let them be $((v_1, v_x), R_1)$, ..., $((v_q, v_x), R_q)$. We define a new problem $\mathcal{N}' = \langle V, \mathcal{D}, E_x \rangle$ as illustrated in Figure 3.5. Since v_x is AC relative to all relations in E_x, it is easy to verify that $\mathcal{N}'$ has a solution. For example, one can construct a solution of $\mathcal{N}'$ by simply picking a value from D_x for v_x and then extending that valuation to $v_1, \ldots, v_q$. Now, we construct a q-ary relation $R = \{\langle \gamma(v_1), \ldots, \gamma(v_q) \rangle \mid \gamma \text{ is a solution of } \mathcal{N}'\}$. The solution set S of $\mathcal{N}'$ can be obtained by using a sequence of the Cartesian product, equality selection, and projection operations [61]. Therefore, $S \in \Gamma^+$. Since $R = \pi_{v_1,\ldots,v_q}(S)$, we have $R \in \Gamma^+$. By Theorem 1.3, R should be 2-decomposable; however, in the sequel we show that it is not, which is a contradiction.

Let $t = \langle \sigma(v_1), \ldots, \sigma(v_q) \rangle$, where σ is a solution of $\mathcal{N}_{-x}$. It is clear that t is a solution of $\mathcal{N}'|_{\{v_1,\ldots,v_q\}}$. For any list of indices I chosen from $\{1, \ldots, q\}$, with $|I| \leq 2$, we claim that $\pi_I(t) \in \pi_I(R)$. We recall that, for any two relations $R_{ix}, R_{jx} \in E_x$, the relation between v_i and v_j in $\mathcal{N}_{-x}$ is $R_{ij} \cap (R_{ix} \circ R_{xj})$. Therefore, any partial solution $\langle \sigma(v_i), \sigma(v_j) \rangle (1 \leq i, j \leq q)$ of $\mathcal{N}'$ can be consistently extended to v_x and, by the construction of $\mathcal{N}'$, further consistently extended to a solution of $\mathcal{N}'$. Thus, $\pi_I(t)$ is in $\pi_I(R)$ for any list of indices I chosen from $\{1, \ldots, q\}$, with $|I| \leq 2$. However, $t \notin R$ because σ cannot be extended to a solution of $\mathcal{N}'$, which implies that R is not 2-decomposable. $\square$

3.7 The Variable Elimination Algorithm DPC*

This section presents a variant of **DPC** for solving BCNs defined over any weak VEP class. The new algorithm, called **DPC*** and presented as Algorithm 3.2, can

Algorithm 3.2: DPC*

Input : A binary constraint network $\mathcal{N} = \langle V, \mathcal{D}, C \rangle$; an ordering $\prec = (v_1, \ldots, v_n)$ on V.

Output: An equivalent subnetwork that is decomposable relative to $\prec$ (see Definition 3.6), or "Inconsistent".

```
1  for k ← n to 1 do
2      if k has only one successor and that successor is i then
3          D_i ← D_i ∩ R_ki(D_k)
4          if D_i = ∅ then
5              return "Inconsistent"
6      else
7          for i < k with R_ik ∈ C do
8              D_k ← D_k ∩ R_ik(D_i)
9              if D_k = ∅ then
10                 return "Inconsistent"
11         for i < k with R_ik ∈ C do
12             for j < i with R_jk ∈ C do
13                 if R_ij ∉ C then
14                     R_ij ← D_i × D_j
15                     C ← C ∪ {R_ij}
16                 R_ij ← R_ij ∩ (R_ik ∘ R_kj);
17                 if R_ij = ∅ then
18                     return "Inconsistent"
19 return N.
```

solve problems that are not solvable by **DPC** (cf. Example 3.4 and Proposition 3.6). Compared with the variable elimination algorithm for solving CRC constraints [135], **DPC*** enforces a weaker AC condition instead of full AC. We first justify the correctness of Algorithm 3.2.

Theorem 3.3. *Let Γ be a complete weak VEP class. Suppose $\mathcal{N}$ is a BCN defined over Γ and $\prec = (v_1, \ldots, v_n)$ any ordering of the variables of $\mathcal{N}$. Then, given $\mathcal{N}$ and $\prec$, Algorithm 3.2 does not return "Inconsistent" iff $\mathcal{N}$ is consistent.*

Proof. Suppose the input network $\mathcal{N}$ is consistent. Since **DPC*** only prunes off

certain infeasible domain values or relation tuples, the algorithm does not find any empty domains or relations in Lines 4, 9, and 17. Thus, it does not return "Inconsistent".

Suppose the algorithm does not return "Inconsistent" and let $\mathcal{N}' = \langle V, \mathcal{D}', C' \rangle$ be the output network, where $\mathcal{D}' = \{D'_1, ..., D'_n\}$. We show that $\mathcal{N}'$ is consistent.

Write $\mathcal{M}^{(0)}$ for $\mathcal{N}$ and write $\mathcal{M}^{(i)}$ for the result of the i-th loop in the call of **DPC*** on input $\mathcal{N}$ and $\prec = (v_1, v_2, ..., v_n)$. Then $\mathcal{N}' = \mathcal{M}^{(n-1)}$ and all $\mathcal{M}^{(i)}$ $(0 \leq i < n)$ are equivalent to $\mathcal{N}$. Let $\mathcal{Q}_i$ be the restriction of $\mathcal{M}^{(i)}$ to $\{v_1, v_2, ..., v_{n-i}\}$ $(0 \leq i < n)$. In essence, $\mathcal{Q}_i$ is obtained by eliminating v_{n-i+1} from $\mathcal{Q}_{i-1}$ (Lines 2-5 or Lines 11-18), while also enforcing AC (Lines 7-10) for v_{n-i+1} relative to all its successors if it has more than one successor. Since Γ is a complete weak VEP class, every BCN defined over Γ has weak VEP. In particular, each $\mathcal{Q}_i (0 \leq i < n)$ is defined over Γ and has weak VEP. This implies that every solution of $\mathcal{Q}_{i+1}$ can be extended to a solution of $\mathcal{Q}_1$. Since no inconsistency is detected in the process, we have $D'_1 \neq \varnothing$ and thus $\mathcal{Q}_{n-1}$ is consistent. By the above analysis, this implies that $\mathcal{Q}_{n-2}, ..., \mathcal{Q}_1, \mathcal{Q}_0 = \mathcal{M}^{(0)} = \mathcal{N}$ are all consistent. □

The preceding proof also gives a way to generate *all* solutions of a consistent input network *backtrack-free* by appropriately instantiating the variables along the input ordering $\prec$. Indeed, for all $1 \leq k < n$, a solution $\langle a_1, \ldots, a_k \rangle$ of $\mathcal{N}'_k$ can be extended to a solution $\langle a_1, \ldots, a_{k+1} \rangle$ of $\mathcal{N}'_{k+1}$ by choosing an element a_{k+1} from the intersection of all $R_{i,k+1}(a_i)$ with $i \leq k$ and $R_{i,k+1} \in C'$, which is always nonempty as shown in the preceding proof. As we know that if Γ is majority-closed, the completion of Γ is also majority-closed [61], and that complete majority-closed classes and complete weak VEP classes are equivalent by Theorem 3.2, this also proves the following result:

Definition 3.6. *A constraint network $\mathcal{N}$ is* decomposable *relative to a variable*

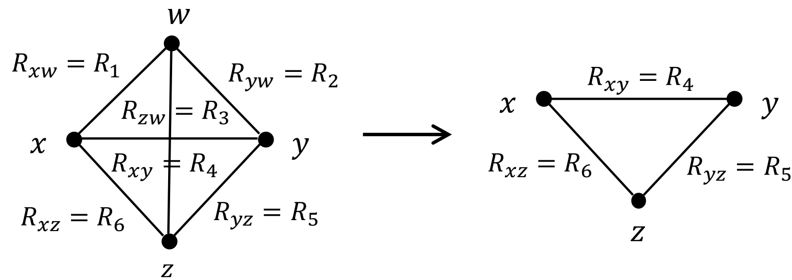


Figure 3.6 : A constraint network $\mathcal{N}$ and its elimination $\mathcal{N}_{-w}$.

ordering $\prec = (v_1, \ldots, v_n)$ *if any partial solution of* $\mathcal{N}$ *on* $\{v_1, ..., v_k\}$ *for any* $1 \leq k < n$ *can be extended to a solution of* $\mathcal{N}$.

Proposition 3.5. *Suppose* $\mathcal{N}$ *is a consistent BCN defined over a majority-closed class and* $\prec = (v_1, ..., v_n)$ *an ordering of variables of* $\mathcal{N}$. *Then, given* $\mathcal{N}$ *and* $\prec$, *Algorithm 3.2 returns an equivalent subnetwork* $\mathcal{N}'$ *that is decomposable relative to* $\prec$.

Note that Lines 2-10 in **DPC***do not achieve DAC of input networks. Therefore, **DPC*** *does not* achieve strong DPC. Since the overall runtime of Lines 2-10 is the same as enforcing DAC, this places **DPC*** in the same time complexity class as **DPC**, which is $O(w^*(\prec)ed^3)$, where $w^*(\prec)$ is the induced width of the ordered constraint graph along the input variable ordering $\prec$. The following example, however, gives a BCN that can be solved by **DPC*** but not by **DPC**, which shows that the loop in Lines 7-10 is necessary.

Example 3.4. *Let* $D = \{a, b, c\}$ *and* φ *be the majority operation on* D *such that for all* $i, j, k \in D$, $\varphi(i, j, k) = a$ *if* $i \neq j$, $j \neq k$, *and* $i \neq k$, *and* $\varphi(i, j, k) = r$ *otherwise, where* r *is the repeated value (e.g.,* $\varphi(b, c, b) = b$*). Let* $\Gamma = \{R_1, R_2, R_3, R_4, R_5, R_6\}$, *where* $R_1 = \{\langle a, a\rangle, \langle a, c\rangle\}$, $R_2 = \{\langle c, c\rangle, \langle c, b\rangle\}$, $R_3 = \{\langle b, b\rangle, \langle b, a\rangle\}$, $R_4 = \{\langle a, c\rangle\}$, $R_5 = \{\langle c, b\rangle\}$, *and* $R_6 = \{\langle a, b\rangle\}$. *Every* $R \in \Gamma$ *is closed under the majority operation* φ *on* D. *Now, consider the constraint network* $\mathcal{N} \in \mathbf{CSP}(\Gamma)$ *as presented in Figure 3.6. Since* $R_{xw} \circ R_{wz} = R_6$, $R_{xw} \circ R_{wy} = R_4$, *and* $R_{yw} \circ R_{wz} = R_5$, *the*

eliminated network $\mathcal{N}_{-w}$ *is the same as the restriction of* $\mathcal{N}$ *to the set of variables* $\{v_x, v_y, v_z\}$*. Let* $\sigma(v_x) = a, \sigma(v_y) = c, \sigma(v_z) = b$*. Then* σ *is a solution of* $\mathcal{N}_{-w}$*, but* σ *cannot be extended to a solution of* $\mathcal{N}$*. Thus,* $\mathcal{N}$ *and hence* Γ *do not have VEP. By Theorem 3.2,* **DPC** *does not decide* **CSP**(Γ)*.*

On the other hand, since Γ *is majority-closed, by Proposition 3.5,* **DPC*** *can correctly decide the consistency of* $\mathcal{N}$*. This observation is confirmed by calling the two algorithms on* $\mathcal{N}$*. Take the PEO* $\prec = (w, x, y, z)$ *as an example; the other PEOs are analogous. Let* $(\mathcal{N}, \prec)$ *be an input to* **DPC***. After processing* w*, we have* $D_x = \{a\}, D_y = \{c\}, D_z = \{b\}$ *and* $R_{xy} = \{\langle a, c\rangle\}$*,* $R_{xz} = \{\langle a, b\rangle\}$*,* $R_{zy} = \{b, c\}$*. We can observe that* $\langle x = a, y = c, z = b\rangle$ *is a solution to the eliminated subnetwork. Thus, if we keep running* **DPC***, it will not detect inconsistency. On the other hand, for* **DPC****, when eliminating* w*,* **DPC*** *makes* w *AC relative to its neighbors. Note that* **DPC** *does not perform this operation. After that,* D_w *is empty, and the algorithm will stop and output "Inconsistent".*

Proposition 3.6. *Let* $\varphi = \{\varphi_1, \ldots, \varphi_n\}$ *be a majority operation on* $\mathcal{D} = \{D_1, \ldots, D_n\}$*. If there exists a domain* D_i *in* $\mathcal{D}$ *that contains more than two elements, then the set* Γ_φ *of binary relations that are closed under* φ *has neither the Helly property nor VEP.*

Proof. Suppose a, b, c are three different values from D_i. It is easy to see that the relations $R_1 = \{\langle a, a\rangle, \langle a, b\rangle\}$, $R_2 = \{\langle a, b\rangle, \langle a, c\rangle\}$, and $R_3 = \{\langle a, a\rangle, \langle a, c\rangle\}$ are all closed under φ. Therefore, R_1, R_2, and R_3 are all in Γ_φ. Because any two of $R_1(a), R_2(a), R_3(a)$ have a common element but $R_1(a) \cap R_2(a) \cap R_3(a) = \varnothing$, this shows that Γ_φ does not have the Helly Property and, hence by Theorem 3.1, does not have VEP. □

This result shows that no complete VEP class could have a domain with 3 or more values. Therefore, there are no interesting complete constraint languages except

Algorithm	**DPC***	**PC2001**	**SAC3-SDS**
Time	$O(w^*(\prec)ed^3)$	$O(n^3d^3)$	$O(ned^3)$

Table 3.1 : Comparison of time complexities among state-of-the-art algorithms for solving majority-closed constraint networks.

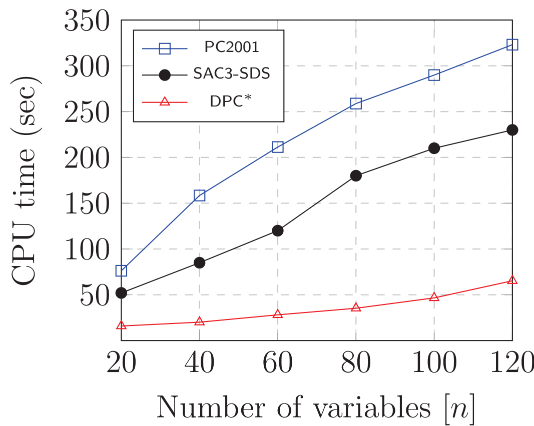
the boolean ones that can be decided by **DPC** (cf. Proposition 3.2), while all binary majority-closed classes (including CRC and tree-preserving constraints) can be decided by **DPC*** (cf. Proposition 3.5).

DPC* can also be used to solve majority-closed constraints of higher arities. This is because, by Theorem 1.3, every relation definable in a majority-closed language is 2-decomposable. Therefore, for each majority relation R of arity $m > 2$, if a constraint $c = ((y_1, ..., y_m), R)$ appears in a constraint network $\mathcal{N}$, we could replace c with a set of binary constraints $c_{ij} = ((y_i, y_j) \mid \pi_{ij}(R))$ $(1 \leq i < j \leq m)$, where $\pi_{ij}(R) = \{\langle t[y_i], t[y_j]\rangle \mid t \in R\}$.

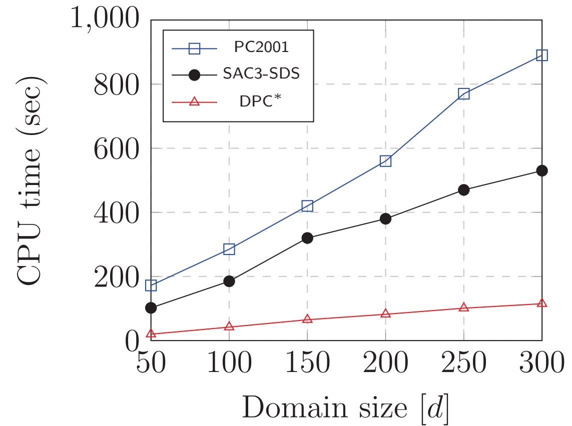
3.8 Evaluations

In this section we compare algorithm DPC* against the state-of-the-art algorithms for solving majority-closed constraint networks. These are **SAC3-SDS** [9] and **PC2001** [12]. **SAC3-SDS** is currently the best *singleton arc-consistency* (SAC) enforcing algorithm [35]. Enforcing either SAC or PC correctly decides the consistency of a majority-closed constraint network [61, 23]. Note that *Singleton linear arc-consistency* (SLAC) is an alternative consistency notion that can be enforced to solve majority-closed constraint networks [71], but, to the best of our knowledge, no practical SLAC algorithms have been made available so far. Comparison of time complexities among the three algorithms is presented in Table 3.1.

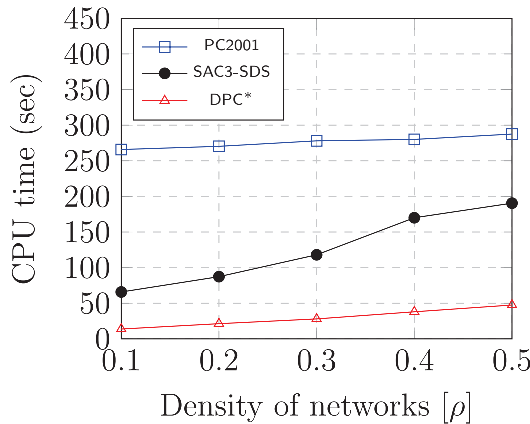
Two different sets of data are considered for experiments, which are described as follows:



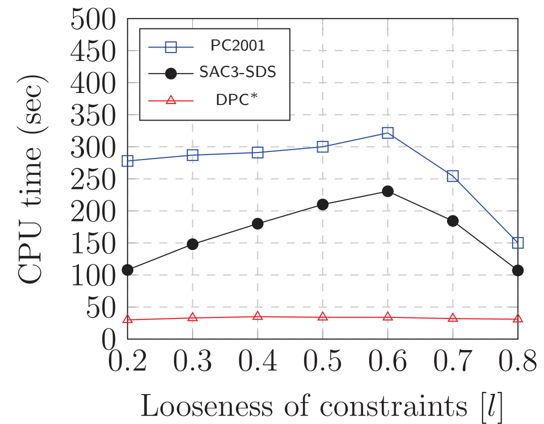
(a) Evaluations in the number n of variables. We set $\rho = 0.5$, $l = 0.3$, $d = 100$.



(b) Evaluations in the size d of domains. We set $\rho = 0.5$, $l = 0.3$, $n = 100$.



(c) Evaluations in the density ρ of networks. We set $l = 0.3$, $d = 100$, $n = 100$.



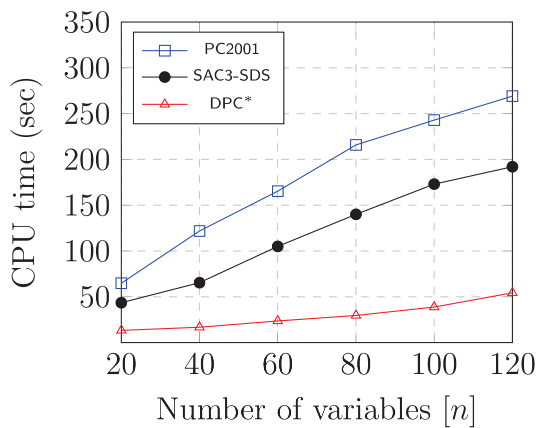
(d) Evaluations in the looseness l of constraints. We set $\rho = 0.5$, $d = 100$, $n = 100$.

Figure 3.7 : Performance comparisons among **DPC***, **SAC3-SDS** and **PC2001** for solving tree-preserving constraint networks.

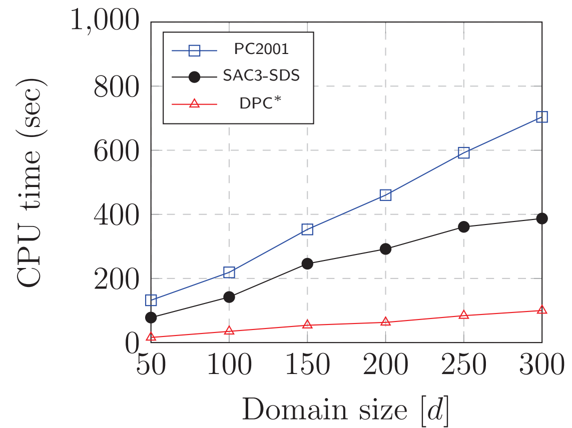
(1) Tree-preserving constraint networks. These networks are randomly generated using the random tree-preserving constraint generator detailed in [69].

(2) Random majority-closed constraint networks. These can be used to test the average performance of different algorithms. To generate such networks, we need to generate random majority-closed constraint languages as follows:

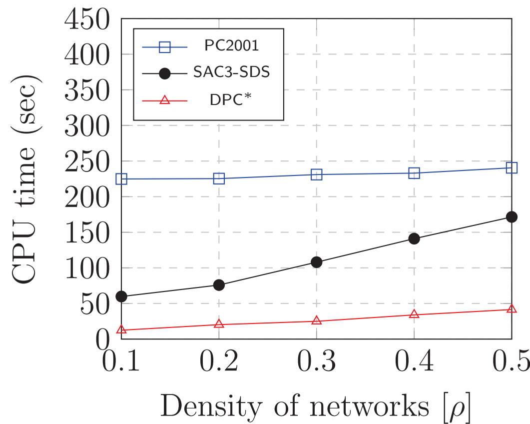
(a) Randomly define a majority operation $\otimes_i : D_i^3 \to D_i$ for each domain $D_i \in \mathcal{D}$



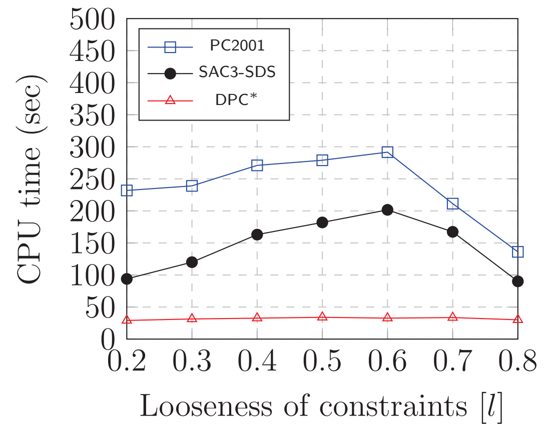
(a) Evaluations in the number n of variables. We set $\rho = 0.5$, $l = 0.3$, $d = 100$.



(b) Evaluations in the size d of domains. We set $\rho = 0.5$, $l = 0.3$, $n = 100$.



(c) Evaluations in the density ρ of networks. We set $l = 0.3$, $d = 100$, $n = 100$.



(d) Evaluations in the looseness l of constraints. We set $\rho = 0.5$, $d = 100$, $n = 100$.

Figure 3.8 : Performance comparisons among **DPC***, **SAC3-SDS** and **PC2001** for solving random majority-closed constraint networks.

as follows: for any $x, y, z \in D_i$,

$$\otimes_i(x, y, z) = \begin{cases} \text{any } v \in D_i, \text{if } x, y, z \text{ are all different,} \\ \text{any repeated value of } x, y, z, \text{otherwise.} \end{cases}$$

Note that v is chosen randomly for each triple $\langle x, y, z \rangle$, and it is a random choice in D_i.

(b) Randomly generate constraints $R_{ij} \subseteq D_i \times D_j$ and test whether

$$\{\langle \otimes_i(t_x[1], t_y[1], t_z[1]), \otimes_j(t_x[2], t_y[2], t_z[2])\rangle \mid t_x, t_y, t_z \in R_{ij}\} \subseteq R_{ij} \tag{3.2}$$

holds. By definition, R_{ij} is majority-closed under $(\otimes_i, \otimes_j)$ iff (3.2) holds.

We used the model in [13, 40] to generate random consistent constraint networks for experiments. These constraint networks were generated by varying four parameters: (1) the number of variables n, (2) the size of the domains d, (3) the density of the constraint networks ρ (i.e. the ratio of non-universal constraints to n^2) and (4) the looseness of constraints l (i.e. the ratio of the number of allowed tuples to d^2). We fix three of the four parameters and vary the remaining parameter. All algorithms were implemented taking equal care and using Python 3.6. Experiments were carried out on a computer with an Intel Core i5-4570 processor (3.2 GHz frequency per CPU core) and 4 GB of memory.

The graphs in Figure 3.7 and Figure 3.8 illustrate the experimental comparisons among algorithms **DPC***, **SAC3-SDS** and **PC2001** for solving tree-preserving and random majority-closed constraint networks respectively. The data points in each graph are CPU times averaged over 20 instances.

From Figure 3.7 and Figure 3.8, we observe that the corresponding results regarding tree-preserving and random majority-closed constraint networks are qualitatively similar. Therefore, our analysis only focuses on Figure 3.7 and the results are applicable to Figure 3.8 as well.

We observe in Figure 3.7a and Figure 3.7b that all algorithms approximately show linear time behaviors with respect to n and d. On the other hand, Figure 3.7c shows that **PC2001** is not sensitive to the density of networks whereas **DPC*** and **SAC3-SDS** perform better when the density of networks is lower. Figure 3.7d

shows that the CPU time for **DPC*** remains almost unchanged when increasing the looseness of constraints. However, the CPU times for **PC2001** and **SAC3-SDS** both go up and then drop down when increasing the looseness of constraints. Finally, we also observe in all the graphs in Figure 3.7 that the performance differences among **DPC***, **PC2001** , and **SAC3-SDS** are remarkable. In particular, **DPC*** is up to 7 times and 5 times faster than **PC2001** and **SAC3-SDS** respectively. This is mainly due to the fact that **DPC*** is a single pass algorithm over the ordered input constraint networks and, hence, very scalable compared to **PC2001** and **SAC3-SDS**.

3.9 Conclusion

This chapter investigated which constraint satisfaction problems can be efficiently decided by enforcing directional path-consistency. Given a complete binary constraint language Γ, it turns out that **DPC** can decide **CSP**(Γ) if Γ is defined over domains with less than three values. For a possibly incomplete binary constraint language Γ, we proved that Γ has the Helly property if, and only if, for any not trivially inconsistent and triangulated binary constraint network $\mathcal{N}$ over Γ, $\mathcal{N}$ is consistent if it is strongly DPC relative to the inverse ordering of some perfect elimination ordering of the constraint graph of $\mathcal{N}$. The classes of row-convex and tree-convex constraints are examples of constraint classes which have the Helly property. More importantly, we presented the algorithm **DPC***, a simple variant of **DPC**, which can decide the CSP of any majority-closed constraint language, and is sufficient for guaranteeing backtrack-free search for majority-closed constraint networks, which have found applications in various domains, such as scene labeling, temporal reasoning, geometric reasoning, and logical filtering. Our evaluations also show that **DPC*** significantly outperforms the state-of-the-art algorithms for solving majority-closed constraint networks.

Chapter 4

A Distributed Partial PC Algorithm for Solving CRC Constraint Networks

4.1 Contribution

The class of CRC constraints generalizes several tractable classes of constraints and is expressive enough to model problems in domains such as temporal reasoning, geometric reasoning, and scene labelling. This chapter presents the first distributed deterministic algorithm for connected row-convex (CRC) constraints. Our distributed (partial) path consistency algorithm efficiently transforms a CRC constraint network into an equivalent constraint network, where all constraints are minimal (i.e., they are the tightest constraints) and generating all solutions can be done in a backtrack-free manner. When compared with the state-of-the-art distributed algorithm for CRC constraints, which is a randomized one, our algorithm guarantees to generate a solution for satisfiable CRC constraint networks and it is applicable to solve large networks in real distributed systems. The evaluations show that our algorithm outperforms the state-of-the-art algorithm in both practice and theory.

4.2 Introduction and Chapter Outline

Although solving general constraint satisfaction problems (CSPs) is known to be NP-hard, many subclasses have been identified as tractable. The class of *connected row-convex* (CRC) constraints [40] is an important tractable subclass of CSPs, which generalizes several subclasses of constraints such as 2SAT, binary integer linear constraints, and monotone constraints [40]. The CRC constraint class is very expressive

and can model problems in domains such as temporal reasoning [99, 73], VLSI design [15], geometric reasoning [72], scene labelling [7] as well as logical filtering [76].

In this chapter we are interested in handling CRC constraints from a distributed CSP perspective. Modelling problems from a distributed perspective is attractive when *privacy* and *autonomy* are of concern, as the conventional centralized approaches are often not applicable in this case. The following example illustrates a distributed CSP involving CRC constraints.

Example 4.1. *Two agents A, B have their* private variables *$V_P^A = \{x^A, y^A\}$ and $V_P^B = \{y^B, z^B\}$, respectively, defined on finite domains. The private variables are subject to binary constraints such as $f^A(y^A) + x^A \cdot y^A \leq 0.5$, where f^A is a real-valued function that is either strictly increasing or decreasing. Agents A and B also have* shared variables *$V_S^A = \{z^A, t^A\}$ and $V_S^B = \{x^B, t^B\}$, respectively, which are connected with other agent's shared variables through constraints, such as $f^A(z^A) + x^B \leq 0.3$. They can be also connected with the agent's private variables through constraints. Each agent has control over only its own variables, where the values of its private variables are not known to the other agent. Figure 4.1 illus-*

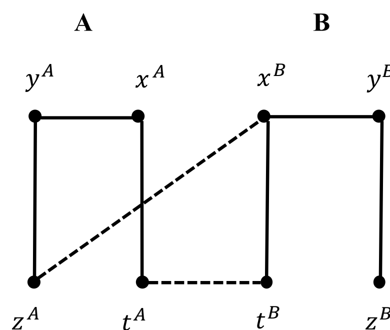


Figure 4.1 : A constraint graph of Example 4.1.

trates an example constraint graph, where all edges correspond to the constraints mentioned in the example. All constraints here are CRC [40]. The problem cannot be solved in a centralized way; a distributed algorithm is therefore required so as to

take consideration of the privacy of agents.

In the literature, Yokoo *et al.* [130] proposed a distributed backtracking algorithm for *general* distributed CSPs and, since then, efficient algorithms have been developed for *specific* distributed CSPs that are tailored to the problem domains of interest, *e.g.*, the distributed scheduling problem [85], the distributed plan coordination problem [34], and the distributed simple temporal problem (STP) [14].

Recently, Kumar *et al.* [75] proposed a distributed algorithm, called D-CRC, for solving CRC constraints, which was shown to be more efficient than the state-of-the-art centralized algorithm for CRC constraints. There are, however, several drawbacks of D-CRC: (i) it is based on randomization and as such it does not guarantee to return a solution even when the input CSP is consistent; and (ii) it cannot determine the inconsistency of the input; and (iii) it cannot assign more than one variable to each agent, which makes the algorithm unrealistic to solve large networks in real distributed systems.

The previous observations suggest us that designing an efficient *deterministic* distributed algorithm for reasoning with CRC constraints is critically important. To tackle this problem, we adopt in this chapter a powerful method, called P^{3}C [104], which was originally designed to solve simple temporal networks (STNs) and makes use of the convex property of simple temporal constraints. Recently, Boerkoel and Durfee [14] successfully extended P^{3}C to STNs in distributed settings, which suggests that their approach can also be adapted to distributed solving of other convex constraints, such as CRC constraints. However, while the classes of CRC constraints and simple temporal constraints both share the convex properties, it was observed that, as opposed to simple temporal constraints, CRC constraints are not distributive in the sense that the relational composition operation is not distributive over nonempty intersections [86]. This presents a significant obstacle for a straightfor-

ward adaptation of the machinery devised for STNs to CRC constraints.

In this chapter, we present a deterministic distributed algorithm, called D∆CRC, for solving CRC constraint networks. To this end, we prove that P^3C can indeed be adapted to transform an input CRC constraint network into an equivalent constraint network, where all constraints are minimal (*i.e.*, they are the tightest constraints) and generating all solutions can be done in a backtrack-free manner. Then we extend the result to distributed settings by employing the communication mechanism by Boerkoel and Durfee [14] that was originally devised for distributed STNs. Our algorithm D∆CRC does not suffer from the aforementioned deficiencies that the state-of-the-art algorithm D-CRC has, and it is more efficient than D-CRC in theory as well as in practice.

The remainder of the chapter is organized as follows. After a short introduction of basic definitions and notations in section 4.3, We propose in section 4.4 a centralized strong partial path consistency algorithm (△CRC) for CRC constraint networks. Then in section 4.5 we extend this to a distributed setting and present the distributed △CRC algorithm (D∆CRC) for CRC constraint networks, where we adopt the communication mechanism from [14]. In section 4.6 we evaluate the D∆CRC algorithm against the state-of-the-art algorithm for solving CRC constraints and conclude the chapter in section 4.7 with further discussion.

4.3 Preliminaries

Basic notions and results about binary constraint networks, triangulated constraint graphs, and connected row-convex constraints can be found in section 1.3.

4.4 An Efficient Centralized Algorithm for CRC Constraints

In this section we present an efficient algorithm to enforce strong partial path-consistency (PPC) on CRC constraint networks. The algorithm, called △CRC, is

adapted from the well-known P^{3}C algorithm [104], which enforces PPC on simple temporal networks and was further generalized to enforce PPC on qualitative constraint networks defined over distributive subalgebras [87].

As opposed to simple temporal constraints and distributive subalgebras, CRC constraints are defined over finite domains and are not distributive [86] (*i.e.*, the relational composition operation is not distributive over nonempty intersections of CRC constraints). Consequently, naively adapting the algorithms from [104, 87] to CRC constraints does not lead to an algorithm that enforces strong PPC on CRC constraint networks. We solve this in the following way:

First, as the domains of CRC networks are finite, arc-consistency (AC) is not automatically achieved with a naive adaptation of P^{3}C. Therefore, including an AC enforcing mechanism in the algorithm is necessary for enforcing strong PPC on the CRC networks. Instead of simply including an AC enforcing mechanism as a preprocessing procedure (cf. [135]) that can cause extra computing effort, we integrate the AC enforcing mechanism tightly into the algorithm.

Second, contrary to the proofs in [104, 86], which implicitly make use of distributivity of constraints, we prove that $\triangle$CRC enforces strong PPC without making use of distributivity of CRC constraints (cf. Theroem 4.1).

Our algorithm $\triangle$CRC is presented as Algorithm 4.1. It takes as input a CRC network $\mathcal{N} = \langle V, D, C \rangle$ and an ordering $\prec = (v_n, \dots, v_1)$ on V. It first eliminates variables v_k along the ordering $\prec$ and propagates the information of constraints involving v_k to its successors $F_k := \{v_i \mid R_{ki} \in C, i < k\}$ by using the following elimination rule.

Eliminate $\mathbf{v_k}$: For all $v_i, v_j \in F_k$ update R_{ij} as $R_{ij} \cap (R_{ik} \circ R_{kj})$ and update D_i as $D_i \cap R_{ki}(D_k)$.

Algorithm 4.1: △CRC

Input : A constraint network $\mathcal{N} = \langle V, D, C\rangle$, where C is a set of CRC constraints; An ordering $\prec = (v_n, \ldots, v_1)$ on V.

Output: A triangulated PPC network that is equivalent to $\mathcal{N}$ if $\mathcal{N}$ is consistent, or "inconsistent".

```
   // Phase 1: Eliminate variables
1  for k ← n to 2 do
2      (result, N) ← Eliminate(v_k, N, ≺)
3      if result = False then
4          return "inconsistent"
   // Phase 2: Reinstate variables
5  for k ← 2 to n do
6      N ← Reinstate(v_k, N, ≺)
7  return N
8  Function Eliminate(v_k, N = ⟨V, D, C⟩, ≺)
9      for i ← k − 1 to 1 s.t. R_ik ∈ C do
10         for j ← i − 1 to 1 s.t. R_jk ∈ C do
11             if R_ij ∉ C then
12                 add constraint R_ij := D_i × D_j to C
13             R_ij ← R_ij ∩ (R_ik ∘ R_kj)
14             if R_ij = ∅ then
15                 return (False, N)
16         D_i ← D_i ∩ R_ki(D_k)
17         if D_i = ∅ then
18             return (False, N)
19     return (True, N)
20 Function Reinstate(v_k, N = ⟨V, D, C⟩, ≺)
21     C* ← C
22     for i ← k−1 to 1 s.t. R_ik ∈ C do
23         for j ← i−1 to 1 s.t. R_jk ∈ C do
               // R*_kj and R*_ki are in C*
24             R_ki ← R_ki ∩ (R*_kj ∘ R_ji)
25             R_kj ← R_kj ∩ (R*_ki ∘ R_ij)
26         D_k ← D_k ∩ R_ik(D_i)
27     return ⟨V, D, C⟩
```

After the elimination phase, Algorithm 4.1 reinstates the variables according the inverse ordering $\prec^{-1}$ and propagates the information of the constraints back to the neighboring predecessors.

Reinstate $\mathbf{v_k}$: For all $v_i, v_j \in F_k$, update R_{ik} as $R_{ik} \cap (R_{ij} \circ R_{jk})$ and R_{jk} as $R_{jk} \cap (R_{ji} \circ R_{ik})$ and update D_k as $D_k \cap R_{ik}(D_i)$.

We first show that Algorithm 4.1 outputs a triangulated network.

Lemma 4.1. *Let $(\mathcal{N}, \prec)$ be an input to Algorithm 4.1 where $\mathcal{N}$ is consistent. Then Algorithm 4.1 outputs a triangulated CRC constraint network $\mathcal{N}'$ which is equivalent to $\mathcal{N}$ and has $\prec$ as its perfect elimination ordering.*

Proof. Suppose $\mathcal{N}$ is consistent and $\mathcal{N}'$ is the output of Algorithm 4.1. Because CRC constraints are closed under intersection and composition (cf. Lemma 1.1), $\mathcal{N}'$ is also a CRC constraint network. Moreover, as the operations (*i.e.*, intersection, composition and adding universal relations) in Algorithm 4.1 do not modify the solution space of $\mathcal{N}$, $\mathcal{N}'$ is equivalent to $\mathcal{N}$.

We next prove that $\mathcal{N}'$ is triangulated. Write $\mathcal{N}^d = \langle V, D, C^d \rangle$ for the network obtained after the elimination phase (lines 1–4) of Algorithm 4.1. For any $1 < k \leq n$, note that the set of variables F_k induces a subnetwork of $\mathcal{N}^d$ whose constraint graph is complete. This shows that $\prec = (v_n, \ldots, v_1)$ is a perfect vertex elimination ordering in $G_{\mathcal{N}^d}$. Since the reinstatement phase does not alter the graph structure, $\prec$ is also a perfect vertex elimination ordering in the constraint graph of $\mathcal{N}'$. By Proposition 1.1, we know $\mathcal{N}^d$ and $\mathcal{N}'$ are both triangulated graphs. □

Algorithm 4.1 also decides the consistency of CRC constraint networks. The proof uses the *Helly property* as stated in the following lemma.

Lemma 4.2. [7] *Let F be a finite collection of (0,1)-row vectors that are row-convex and of equal length such that every pair of row vectors in F have a non-zero entry in common. Then all row vectors in F have a non-zero entry in common.*

The following result follows from [135, Theorem 1]. For completeness we include a proof here.

Proposition 4.1. *Let $(\mathcal{N}, \prec)$ be an input to Algorithm 4.1. Then $\mathcal{N}$ is consistent if and only if the algorithm does not return "inconsistent" in the elimination phase.*

Proof. Suppose that the input constraint network $\mathcal{N}$ is consistent. Then, because the operations in Algorithm 4.1 do not modify the solution set of $\mathcal{N}$, the algorithm does not find any empty relation in lines 14–15 or empty domain in lines 17–18. Thus, it does not return "inconsistent" (line 4).

Let $\mathcal{N}^d = \langle V, D, C^d \rangle$ be the network obtained after the elimination phase (lines 1–4) of Algorithm 4.1. Now suppose that the elimination step (lines 1–4) of the algorithm did not return "inconsistent" as its output. We show that $\mathcal{N}^d$ is consistent. Let $\mathcal{N}_k^d$ be the subnetwork of $\mathcal{N}^d$ induced by variables $v_1, \ldots, v_k$. We claim that $\mathcal{N}_k^d$ is consistent for $k = 1, \ldots, n$ and prove the claim by induction on k. First, $\mathcal{N}_1^d$ is consistent, because D_1 is not empty; otherwise the elimination step would have detected the inconsistency in lines 17–18. Now, suppose that $\mathcal{N}_k^d$ is consistent and let $A_k := \langle a_1, \ldots, a_k \rangle \in D_1 \times \cdots \times D_k$ be a solution for $\mathcal{N}_k^d$. We show that A_k can be extended to a solution $A_{k+1} = \langle a_1, \ldots, a_{k+1} \rangle$ for $\mathcal{N}_{k+1}^d$. To this end, we need to show that $\langle a_i, a_{k+1} \rangle \in R_{i,k+1}$ for all $i \leq k$ with $R_{i,k+1} \in C^d$. Observe that after the elimination phase, (i) for any variable v_i with $i \leq k$, $R_{i,k+1}$ is arc-consistent, and (ii) for any pair of variables (v_i, v_j) with $i, j \leq k$ with $R_{i,k+1}, R_{j,k+1} \in C^d$, we have that $R_{ij} \subseteq R_{i,k+1} \circ R_{k+1,j}$. Since $\langle a_i, a_j \rangle \in R_{ij}$, we also have $R_{i,k+1}(a_i) \cap R_{j,k+1}(a_j) \neq \varnothing$ for all $i, j \leq k$ with $R_{i,k+1}, R_{j,k+1} \in C^d$.

Then, by Lemma 4.2, the intersection of all $R_{i,k+1}(a_i)$ with $i \leq k, R_{i,k+1} \in C^d$ is not empty. Thus there exists a value a_{k+1} with $\langle a_i, a_{k+1} \rangle \in R_{i,k+1}$ for all $i \leq k, R_{i,k+1} \in C^d$ and we showed that $A_{k+1} = \langle a_1, \ldots, a_{k+1} \rangle$ is a solution of $\mathcal{N}_{k+1}^d$. Thus $\mathcal{N}_k^d$ is consistent for $k = 1, \ldots, n$ and in particular for $\mathcal{N}^d = \mathcal{N}_n^d$. Since operations in Algorithm 4.1 do not modify the solution set of $\mathcal{N}$, networks $\mathcal{N}^d$ and $\mathcal{N}$ are equivalent. Thus, $\mathcal{N}$ is consistent. $\square$

The preceding proof also shows that one can generate *all* solutions of a consistent input network *backtrack-free* by applying $\triangle$CRC and instantiating the variables along the inverse of the input ordering $\prec$. This is because for all $1 \leq k < n$ a solution $\langle a_1, \ldots, a_k \rangle$ of $\mathcal{N}_k^d$ can be extended to a solution $\langle a_1, \ldots, a_{k+1} \rangle$ of $\mathcal{N}_{k+1}^d$ by choosing an element a_{k+1} from the intersection of all $R_{i,k+1}(a_i)$ with $i \leq k$, $R_{i,k+1} \in C^d$, which is not empty as shown in the preceding proof. This proves the following result.

Proposition 4.2. *Algorithm 4.1 returns, on input* $(\mathcal{N}, \prec)$*, an equivalent CRC network* $\mathcal{N}'$ *that is decomposable with respect to the inverse of* $\prec$*, if* $\mathcal{N}$ *is consistent.*

We next show that Algorithm 4.1 also enforces strong PPC.

Theorem 4.1. *Algorithm 4.1 returns, on input* $(\mathcal{N}, \prec)$*, a network* $\mathcal{N}'$ *that is AC and PPC, if* $\mathcal{N}$ *is consistent.*

Proof. Suppose $\mathcal{N}$ is consistent. Let $\mathcal{N}' = \langle V, D, C \rangle$ be the output of Algorithm 4.1 on $(\mathcal{N}, \prec)$, and $\mathcal{N}'_k$ be the subnetwork of $\mathcal{N}'$ induced by $v_1, v_2, ..., v_k$. We note that $\mathcal{N}'_k$ is triangulated for all k (cf. Lemma 4.1). Now we claim that $\mathcal{N}'_k$ is AC and PPC for $1 \leq k \leq n$. We prove the claim by induction on k.

First, $\mathcal{N}'_1$ is trivially AC and PPC, as D_1 is not empty.

Now suppose $\mathcal{N}'_{k-1}$ is AC and PPC. We first prove that $\mathcal{N}'_k$ is AC. Let $i, j < k$. Then all $R_{ij} \in C$ are AC by the induction hypothesis. Now suppose $R_{ik} \in C$. Note that R_{ki} is AC due to line 26 in Algorithm 4.1. We show that R_{ik} is also AC. If there is no $j \neq i$ with $v_j \in F_k$, then R_{ik} is AC by line16. Thus suppose there exists $j \neq i$ with $v_j \in F_k$ and let $a_i \in D_i$. Then by the induction hypothesis, R_{ij} is AC and PPC with respect to $\mathcal{N}'_{k-1}$, thus by Corollary 1.1 it is minimal with respect to $\mathcal{N}'_{k-1}$. There exists a solution for $\mathcal{N}'_{k-1}$ which assigns a_i to v_i. Furthermore, this solution can be extended to a solution σ of $\mathcal{N}'$ as $\mathcal{N}'$ is decomposable with respect to the inverse of $\prec$ by Proposition 4.2. Suppose σ assigns a_k to v_k. Then $(a_i, a_k) \in R_{ik}$

holds. This proves that $\mathcal{N}'_k$ is AC.

We now show that $\mathcal{N}'_k$ is PPC. Since all paths π in $\mathcal{N}'_{k-1}$ are PC by the induction hypothesis, we can assume that π includes v_k, and because $\mathcal{N}'_k$ is triangulated, by Theorem 1.1 we can assume that π is of length 2.

Suppose π has the form (v_i, v_k, v_j) with $i, j < k$ (see Figure 4.2 for an illustration). By induction hypothesis $\mathcal{N}'_{k-1}$ is PPC and, by Corollary 1.1, R_{ij} is minimal with

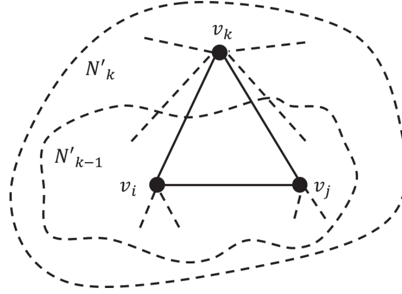


Figure 4.2 : Illustration of proof of Theorem 4.1.

respect to $\mathcal{N}'_{k-1}$. Then there exists a solution of $\mathcal{N}'_{k-1}$ that assigns values a_i and a_j to variables v_i and v_j, respectively. This solution can be extended to a solution σ of $\mathcal{N}'$ by Proposition 4.2. Suppose σ assigns a_k to v_k. Then, $(a_i, a_k) \in R_{ik}$ and $(a_k, a_j) \in R_{kj}$. Thus (v_i, v_k, v_j) is PC.

Now suppose π has the form (v_i, v_j, v_k) with $i, j < k$. We show that for any $(a_i, a_k) \in R_{ik}$ there exists a_j such that $(a_i, a_j) \in R_{ij}$ and $(a_j, a_k) \in R_{jk}$, *i.e.*, $R_{ij}(a_i) \cap R_{kj}(a_k) \neq \varnothing$. By lines 22–25 in the algorithm, we have for all $\mu < k$ with $R_{k\mu} \in C$ that

$$R_{k\mu} = R^*_{k\mu} \cap \bigcap_{v_\nu \in F_k} (R^*_{k\nu} \circ R_{\nu\mu}) \tag{4.1}$$

This in particular means that

$$R_{kj} = R^*_{kj} \cap \bigcap_{v_\nu \in F_k} (R^*_{k\nu} \circ R_{\nu j}) \tag{4.2}$$

Thus, to show that $R_{ij}(a_i) \cap R_{kj}(a_k)$ is not empty, it suffice to show by the Helly property (cf. Lemma 4.2) that the intersection of any two of the following sets is not empty:

$$R_{ij}(a_i), \quad R^*_{kj}(a_k), \quad \bigcap_{v_\nu \in F_k} (R^*_{k\nu} \circ R_{\nu j})(a_k)$$

First, from equation (4.1) it follows that $R_{ki} \subseteq R^*_{k\nu} \circ R_{\nu i}$ for all $v_\nu \in F_k$. Thus $(a_k, a_i) \in R_{ki} \subseteq R^*_{kj} \circ R_{ji}$ and the intersection of $R_{ij}(a_i)$ and $R^*_{kj}(a_k)$ is not empty.

Then, because $R_{kj} \neq \varnothing$ is AC by the induction hypothesis, we know $R_{kj}(a_k) \neq \varnothing$. Then equation (4.2) implies that the intersection of $R^*_{kj}(a_k)$ and $\bigcap_{v_\nu \in F_k}(R^*_{k\nu} \circ R_{\nu j})(a_k)$ is not empty.

Finally, for an arbitrary $\nu < k$ with $v_\nu \in F_k$ we have by equation (4.1) that $R_{ki} \subseteq R^*_{k\nu} \circ R_{\nu i}$. Since $R_{\nu i}$ is PC with respect to $\mathcal{N}'_{k-1}$ we also have that $R_{ki} \subseteq R^*_{k\nu} \circ (R_{\nu j} \circ R_{ji})$. Then, because $(a_k, a_i) \in R_{ki} \neq \varnothing$, we have that $(a_k, a_i) \in (R^*_{k\nu} \circ R_{\nu j}) \circ R_{ji}$. Thus, the intersection of $R_{ij}(a_i)$ and $(R^*_{k\nu} \circ R_{\nu j})(a_k)$ is not empty for all $\nu < k$ with $v_\nu \in F_k$. Then, by the Helly property (cf. Lemma 4.2), the intersection of $R_{ij}(a_i)$ and $\bigcap_{v_\nu \in F_k}(R^*_{k\nu} \circ R_{\nu j})(a_k)$ is not empty. Altogether, we have that $R_{ij}(a_i) \cap R_{kj}(a_k) \neq \varnothing$ and have showed π is PC with respect to $\mathcal{N}'_k$.

In summary, we have proved that $\mathcal{N}'_k$ is AC and PPC for all $1 \leq k \leq n$. Thus $\mathcal{N}' = \mathcal{N}'_n$ is AC and PPC. □

By Corollary 1.1 and Theorem 4.1, we have the following result.

Corollary 4.1. *Algorithm 4.1 transforms an input CRC constraint network into an equivalent network where all its constraints are minimal, if no inconsistency is detected.*

By the above result, the minimal network of a consistent CRC constraint network $\mathcal{N}$ can be computed as follows: (i) complete $\mathcal{N}$ by adding edges labeled with universal constraints, and (ii) apply Algorithm 4.1 on the completion of $\mathcal{N}$.

We now turn our attention to the analysis of the time complexity of Algorithm 4.1. We first recall some concepts in graph theory required for the analysis of the time complexity of Algorithm 4.1.

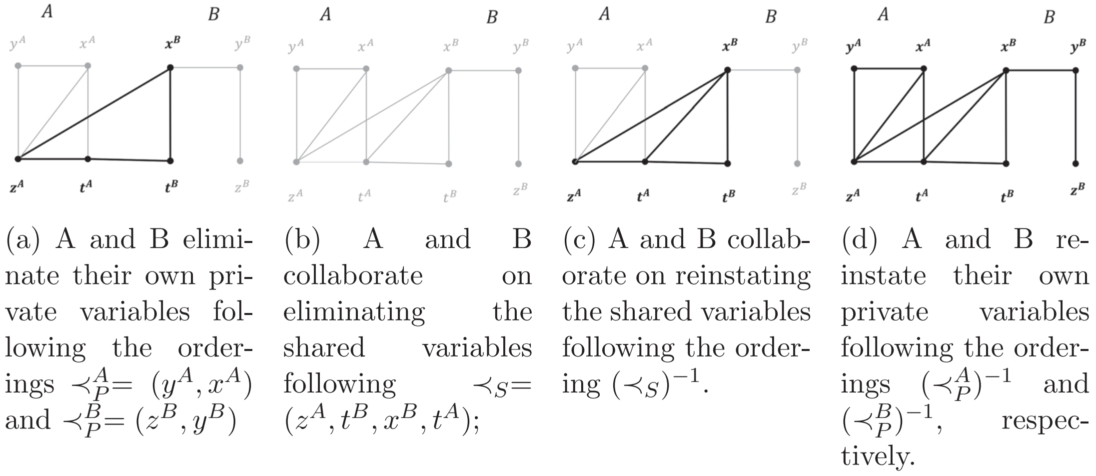
An *ordered graph* is a pair $(G, \prec)$, where $G = (V, E)$ is an undirected graph and $\prec$ is an ordering on V. The nodes adjacent to v that succeeds it in the ordering are called its *children*. The *width of a node* in an ordered graph is its number of children. The *width of an ordering* $\prec$, denoted by $w(\prec)$, is the maximum width among all of its nodes. The *induced graph* of an ordered graph $(G, \prec)$ is an ordered graph $(G^*, \prec)$, where $G^* = (V, E^*)$ is obtained from G as follows: the nodes of G are processed from first to last with respect to $\prec$; when a node v is processed, we connect all of its children with edges. The *induced width* of an ordering $\prec$, denoted by $w^*(\prec)$, is the width of the ordering $\prec$ with respect to G^*.

Proposition 4.3. *Algorithm 4.1 runs in time $O(w^*(\prec)ed)$, where e is the number of edges of the induced graph of $G_{\mathcal{N}}$ and d is the largest domain size.*

Proof. Because the elimination phase and the reinstatement phase have the same time complexity, we will only consider the elimination phase. Let $(G^* = (V, E^*), \prec)$ be the induced graph of $(G_{\mathcal{N}}, \prec)$. Given $v_k \in V$, let $F_k = \{v_i \mid i < k, e_{ik} \in G_{\mathcal{N}}\}$. We first analyze the time complexity of function ELIMINATE. Lines 11–18 are executed at most $|F_k|^2$ times. Since both the composition operation and the intersection operation run in $O(d)$ [135, 40], the operations in line 13 and line 16 takes $O(d)$ time. We can conclude that it takes $O(|F_k|^2 d)$ time for function ELIMINATE to eliminate variable v_k. Therefore, the time complexity of the elimination phase is $O(\sum_{k=1}^{n} |F_k|^2 d)$. Because $|F_k| \leq w^*(\prec)$, we have

$$\sum_{k=1}^{n} |F_k|^2 \leq \sum_{k=1}^{n} (|F_k| w^*(\prec)) = w^*(\prec) \sum_{k=1}^{n} |F_k| \tag{4.3}$$

$$= w^*(\prec)e, \tag{4.4}$$



(a) A and B eliminate their own private variables following the orderings $\prec_P^A = (y^A, x^A)$ and $\prec_P^B = (z^B, y^B)$

(b) A and B collaborate on eliminating the shared variables following $\prec_S = (z^A, t^B, x^B, t^A)$;

(c) A and B collaborate on reinstating the shared variables following the ordering $(\prec_S)^{-1}$.

(d) A and B reinstate their own private variables following the orderings $(\prec_P^A)^{-1}$ and $(\prec_P^B)^{-1}$, respectively.

Figure 4.3 : Execution of DΔCRC on Example 4.1.

where $e = |E^*|$. Therefore, Algorithm 4.1 runs in $O(w^*(\prec)ed)$. □

Note that when the constraint network is complete, Algorithm 4.1 runs in time $O(n^3d)$. This can be compared with PC-CRC [40], the state-of-the-art PC enforcing algorithm for CRC constraints, which runs in time $O(n^3d^2)$. It is worth mentioning that Algorithm 4.1 performs even more efficiently when the input networks are sparse.

4.5 An Efficient Distributed Algorithm for CRC Constraints

In this section, we extend $\triangle$CRC to a distributed algorithm, called DΔCRC, for solving distributed CRC networks. Except for some details, we follow the work by Boerkoel and Durfee [14], which extends P^{3}C to solving simple temporal networks in distributed settings. For further details, we refer the readers to [14].

Recall that the definition of a binary distributed constraint network, where each agent owns a portion of the whole constraint network, is given in Definition 1.3.

We note that in Definition 6.5 each local constraint network $\mathcal{N}_i$ corresponds to an agent i and that C_X is a set of constraints that are shared by two different agents.

Algorithm 4.2: DΔCRC

Input : $\mathcal{N}_i = \langle V_i, D_i, C_i \rangle$: Agent i's part of a distributed constraint network $\mathcal{M}$.
$\prec^i_P = (v_p, \ldots, v_1)$: $\mathcal{N}_i$'s private variable elimination ordering.
$\prec_S = (w_s, \ldots, w_1)$: $\mathcal{M}$'s shared variable elimination ordering.

Output: Agent i's part of a PPC constraint network that is equivalent to $\mathcal{M}$.

// Phase 1: Eliminate private variables
1 **for** $\ell \leftarrow p$ **to** 1 **do**
2   (Result, $\mathcal{N}_i$) $\leftarrow$ ELIMINATE(v_ℓ, $\mathcal{N}_i$, $\prec^i_P$);
3   **if** $\mathcal{N}_i$ = False **then**
4   Broadcast "inconsistent"
5   **return** "inconsistent"

// Phase 2: Eliminate shared variables
6 **for** $\ell \leftarrow s$ **to** 1 **s.t.** $w_\ell \in V^i_S$ **do**
7   **foreach** $w_j \in V^i_X$ **s.t.** $j > \ell$ *and* $R_{j\ell} \in C_X$ **do**
8   Wait for the elimination of w_j by other agent and for the updated information about constraints involving w_ℓ.
9   (Result, $\mathcal{N}_i$) $\leftarrow$ ELIMINATE(w_ℓ, $\mathcal{N}_i$, $\prec_S$,)
10   **if** Result = False **then**
11   Broadcast "inconsistent"
12   **return** "inconsistent"
13   **else**
14   **for** $j, k < \ell$ **s.t.** $R_{j\ell}, R_{\ell k} \in C_X$ **do**
15   Send updated information about R_{jk} to the agents to whom variables w_j and w_k belong.

// Phase 3: Reinstate shared variables
16 **for** $\ell \leftarrow 1$ **to** s **s.t.** $w_\ell \in V^i_S$ **do**
17   **for** $j \leftarrow \ell - 1$ **to** 1 **s.t.** $R_{j\ell} \in C_X$ **do**
18   **for** $k \leftarrow j - 1$ **to** 1 **s.t.** $R_{k\ell} \in C_X$ **do**
19   Wait for the reinstatement of w_j and w_k by another agents and for the updated information about R_{jk}.
20             $R_{j\ell} \leftarrow R_{j\ell} \cap (R_{jk} \circ R_{k\ell})$
21             $R_{k\ell} \leftarrow R_{k\ell} \cap (R_{kj} \circ R_{j\ell})$
22         $D_\ell \leftarrow D_\ell \cap R_{j\ell}(D_j)$
23   **for** $k \leftarrow \ell + 1$ **to** s **s.t.** $w_k \in V^i_X$, $R_{k\ell}, R_{kj} \in C_X$ **do**
24   Send updated relation about $R_{j\ell}$ to the agent to whom variable w_k belongs.

// Phase 4: Reinstate private variables
25 **for** $\ell \leftarrow 1$ **to** p **do**
26     $\mathcal{N}_i \leftarrow$ REINSTATE(v_ℓ, $\mathcal{N}_i$, $\prec^i_P$)

Definition 4.1. *Given a binary distributed constraint network* $\mathcal{M} = (\mathcal{P}, C_X)$*, a variable* $v \in V_j$ *is called an* external variable *of agent* i *if there is an external*

constraint $((v, w), R) \in C_X$ *with* $v \in V_i$ *and* $w \in V_j$, $i \neq j$. *We use the notation* V_X^i *for the set of external variables of agent* i.

Definition 4.2. *Let* $\mathcal{V}(C_X)$ *be the set of all variables that appear in some constraint in* C_X. *Then each* V_i *can be partitioned into two disjoint sets: the* private variable *set* $V_P^i = \{v_i \mid v_i \in V_i, v_i \notin \mathcal{V}(C_X)\}$ *and the* shared variable *set* $V_S^i = \{v_i \mid v_i \in V_i, v_i \in \mathcal{V}(C_X)\}$. *The* private subnetwork *of* $\mathcal{N}_i$, *denoted by* $\mathcal{N}_p^i$, *is the subnetwork induced by* V_P^i, *and the* shared subnetwork *of* $\mathcal{M}$, *denoted by* $\mathcal{M}_S$, *is the subnetwork induced by* $\bigcup_{\mathcal{N}_i \in \mathcal{P}} V_S^i$. $\prec_S$ *is an ordering of variables of the shared subnetwork and* $\prec_P^i$ *is an ordering of agent i's private variables.*

DΔCRC, which is presented as Algorithm 4.2, is a distributed version of Algorithm 4.1. Given a distributed constraint network $\mathcal{M}$, the algorithm takes as its input agent i's part of $\mathcal{M}$ (*i.e.*, $\mathcal{N}_i = \langle V_i, D_i, C_i \rangle$) as well as private and shared elimination orderings $\prec_P^i$ and $\prec_S$. Like Algorithm 4.1, Algorithm 4.2 eliminates all variables of $\mathcal{N}_i$ and then reinstates them. Concerning private variables in V_P^i, agent i can eliminate and reinstate them independently. However, agent i needs to collaborate with other agents that are connected through external constraints so as to correctly eliminate and reinstate shared variables in V_S^i. In order to avoid using outdated information, the algorithm adopts the communication mechanism of the distributed PPC algorithm introduced in [14].

We explain the algorithm based on Example 4.1.

Phase 1: Agent i eliminates its private variables along the ordering $\prec_P^i$ independently (see Figure 4.3a).

Phase 2: Agent i eliminates its shared variables along the ordering $\prec_S$. Before eliminating a shared variable $v \in V_S^i$, agent i must wait for possible updates related to v (line 8). In Figure 4.3b, when agent B eliminates x^B, because

z^A precedes x^B and connected to x^B, z^A should be eliminated before x^B. Therefore, agent B must wait for agent A to eliminate z^A and update $R_{t^A x^B}$.

Phase 3: Agent i reinstates its shared variables along the ordering $(\prec_S)^{-1}$. When agent i reinstates a variable $w_\ell \in V_S^i$, for any pair (w_j, w_k) of other agents' shared variables that precede w_ℓ and connected to w_ℓ through a relation, agent i must wait for all updates on R_{jk} before updating $R_{\ell j}$ and $R_{\ell k}$ using R_{jk} (line 19). In Figure 4.3c, when agent A reinstates variable z^A, it needs to update relations $R_{z^A x^B}$ and $R_{z^A t^A}$ using $R_{t^A x^B}$. In order to avoid using outdated information, all possible changes on $R_{t^A x^B}$ must be made beforehand.

Phase 4: Agent i reinstates its private variables following the ordering $(\prec_P^i)^{-1}$ (see Figure 4.3d).

We can prove the following using the proof idea in [14, Theorem 6]

Theorem 4.2. *Algorithm 4.2 decides the consistency of its input distributed constraint network and enforces strong PPC on it if no inconsistency is detected.*

Because Algorithm 4.2 returns the same networks as the its centralized counterpart Algorithm 4.1, we can also show the following results similarly to Proposition 4.2 and Theorem 4.1,

Proposition 4.4. *Given an input $(\mathcal{M}, \prec_P^1, \ldots, \prec_P^m, \prec_S)$, Algorithm 4.2 returns a network $\mathcal{M}'$ that is decomposable with respect to the inverse of $\prec = (\prec_P^1, \ldots, \prec_P^m, \prec_S)$, if $\mathcal{M}$ is consistent.*

Theorem 4.3. *Algorithm 4.2 transforms a consistent input CRC constraint network into an equivalent one where all its constraints are minimal.*

Consequently, agents can jointly generate any solution of a consistent $\mathcal{M}$ backtrack-free by applying Algorithm 4.2 to it and instantiating the variables following the inverse of $\prec = (\prec_P^1, \ldots, \prec_P^m, \prec_S)$.

Proposition 4.5. *Algorithm 4.2 has the same time complexity as its centralized counterpart Algorithm 4.1.*

Proof. At each constraint update at most one message is sent or received. Thus the runtime added for communication is $O(e)$, where e is the number of edges of the triangulated constraint graph $G_{\mathcal{N}_i}$. Since the agents may need to wait for constraints update from other agents, in the worst case, the elimination and reinstatement of all shared variables must be done sequentially. Consequently, Algorithm 4.2 has the same time complexity class as Algorithm 4.1. □

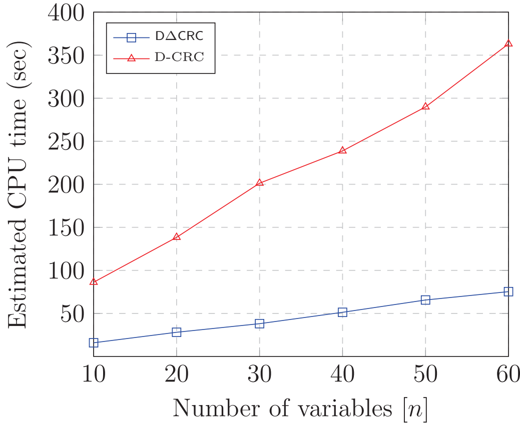
4.6 Evaluations

In this section we theoretically and experimentally compare our algorithm DΔCRC against the state-of-the-art algorithm D-CRC (sections 4.6.1 and 4.6.2). Furthermore, we analyze our algorithm in more detail (section 4.6.3).

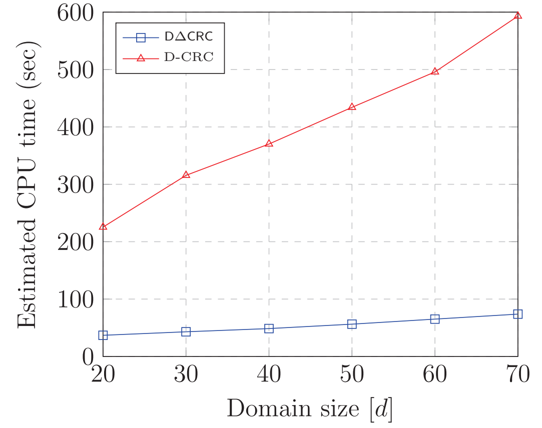
For the experimental comparisons we implemented both DΔCRC and D-CRC on the FRODO 2.0 [80] platform that simulates parallel algorithms for CSPs on a single machine. We measured the computing time of both algorithms using the *simulated time metric* [116], which is a common metric for computing times of parallel algorithms that run in a simulated environment. In all our experiments we set for both distributed algorithms the default communication latency to zero. Our experiments were carried out on a computer with an Intel Core i5-4570 processor with a 3.2 GHz frequency per CPU core, 4 GB memory.

4.6.1 Theoretical Comparisons

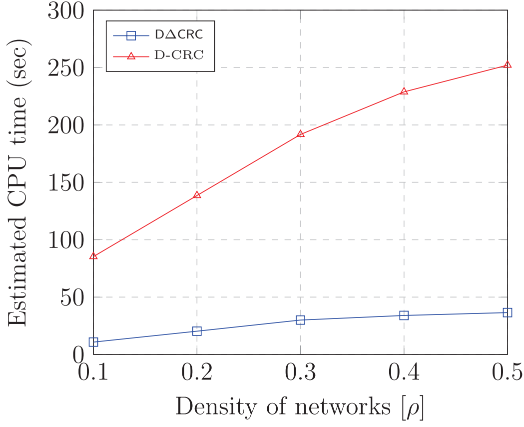
First of all, D-CRC by Kumar *et al.* [75] is a randomized algorithm for solving CRC constraint networks. As such, D-CRC cannot detect inconsistency of an input CRC constraint network; at best, the user can stop it after a time-out to declare the input as inconsistent; nevertheless this does not guarantee the inconsistency of



(a) Performance evaluation of the two algorithms in the number n of variables. We set $\rho = 0.5$ and $d = 20$.



(b) Performance evaluation of the two algorithms in the size d of domains. We set $\rho = 0.5$ and $n = 30$.



(c) Performance evaluation of the two algorithms in the density ρ of the input networks. We set $n = 30$ and $d = 20$.

Figure 4.4 : Performance comparisons between DΔCRC and D-CRC.

its input, *i.e*, it can generate false negatives. By contrast, our algorithm DΔCRC is deterministic, sound and complete.

The expected time complexity of D-CRC is $O(\gamma n^2 d^2)$, where γ is the largest vertex degree of the constraint graph. By contrast, the time complexity of our algorithm DΔCRC is $O(w^*(\prec)ed)$ (see Proposition 4.5). Since $w^*(\prec) \leq \gamma$ and $e \leq n^2$, our algorithm DΔCRC outperforms D-CRC at least by a factor of d on average.

For our experimental comparisons, we also included a preprocessing procedure for D-CRC and procedures for generating the input orderings and solutions for DΔCRC. The theoretical time complexities of those procedures are, however, all dominated by the main algorithms.

4.6.2 Experimental Comparisons

We considered random consistent CRC constraint networks that were used in the literature for evaluations (cf. [40], [135] and [75]). These CRC constraint networks were generated by varying three parameters that affect the time complexity of both algorithms: (i) the number n of variables; (ii) the size d of the largest domain; (iii) the density $\rho = 2|C|/n(n+1)$ of the input CRC constraint network. We fixed two from three parameters and varied the remaining parameter. When we fixed ρ we set it 0.5, as distributed problems usually involve sparse networks, $\rho = 0.5$ meaning a high density value for practical distributed problems.

For the comparisons we assigned to each agent only one single variable (*i.e.*, the number of agents is equal to the number of variables), because D-CRC has the limitation that it does not allow each agent to possess more than one variable. By contrast, our algorithm DΔCRC is more flexible and can assign multiple variables to the agents. But for fair comparisons, we used the same agent setting for D-CRC and DΔCRC.

The graphs in Figures 4.4a–c illustrate the experimental comparisons between algorithms DΔCRC and D-CRC. The data points in each graph are computing times averaged over 20 instances.

We observe in the graphs that both algorithms show linear time behaviors with respect to n and d and a sublinear time behavior with respect to ρ. We also observe that the performance differences between the two algorithms are remarkable. DΔCRC not only runs faster than D-CRC, but it also scales up to 7 times better

than D-CRC with increasing parameter values. This owes to the theoretical property of DΔCRC, *i.e.*, it leverages the input network structure and scales better also in theory with the increasing size of the domain. All in all, we can conclude that DΔCRC is more suitable than D-CRC for distributed problems.

4.6.3 In-depth Evaluation of DΔCRC

As mentioned previously, the setting for the experimental comparisons did not allow assign an agent to two or more variables, *i.e.*, the number of agents had to be equal to the number of variables. Therefore, we also evaluated our algorithm exclusively (see Figure 4.5), where we changed the number n_A of agents in the input networks. Each agent is assigned to around $\lfloor n_A/n \rfloor$ variables that are chosen randomly from the input network.

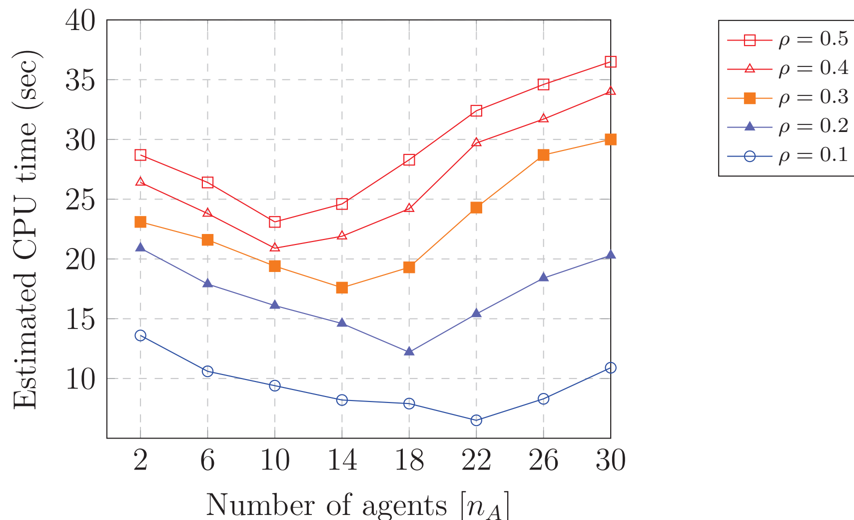


Figure 4.5 : Evaluation of our algorithm DΔCRC for different number of agents and network density. We set $n = 30$ and $d = 20$.

We observe in Figure 4.5 that the performance graph of DΔCRC forms a U-shape for all the network densities. This is because more agents allows for more constraints to be handled concurrently, but from a certain number of agents on, this effect is dominated by the delays caused by the inter-agent communications.

In the figure we also notice that the number of agents needed for the optimal performance shifts to the right with decreasing network density. This phenomenon owes to the fact that networks with lower densities involve less constraints between agents, allowing the agents to handle more constraints concurrently.

The results also show that DΔCRC runs up to two times faster when the agents are assigned to two or more variables and not to only one variable. Consequently, DΔCRC can outperform D-CRC even more significantly.

4.7 Conclusion

In this chapter, we have proposed the first deterministic distributed algorithm, called DΔCRC, for solving CRC constraints. The algorithm can efficiently transform an input CRC constraint network into an equivalent constraint network, where all constraints are minimal, and can generate all solutions in a backtrack-free manner.

DΔCRC does not suffer from the problems that the state-of-the-art algorithm D-CRC has: (i) it is sound and complete and (ii) it can assign more than one variable to each agent, allowing the algorithm to solve large networks in real distributed systems. Furthermore, our theoretical and experimental comparisons showed that DΔCRC significantly outperforms D-CRC.

The results of this chapter can be easily extended to the class of tree-preserving constraints, as they both the classes of CRC constraints and tree-preserving constraints enjoy the Helly property. One may wonder if the results can be extended to other majority closed constraint languages as well. However, since the proofs of Propositon 4.1 and Theorem 4.1 heavily rely on the Helly property, which is not enjoyed by all majority closed constraint languages, we may need to devise new methods to establish these results.

Chapter 5

A Distributed AC Algorithm for Solving Multiagent STPs

5.1 Contribution

The Simple Temporal Problem (STP) is a fundamental temporal reasoning problem and has recently been extended to the Multiagent Simple Temporal Problem (MaSTP). In this chapter we present a novel approach that is based on enforcing arc-consistency (AC) on the input (multiagent) simple temporal network. We show that the AC-based approach is sufficient for solving both the STP and MaSTP and provide efficient algorithms for them. As our AC-based approach does not impose new constraints between agents, it does not violate the privacy of the agents and is superior to the state-of-the-art approach to MaSTP. Empirical evaluations on diverse benchmark datasets also show that our AC-based algorithms for STP and MaSTP are significantly more efficient than existing approaches.

5.2 Introduction and Chapter Outline

The Simple Temporal Problem (STP) [38] is arguably the most well-known quantitative temporal representation framework in AI. The STP considers time points as the variables and represents temporal information by a set of unary or binary constraints, each specifying an interval on the real line. Since its introduction in 1991, the STP has become an essential sub-problem in planning or scheduling problem [2].

While the STP is initially introduced for a single scheduling agent and is solved by centralized algorithms, many real-world applications involve multiple agents who

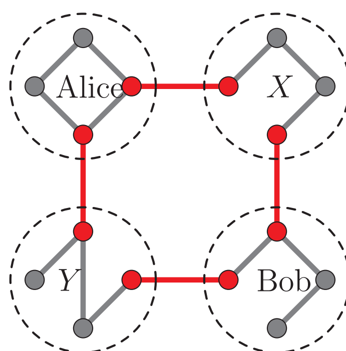


Figure 5.1 : An illustration of Example 5.1. Alice, Bob, company X, company Y are four agents, each owning a local simple temporal network. The circles represent variables and edges constraints. Red edges represent constraints that are shared by two different agents.

interact with each other to find a solution like the following example:

Example 5.1. *When Alice is looking for a position at company X, she might need to arrange an interview appointment with X. Suppose that her colleague Bob is also applying for the position and Alice and Bob are both applying for another position at another company Y. To represent and solve such an interview scheduling problem, we need a multiagent framework (see Figure 5.1 for an illustration).*

Recently, the extension of STP to multiagent STP (MaSTP) has been provided in [14], which presents a formal definition of the MaSTP as well as a distributed algorithm, called D△PPC, for computing the complete joint solution space.

However, as D△PPC is based on the P^{3}C algorithm [104], which triangulates the input constraint graph, it has the drawback of creating new constraints between agents that are possibly not directly connected. In Figure 5.1, D△PPC triangulates the inner cycle by adding at least one new constraint either between X and Y or between Alice and Bob. Neither of these new constraints are desirable, as they introduce constraints between two previously not directly connected agents and thus present a threat to the privacy of the relevant agents.

As the recent technological advancements have allowed for solving larger problems that are highly interwoven and dependent on each other, efficiency and privacy have become critical requirements. To address this challenge, we propose a new approach to solve the MaSTP, which is based on *arc-consistency*.

A constraint R between two variables x, y is called arc-consistent (AC), if for every value d_x from the domain of x there is a value d_y in the domain of y such that $(d_x, d_y) \in R$. While AC is an important tool for solving finite (multiagent) constraint satisfaction problems (CSPs) [101, 6, 103, 52] at first glance it is not clear how it can be applied to solving CSPs with real domains such as the STP, because either the existing AC algorithms are fine-grained and work with each single element of a domain to enforce AC, which is impossible for real domains, or they are coarse-grained, but cannot guarantee their termination, as real domains can be infinitely refined when constraints are propagated.[1]

This chapter is organized as follows. Section 5.3 introduces some necessary definitions and notations. We provide the first AC-based approach for solving STP and analyze its computational complexity in section 5.4. We then provide the first AC-based approach for solving multiagent STP, which preserves the privacy of the agents, and analyze its computational complexity in section 5.5. Finally, we experimentally show that both our centralized and distributed algorithms outperform their existing counterparts for solving STP in section 5.6.

5.3 Preliminaries

This section briefly introduces the STP. Details can be found in [38].

The *simple temporal problem* (STP) is a constraint satisfaction problem where

[1] [38] for example, suggest discretizing the domains to overcome this issue, in which case the total number of contraint propagations would depend on the sizes of the domains. The performance of our AC algorithm for (multiagent) STP does not depend on the sizes of the domains.

each constraint is a set of linear inequalities of the form

$$a_{vw} \leq w - v \leq b_{vw}, \tag{5.1}$$

where a_{vw}, b_{vw} are constants and v, w are variables defined on a continuous domain representing time points. The constraint in (5.1) is abbreviated as $I_{vw} = [a_{vw}, b_{vw}]$. As (5.1) is equivalent to $-b_{vw} \leq v - w \leq -a_{vw}$, we also obtain $I_{wv} = I_{vw}^{-1} = [-b_{vw}, -a_{vw}]$. The *domain* of each variable v is an interval $I_v = [a_v, b_v]$, where I_v could be a singleton or empty. Assume that o is a special *auxiliary* variable that represents the fixed zero temporal point. Then the domain I_v can also be regarded as a constraint from o to v and $I_v = [a_v, b_v] = [a_{ov}, b_{ov}] = I_{ov}$.

Algebraic operations on STP constraints are defined as follows. The *intersection* of two STP constraints defined on variables v, w yields a new constraint over v, w that represents the conjunction of the constraints. It is defined as

$$I_{vw} \cap I'_{vw} := [\max\{a_{vw}, a'_{vw}\}, \min\{b_{vw}, b'_{vw}\}].$$

The *composition* of an STP constraint I_{vu} over variables v, u and another STP constraint I_{uw} over u, w yields a new STP constraint over v, w that is inferred from the other two constraints and is defined as $I_{vu} \otimes I_{uw} := [a_{vu} + a_{uw}, b_{vu} + b_{uw}]$. Here we require that $[a, b] \otimes \varnothing = \varnothing$ for any $a \leq b$.

Remark 5.1. *For STP constraints, the composition and intersection are associative and, as noted in [38], composition distributes over* non-empty *intersection for intervals, i.e., $I \otimes (J \cap K) = (I \otimes J) \cap (I \otimes K)$ for any three intervals I, J, K such that $J \cap K \neq \varnothing$.*

Definition 5.1. *An instance of STP is called a* simple temporal network *(STN) and is a tuple $\langle V, D, C \rangle$, where V is a finite set of variables, $D = \{I_v \mid v \in V\}$ is a*

set of intervals, and C is a set of STP constraints defined on V.

We assume that all variables in V appear in C and at *most* one constraint exists between any pair of variables v and w. Moreover, if $I_{vw} = [a, b]$ is the constraint in C from v to w, we always assume that the constraint $I_{wv} = I_{vw}^{-1} = [-b, -a]$ is also in C. As previously mentioned, the domain I_v of each variable v can be regarded as either a unary constraint, or a binary constraint $I_{ov} = I_v$, where o is a fixed variable representing the zero time point.

An STN naturally induces a graph in the following sense.

Definition 5.2. *The* constraint graph $G_{\mathcal{N}} = (V, E)$ *of an STN* $\mathcal{N} = \langle V, D, C\rangle$ *is an undirected graph, where the set E of edges consists of constrained unordered pairs of variables in C, i.e.,*

$$E = \{\{v, w\} \mid v, w \in V, v \neq w, I_{vw} \in C\}.$$

Let $G_{\mathcal{N}} = (V, E)$ be the constraint graph of an STN $\mathcal{N}$. We can use a labelled *directed* graph to illustrate $\mathcal{N}$, where for any undirected edge $\{v, w\} \in E$ there is exactly one directed edge (v, w) that is labelled with the corresponding interval $[a_{vw}, b_{vw}]$.

A *path* π from v to w in $G_{\mathcal{N}}$ is a sequence of variables $u_0, u_1, ..., u_k$ such that $v = u_0$, $w = u_k$, and $\{u_s, u_{s+1}\}$ is an edge in E for each $s = 0, \ldots, k-1$ (k is called the length of π). We write $\bigotimes \pi$ for the composition of all these $I_{u_s,u_{s+1}}$, i.e.,

$$\bigotimes \pi = I_{u_0,u_1} \otimes I_{u_1,u_2} \otimes ... \otimes I_{u_{k-1},u_k} \tag{5.2}$$

If $v = w$, then we call π a *cycle* at v. For a cycle π, let $[a, b] = \bigotimes \pi$. We call π a *negative cycle* if $b < 0$.

Definition 5.3. *A* solution *of an STN* $\mathcal{N} = \langle V, D, C\rangle$ *is an assignment, that assigns to each variable* $v \in V$ *a time point from* $I_v \in D$ *such that all constraints in* C *are satisfied.* $\mathcal{N}$ *is said to be* consistent *if* $\mathcal{N}$ *has a solution. Two STNs are said to be* equivalent *if they have the same solution set.*

Definition 5.4. *Let* $\mathcal{N} = \langle V, D, C\rangle$ *be a consistent STN and let* v *and* w *be variables in* V*. A constraint* I_{vw} *from* v *to* w *is said to be* minimal *if every assignment that assigns time points from domains* I_v *and* I_w *to* v *and* w*, respectively, and satisfies* I_{vw} *can be extended to a solution of* $\mathcal{N}$*. A domain* I_v *of* $v \in V$ *is said to be minimal if every assignment of a time point from* I_v *to* v *can be extended to a solution of* $\mathcal{N}$*. We say* $\mathcal{N}$ *is minimal if every constraint in* C *as well as every domain in* D *is minimal (note that, since we regard domains as constraints between the zero time point* o *and variables, we also require the domains to be minimal).*

5.4 Solving the STP with Arc-Consistency

In this section we show that enforcing arc-consistency is sufficient to solve the STP.

Definition 5.5. *Let* $\mathcal{N} = \langle V, D, C\rangle$ *be an STN. Suppose* v *and* w *are two variables in* V*,* I_v *and* I_w *are, respectively, their domains, and* I_{vw} *is a constraint in* C *from* v *to* w*. We say that* I_{vw} *is* arc-consistent *(AC) (relative to* I_v *and* I_w*) if for any* $t_v \in I_v$ *there exists some* $t_w \in I_w$ *such that* $t_w - t_v \in I_{vw}$*, i.e.,* $a_{vw} \leq t_w - t_v \leq b_{vw}$*. We say that* $\mathcal{N}$ *is AC if both* I_{vw} *and* I_{wv} *are AC for every constraint* $I_{vw} \in C$*.*

An STN $\mathcal{N}' = \langle V, D', C\rangle$ *with* $D' = \{I'_v \mid v \in V\}$ *is called the AC-*closure *of* $\mathcal{N}$*, if* $\mathcal{N}'$ *is the largest arc-consistent STN which is equivalent to* $\mathcal{N}$*, in the sense that for every other arc-consistent STN* $\mathcal{N}'' = \langle V, D'', C\rangle$ *with* $D'' = \{I''_i \mid v \in V\}$*, we have that* $I''_v \subseteq I'_v$ *for all* $v \in V$*.*

Lemma 5.1. *Let* $\mathcal{N} = \langle V, D, C\rangle$ *be an STN and* $v, w \in V$ *two variables that are*

constrained by I_{vw} in C. Then I_{vw} is arc-consistent relative to I_v and I_w iff $I_v \subseteq I_w \otimes I_{wv}$.

Proof. It suffices to show that

$$I_w \otimes I_{wv} = \{x \in \mathbb{R} \mid \exists y \in I_w \text{ s.t. } y - x \in I_{vw}\} \tag{5.3}$$

Let $I_v = [a, b]$, $I_w = [c, d]$ and $I_{vw} = [e, f]$. Then

$$\begin{aligned}
&\{x \in \mathbb{R} \mid \exists y \in I_w \text{ s.t. } y - x \in I_{vw}\} \\
&= \{x \in \mathbb{R} \mid \exists c \leq y \leq d \text{ s.t. } e \leq y - x \leq f\} \\
&= \{x \in \mathbb{R} \mid \exists c \leq y \leq d \text{ s.t. } y - f \leq x \leq y - e\} \\
&= \{x \in \mathbb{R} \mid c - f \leq x \leq d - e\} \\
&= [c, f] \otimes [-f, -e] = I_w \otimes I_{wv},
\end{aligned}$$

which proves Eq. (5.3). □

Lemma 5.2. *Let $\mathcal{N} = \langle V, D, C\rangle$ be an arc-consistent STN and $v, w \in V$ two variables that are constrained by I_{vw} in C. Then $I_v \subseteq I_w \otimes I_{wv}$.*

Proof. This follows directly from Lemma 5.1 and that I_{vw} is AC relative to I_v and I_w. □

The following result directly follows from Lemma 5.2.

Corollary 5.1. *Let $\mathcal{N} = \langle V, D, C\rangle$ be an arc-consistent STN. Let π be a path in $\mathcal{N}$ from w to v. Then $I_v \subseteq I_w \otimes \bigotimes \pi$.*

Lemma 5.3. *Let $\mathcal{N} = \langle V, D, C\rangle$ be an arc-consistent STN and v, w variables in V. If $\mathcal{N}$ is consistent, then $I_v \subseteq I_w \otimes I^m_{wv}$, where I^m_{wv} is the minimal constraint from w to v.*

Proof. Since $\mathcal{N}$ is consistent, I_{wv}^m is nonempty. Recall that I_{wv}^m is the intersection of the compositions along all paths in $\mathcal{N}$ from w to v (cf. [38, §3]) and composition distributes over non-empty intersection for intervals. The result follows directly from Corollary 5.1. ☐

Lemma 5.4 ([111])**.** *Suppose $\mathcal{N} = \langle V, D, C\rangle$ is an STN. Then $\mathcal{N}$ is inconsistent if and only if there exists a negative cycle.*

Lemma 5.5. *Given a consistent STN $\mathcal{N} = \langle V, D, C\rangle$ with $n = |V|$, for any path π of length $\geq n$ there is a path π' of length $< n$ such that $\bigotimes \pi' \subseteq \bigotimes \pi$.*

Proof. Since the length of π is $\geq n$, π must have a cycle at a variable v. As the cycle is not negative, removing the cycle and leaving only v in the path results in a path π' with $\bigotimes \pi' \subseteq \bigotimes \pi$. Repeating this procedure until there is no cycle gives the desired result. ☐

Lemma 5.6. *Let $\mathcal{N} = \langle V, D, C\rangle$ be an STN and $\mathcal{N}'$ its AC-closure. Then $\mathcal{N}$ is consistent iff $\mathcal{N}'$ has no empty domain.*

Proof. We prove $\mathcal{N}$ is inconsistent iff $\mathcal{N}'$ has an empty domain. As $\mathcal{N}$ and $\mathcal{N}'$ are equivalent, if $\mathcal{N}'$ has an empty domain, then $\mathcal{N}$ is inconsistent.

Now suppose $\mathcal{N}$ is inconsistent. Then by Lemma 5.4, there exists a negative cycle π in $\mathcal{N}$ at some w such that $\bigotimes \pi = [l, h]$ with $h < 0$. Now let v be a variable in $\mathcal{N}$ with $I_{wv} = [e, f]$ and let $I'_v = [a, b], I'_w = [c, d]$ be the domains of v and w in $\mathcal{N}'$, respectively. Choose $k \in \mathbb{N}$ sufficiently large, such that $kh < b - d - f$. Then, by Lemma 5.3 we have

$$
\begin{aligned}
I'_v &\subseteq I'_w \otimes \left(\bigotimes \pi^k \otimes I_{wv}\right) \qquad (5.4) \\
&= [c, d] \otimes ([kl, kh] \otimes [e, f]) \\
&= [c + kl + e, d + kh + f],
\end{aligned}
$$

where π^k is the concatenation of k copies of path π. Because $kh < b - d - f$, (5.4) is possible only if I'_v is empty. $\square$

Theorem 5.1. *Let $\mathcal{N} = \langle V, D, C\rangle$ be a consistent STN and $\mathcal{N}'$ its AC-closure. Then all domains in $\mathcal{N}'$ are minimal.*

Proof. If the constraint graph $G_{\mathcal{N}}$ is connected, i.e., for any two variables v, w, there is a path in $G_{\mathcal{N}}$ that connects v to w, then we may replace the constraint from v to w with the nonempty minimal constraint I^m_{vw} (or add I^m_{vw}, if there was no constraint between v and w). We write the refined STN as $\mathcal{N}^*$. For any two variables v, w, by Lemma 5.3, I_v is contained in $I_w \otimes I^m_{wv}$ and I_w is contained in $I_v \otimes I^m_{vw}$. This shows that $\mathcal{N}^*$ is the same as the minimal STN of $\mathcal{N}$, and thus, establishes the minimality of each I_v.

In case the constraint graph is disconnected, we consider the restriction of $\mathcal{N}$ to its connected components instead. The same result applies. $\square$

Two special solutions can be constructed if $\mathcal{N}$ is arc-consistent and has no empty domain.

Proposition 5.1. *Let $\mathcal{N} = \langle V, D, C\rangle$ be an arc-consistent STN with $D = \{I_v \mid v \in V\}$ and $I_v = [a_v, b_v]$ for each v. If no I_v is empty, then the assignments $A = \{a_v \mid v \in V\}$ and $B = \{b_v \mid v \in V\}$ are two solutions of $\mathcal{N}$.*

Proof. Let $\mathcal{N}' = \langle V, D', C'\rangle$ be the minimal STN of $\mathcal{N}$. By Theorem 5.1, we have $D' = D$ and $\mathcal{N}'$ is equivalent to $\mathcal{N}$. The above claim follows as the assignments $A = \{a_v \mid v \in V\}$ and $B = \{b_v \mid v \in V\}$ are two solutions of the minimal STN $\mathcal{N}'$ (cf. [37, Corollary 3.2]). $\square$

Theorem 5.2. *Enforcing AC is sufficient to solve STP.*

Proof. Let $\mathcal{N}$ be an STN and $\mathcal{N}'$ its AC-closure. If $\mathcal{N}'$ has an empty domain, then $\mathcal{N}$ has no solution by Lemma 6.1. If $\mathcal{N}'$ does not have an empty domain, then we can use Proposition 5.1 to find a solution. □

Remark 5.2. *(i) As solving an STN is equivalent to solving a system of linear inequalities, the solution set of an STN is a convex polyhedron. Thus any convex combination of the two solutions A and B is again a solution of the STN. (ii) Enforcing AC can in essence find all solutions of an STN: Suppose $\mathcal{N}$ is arc-consistent and has no empty domain. We pick an arbitrary variable v that has not been instantiated yet, then assign any value from D_v to v, and enforce AC on the resulting network. We repeat this process until all variables are instantiated. (iii) Proposition 5.1 can also be obtained by first showing that STP constraints are both max/min-closed, and then using the result in [63, Thm 4.2], which states that the AC-closure of a constraint network over max/min-closed constraints have the maximal and the minimal values of the domains as two solutions. As a consequence of this, Theorem 5.1 can also be obtained, because the solution set of an STN is convex (cf. Remark 5.2 (i)).*

5.4.1 A Centralized AC Algorithm for the STP

In this section we propose an AC algorithm, called ACSTP, to solve STNs. The algorithm is presented as Algorithm 5.1.

Theorem 5.3. *Given an input STN $\mathcal{N}$, Algorithm 5.1 returns "inconsistent" if $\mathcal{N}$ is inconsistent. Otherwise, it returns the AC-closure of $\mathcal{N}$.*

Proof. We first note that intersection and composition of constraints do not change the solution set of the input STN $\mathcal{N}$. This has two implications: First, if a domain I_v becomes empty during the process of the algorithm, then the solution set of $\mathcal{N}$ is empty and $\mathcal{N}$ is inconsistent. Second, if the algorithm terminates and its output $\mathcal{N}'$ is AC, then $\mathcal{N}'$ is the AC-closure of $\mathcal{N}$. Consequently, it suffices to show that if the algorithm terminates and returns $\mathcal{N}'$, then $\mathcal{N}'$ is AC.

Algorithm 5.1: ACSTP

Input : An STN $\mathcal{N} = \langle V, D, C \rangle$ and its constraint graph $G = (V, E)$, where $|V| = n$.
Output: An equivalent network that is AC, or "inconsistent".

```
1  Q ← ∅
2  for k ← 1 to n do
3      foreach v ∈ V do
4          I'_v ← I_v
5          foreach w ∈ V s.t. {v, w} ∈ E do
6              I_v ← I_v ∩ I_w ⊗ I_wv
7          if I_v = ∅ then return "inconsistent"
8          if I'_v = I_v then Q ← Q ∪ {v}
9          else Q ← Q \ {v}
10     if #Q = n then return N
11 return "inconsistent"
```

We first consider the case, where the algorithm returns $\mathcal{N}'$ in line 21 at the kth iteration of the for-loop (lines 2–10) for some $1 \leq k \leq n$. We show that $\mathcal{N}'$ is AC. Let I_v^k be the domain of v obtained after the kth iteration of the for-loop. Due to lines 6 and 8, we have for all $\{v, w\} \in E$ that $I_v^k \subseteq I_v^{k-1} \cap (I_w^{k-1} \otimes I_{wv})$ and $I_w^{k-1} = I_w^k$. Thus we have for $\{v, w\} \in E$ that $I_v^k \subseteq I_w^k \otimes I_{wv}$, which is by Lemma 5.1 equivalent to saying that I_{vw} is AC w.r.t. domains I_v^k and I_w^k. Hence, the output $\mathcal{N}'$ is AC.

Now suppose that the algorithm exited in line 11 returning "inconsistent". Thus, at the nth iteration of the for-loop we have $\#Q < n$ in line 21. We prove that $\mathcal{N}$ is inconsistent by contradiction. Assume that $\mathcal{N}$ is consistent. For any $v \in V$ and any $k \geq 1$, we write Π_v^k for the set of paths from o (the auxiliary variable denoting the zero time point) to v with length $\leq k$ in the constraint graph of $\mathcal{N}$. We claim

$$I_v^{k-1} \subseteq \bigcap_{\pi \in \Pi_v^k} \bigotimes \pi \tag{5.5}$$

for any $k \geq 1$. Then, with I_v^m being the minimal domain of v, we have

$$I_v^m \subseteq I_v^{n-1} \subseteq \bigcap_{\pi \in \Pi_v^n} \bigotimes \pi = I_v^m,$$

because I_v^m is the intersection of the compositions along all paths in $\mathcal{N}$ from o to v (cf. [38, §3]), where it suffices to only build compositions along paths of length $\leq n$ by Lemma 5.5. Thus $I_v^n = I_v^{n-1} = I_v^m$ for all $v \in V$, which is a contradiction to our assumption that at the nth iteration of the for-loop we have $\#Q < n$ in line 21.

We now prove (5.5) by using induction on k. First, for $k = 1$, since Π_v^1 contains only one path of length 1 (i.e., the edge $\{o, v\}$), we have $I_v^0 = I_v = \bigcap_{\pi \in \Pi_v^1} \bigotimes \pi$. Now suppose (5.5) is true for $k - 2$ for all $w \in V$. Then by line 6 and our induction hypothesis we have

$$\begin{aligned}
I_v^{k-1} &\subseteq I_v^{k-2} \cap \left(\bigcap_w I_w^{k-2} \otimes I_{wv} \right) \\
&\subseteq I_v^{k-2} \cap \left(\bigcap_w \left(\bigcap_{\pi \in \Pi_w^{k-1}} \bigotimes \pi \right) \otimes I_{wv} \right) \\
&\subseteq \left(\bigcap_{\pi \in \Pi_v^{k-1}} \bigotimes \pi \right) \cap \left(\bigcap_{(\pi \in \Pi_v^k) \wedge (|\pi| \geq 2)} \bigotimes \pi \right) \\
&= \bigcap_{\pi \in \Pi_v^k} \bigotimes \pi,
\end{aligned}$$

which proves (5.5). □

Theorem 5.4. *Algorithm 5.1 runs in time $O(en)$, where e is the number of edges of the constraint graph of the input STN and n is the number of variables.*

Proof. There are at most n iterations of the for-loop and each iteration involves $O(e)$ operations. □

Remark 5.3. *Algorithm 5.1 can also be understood as computing the shortest path from a source vertex o to every other vertex v and the shortest path from every other vertex v to the source vertex o. This can be realized in time $O(en)$ by using a shortest path tree algorithm with negative cycle detection (cf. [118, Section 7.2] and [70, Section 7.1].*

5.5 Solving the MaSTP with Arc-Consistency

In this section we extend ACSTP to a distributed algorithm DisACSTP to solve multiagent simple temporal networks (MaSTNs).

A *multiagent simple temporal network (MaSTN)* [14] is defined similarly as Definition 1.3.

Constraint graphs for MaSTNs can be defined analogously as that for STNs, where we use E^{X} for the set of edges corresponding to constraints in C^X. See Figure 5.1 for an illustration. In Figure 5.1, the edges in E^{X} are represented as red lines.

Definition 5.6. *Suppose $\mathcal{M} = \langle \mathcal{P}, C^X \rangle$ is an MaSTN. Let $I_{vw} \in C^X$ with $v \in V_i, w \in V_j$ be an external constraint. We say that I_{vw} is an external constraint of agent i, and write C_i^X for the set of external constraints of agent i. We call v and w a* shared *and an* external *variable of agent i, respectively. We write V_i^X for the set of external variables of agent i. In Figure 5.1, the vertices for shared variables are represented as red circles.*

DisACSTP is presented in Algorithm 5.2. In DisACSTP each agent i gets as input its portion $\mathcal{N}_i$ of the input MaSTN $\mathcal{M}$ and the set C_i^{X} of its external constraints, and runs its own algorithm. Similar to ACSTP, DisACSTP updates the domains of $\mathcal{N}_i$ at each iteration of the for-loop and maintains a queue Q_i to record the information about the unchanged domains. When a domain becomes empty during the updates, then the agent can terminate the algorithm and conclude that the

Algorithm 5.2: DisACSTP

Input : $\mathcal{N}_i$: agent i's portion of MaSTN $\mathcal{M}$;
V_i^{X}: the set of agent i's external variables;
C_i^{X}: the set of agent i's external constraints;
$parent(i)$: the parent of agent i w.r.t. $T(\mathcal{M})$;
$children(i)$: the children of agent i w.r.t. $T(\mathcal{M})$;
n: the number of variables of $\mathcal{M}$.

Output: Agent i's portion of the AC-closure of $\mathcal{M}$ or "inconsistent".

```
1  Q_i ← ∅
2  for k ← 1 to n do
3      Send the domains of the shared variables to the neighbors.
4      Receive the domains of the external variables from the neighbors.
5      foreach v ∈ V_i do
6          I'_v ← I_v
7          foreach w ∈ V_i ∪ V_i^X s.t. {v, w} ∈ E_i ∪ E_i^X do
8              I_v ← I_v ∩ I_w ⊗ I_wv
9          if I_v = ∅ then
10             Broadcast "inconsistent".
11             return "inconsistent"
12         if I'_v = I_v then  Q_i ← Q_i ∪ {v}
13         else  Q_i ← Q_i \ {v}
14     if #Q_i = #v then
15         if root(i) then
16             Send inquiry ("Are all Q_i full?", k) to children(i)
17         while true do
18             m ← ReceiveMessage()
19             if m is domains of external variables from a neighbor then
20                 break
21             if m is inquiry ("Are all Q_i full?", k) then
22                 if leaf(i) then
23                     Send feedback ("yes", k) to parent(i)
24                 else Send m to children(i)
25             if m is feedback ("yes", k) then
26                 if all feedbacks received from children(i) then
27                     if root(i) then
28                         Broadcast "arc-consistent"
29                         return N_i
30                     else Send m to parent(i)
31             if m is "arc-consistent" then
32                 return N_i
33             if m is "inconsistent" then
34                 return "inconsistent"
35 return "inconsistent"
```

input MaSTN $\mathcal{M}$ is inconsistent. There are however aspects in DisACSTP that are different from ACSTP, which stem from the fact that in MaSTP an agent cannot have the global knowledge of the states of other agents' processes without sharing certain information with other agents. These aspects are the following:

1. The total number n of the variables in the input MaSTN is initially not known to individual agents. This, however, can easily be determined using an echo algorithm [22]. We can therefore regard n as given as an input to DisACSTP.

2. As the agents may run their processes at different paces, at each iteration of the for-loop (lines 2–34), they synchronize the domains of their external variables (lines 3–4). Otherwise, some agents might use stale external domains and make wrong conclusions.

3. When a domain becomes empty while running DisACSTP, an agent broadcasts (lines 9–11) this information to other agents so that they can terminate their algorithms as soon as possible.

4. If the queue Q_i of an agent i is full (i.e., it contains all of the agent's variables in V_i) after an iteration of the for-loop, then the agent shares this information with all other agents in $\mathcal{M}$ so as to jointly determine whether the queues of all agents are full and the network is arc-consistent (lines 15–16 and 21–30).

5. If the queue Q of an agent is *not* full after an iteration of the for-loop, then the agent broadcasts this information to all other agents, so that they can move to the next iteration of the for-loop as soon as possible.

All the preceding aspects are subject to communication of certain information between agents. DisACSTP coordinates this communication while (i) preserving the privacy of each agent and (ii) reducing the duration of any idle state of an individual agent. Concretely:

- Each agent shares information only with the agents who are connected through an external constraint. We call them the *neighbors* of the agent. This neighborhood-relationship among the agents induces a graph that we call henceforth the *agent graph*.

- Each agent shares with its neighbors *only* the domains of its shared variables. No other information is shared (such as its network structure, constraints, private variables and their domains) and only the neighbors w.r.t. the agent graph can share the information. This property is a critical advantage over D△PPC [14], as D△PPC often creates new external constraints during the process and reveal more private information of the agents than necessary.

- Each agent uses a *broadcasting* mechanism to share global properties of the input MaSTN, i.e., an agent first sends a message (e.g., "inconsistent") to its neighbors, then the neighbors forward the message to their neighbors and so on, until all agents receive the message. To reduce the number of messages, duplicates are ignored by the agents.

  An agent i broadcasts the following messages: "arc-consistent", "inconsistent" and "Q_i is not full", where the last message is indirectly broadcasted by agent i skipping lines 14–34 and moving to the next iteration of the for-loop and then sending its shared domains to its neighbors. This initiates a chain reaction among the idle neighbors of agent i who have not moved to the next iteration yet, as they quit the idle states (lines 19–20) and move to the next iteration of the for-loop and then send also their shared domains to their idle neighbors (lines 3–4).

- There is a dedicated agent who checks at each iteration of its for-loop (given its queue is full) whether the queues of all other agents are full at the same iteration. This dedicated agent is determined by building a minimal spanning

tree (e.g., by using an echo algorithm [22]) $T(\mathcal{M})$ of the agent graph. The agent who is the root (henceforth the *root agent*) of this tree becomes then the dedicated agent.

The root agent sends an inquiry to its children to check whether the queues of all its descendants are full (lines 15–16). The inquiry is then successively forwarded by the descendants whose queues are full. We have to distinguish here between two cases:

(1) If all descendants' queues are *full*, then the inquiry reaches all the leaf agents and returns back as feedbacks (lines 22–23) until the root agent receives all the feedbacks (lines 25–30) and broadcasts "arc-consistency".

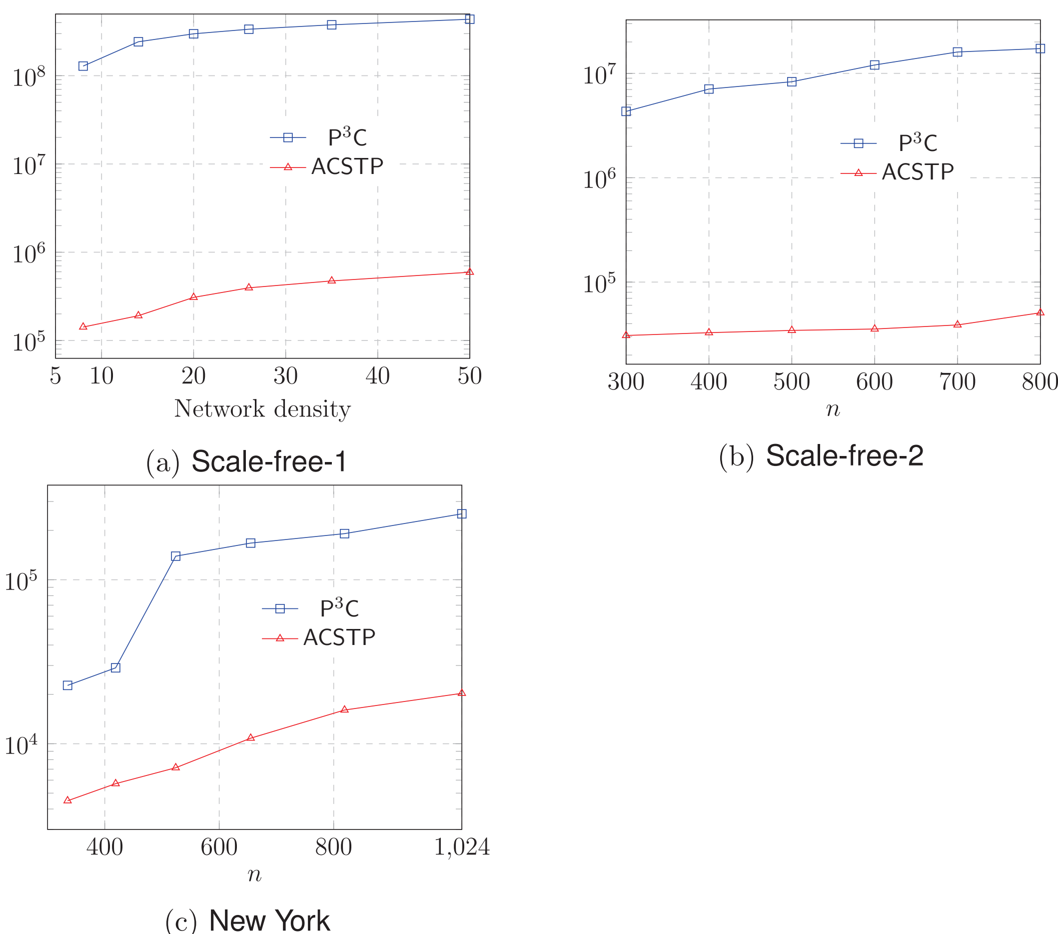
(2) If a descendant's queue is *not full*, then the descendant moves on to the next iteration of the for-loop and initiates a chain reaction among other agents by sending the domains of its shared variables to its neighbors (cf. the second paragraph of the third bullet point).

Due to the properties so far considered, DisACSTP is guaranteed to simulate the behavior of ACSTP while allowing concurrent domain update operations.

Theorem 5.5. *Let $\mathcal{M} = \langle \mathcal{P}, C^X \rangle$ be an MaSTN. Let $\mathcal{N}_{\max}$ be a network with $e_{\max} = \max\{e_i + e_i^{\mathrm{X}} \mid 1 \leq i \leq p\}$, where e_i and e_i^{X} are the number of edges of the constraint graph of $\mathcal{N}_i$ and the number of external constraints of agent i, respectively. Then Algorithm 5.2 enforces AC on $\mathcal{M}$ in time $O(e_{\max} n)$.*

5.6 Evaluation

In this section we experimentally compare our algorithms against the state-of-the-art algorithms for solving STNs. For centralized algorithms, we compare our ACSTP algorithm against P^{3}C algorithm [104]; for distributed algorithms, we compare our DisACSTP algorithm against D$\triangle$PPC algorithm [14]. All experiments for



(a) Scale-free-1

(b) Scale-free-2

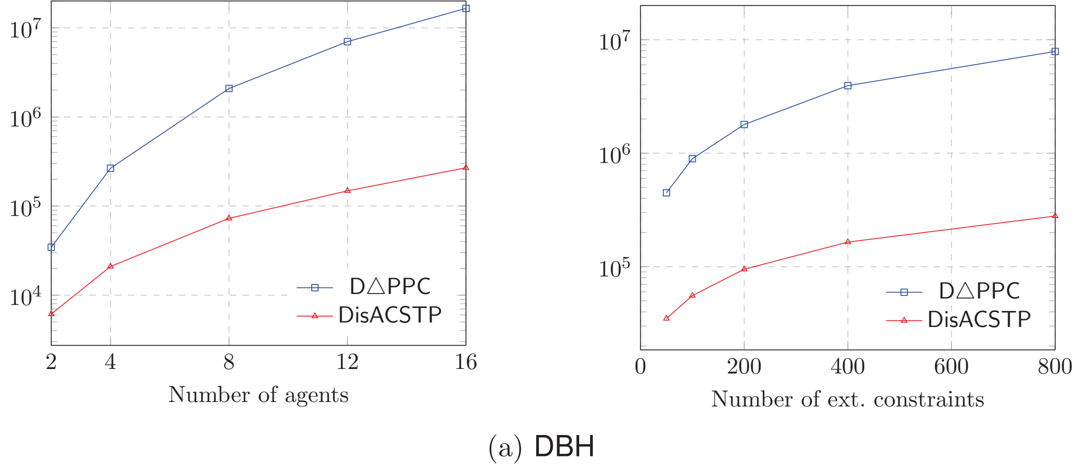
(c) New York

Figure 5.2 : Evaluation of ACSTP and P^{3}C. The y-axes (on the log scale) represent the number constraint checks.

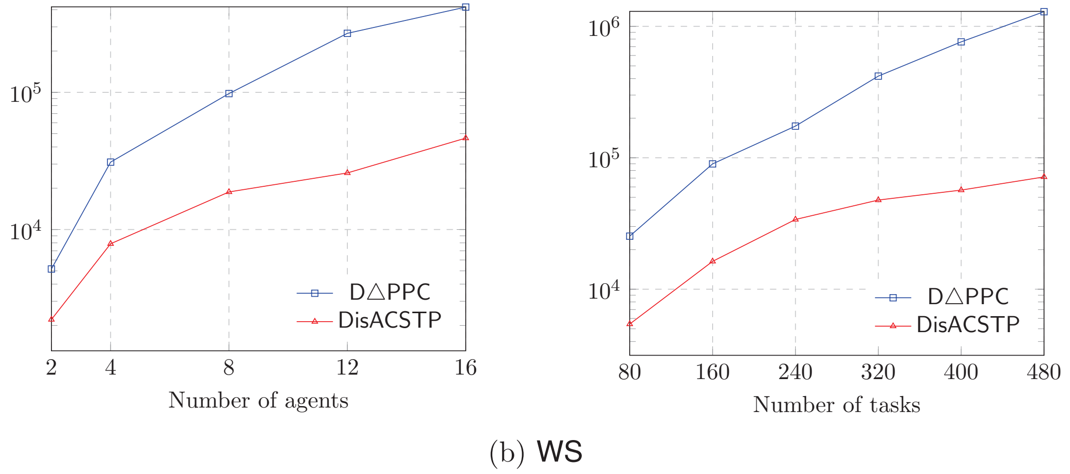
distributed algorithms used an asynchronous simulator in which agents are simulated by processes which communicate only through message passing and default communication latency is assumed to be zero. Our experiments were implemented in Python 3.6 and carried out on a computer with an Intel Core i5 processor with a 2.9 GHz frequency per CPU, 8 GB memory [1].

As measures for comaring performances we use the number of constraint checks and the number of non-concurrent constraint checks (NCCCs) performed by the centralized algorithms and the distributed algorithms, respectively. Given an STN

[1]The source code for our evaluation can be found in https://github.com/sharingcodes/MaSTN



(a) DBH



(b) WS

Figure 5.3 : Evaluation of DisACSTP and D△PPC. The y-axis (on the log scale) represent the number of NCCCs.

$\mathcal{N} = \langle V, D, C \rangle$, a constraint check is performed when we compute relation $r \leftarrow I_{vw} \cap (I_{vu} \otimes I_{uw})$ and check if $r = I_{vw}$ or $r \not\subseteq I_{vw}$.

5.6.1 ACSTP vs. P^{3}C

Datasets

We selected instances from the benchmark datasets of STNs used in [106] for evaluations. We considered the scale-free graphs (Scale-free-1) with 1000 vertices and density parameter varying from 2 to 50. We also considered the scale-free graphs (Scale-free-2) with varying vertex count. The scale-free density parameter for this

selection is 5. Beside these artificially constructed graphs, we also considered graphs that are based on the road network of New York City (New York). This dataset contains 170 graphs on 108–3906 vertices, 113–6422 edges.

Results

The results are presented in Figure 5.2, where base-10 log scales are used for the y-axes. For the scale-free graphs we observe that ACSTP is 100–1000 times faster than P^{3}C. The dataset New York only contains very sparse networks (each network's density is less than 1%), thus both algorithms could easily solve these networks. However, we still observe that ACSTP is about 5–12 times faster than P^{3}C.

5.6.2 DisACSTP vs. D△PPC

Datasets

We selected instances from the benchmark datasets of MaSTNs used in [14] for evaluations. The first dataset BDH was randomly generated using the multiagent adaptation of Hunsberger's [60] random STN generator. Each MaSTN has N agents each with start time points and end time points for 10 activities, which are subject to various local constraints. In addition, each MaSTN has X external contraints. We evaluated the algorithms by varying the number of agents ($N \in \{2, 4, 8, 12, 16\}, X = 50 \times (N-1)$) and the total number of external constraints ($N = 16, X \in \{100, 200, 400, 800\}$).

The second dataset WS is derived from a multiagent factory scheduling domain [126], where N agents are working together to complete T tasks in a manufacturing environment. We evaluated algorithms by varying the number of agents ($N \in \{2, 4, 8, 12, 16\}, T = 20 \times N$) and the total number of tasks ($N = 16, T \in \{80, 160, 240, 320, 400, 480\}$).

Results

The results are presented in Figure 5.3, where base-10 log scales are again used for the y-axes. For the DBH random networks (Figure 5.3a) we observe that DisACSTP is 5–30 times faster than D△PPC. For the WS scheduling networks (Figure 5.3b) DisACSTP is 2–10 times faster than D△PPC. For both datasets we observe that, with increasing x-values, the y-values (i.e., NCCCs) for DisACSTP grow slower than those for D△PPC.

5.7 Conclusion

In this chapter we presented a novel AC-based approach for solving the STP and the MaSTP. We have shown that arc-consistency is sufficient for solving an STN. Considering that STNs are defined over infinite domains, this result is rather surprising. Our empirical evaluations showed that the AC-based algorithms are significantly more efficient than their PC-based counterparts. This is mainly due to the fact that PC-based algorithms add many redundant constraints in the process of triangulation. More importantly, since our AC-based approach does not impose new constraints between agents that are previously not directly connected, it respects as much privacy of these agents as possible. We should note here that even though our distributed algorithm DisACSTP showed remarkable performance, it can be further fine-tuned by using different termination detection mechanisms (cf. [96] and [107, Ch. 14]).

It would be interesting to see how the result in this chapter can be used for solving the general disjunctive temporal problems [115]. Potential extensions of this chapter also include adapting our AC algorithms to incremental algorithms for the STP [105], dynamic situations [102] and uncertainty [120].

Chapter 6

A New Distributed Generalized AC Algorithm

6.1 Contribution

Generalized arc-consistency propagation is predominantly used in constraint solvers to efficiently prune the search space when solving constraint satisfaction problems. Although many practical applications can be modelled as distributed constraint satisfaction problems, no distributed arc-consistency algorithms so far have considered the privacy of individual agents.

In this chapter, we propose a new distributed arc-consistency algorithm, called DisAC3.1, which leaks less private information of agents than existing distributed arc-consistency algorithms. In particular, DisAC3.1 uses a novel termination determination mechanism, which allows the agents to share domains, constraints and communication addresses only with relevant agents. We further extend DisAC3.1 to DisGAC3.1, which is the first distributed algorithm that enforces generalized arc-consistency on k-ary ($k \geq 2$) constraint satisfaction problems. Theoretical analyses show that our algorithms are efficient in both time and space. Experiments also demonstrate that DisAC3.1 outperforms the state-of-the-art distributed arc-consistency algorithm and that DisGAC3.1's performance scales linearly in the number of agents.

6.2 Introduction and Chapter Outline

Since CSP is an NP-complete problem in general, local consistency techniques are often used to prune the search space before or during the search for a solution.

Among those local consistency techniques, *arc-consistency* (AC) is the most studied and used pruning method for solving binary CSPs [8, 12, 89, 100]. AC has been generalized to *generalized arc-consistency* (GAC) for k-ary ($k \geq 2$) CSPs [100].

There are several distributed AC algorithms proposed in the literature, including DisAC3 [6], DisAC4 [103] and DisAC6 [6], which are, respectively, the distributed versions of AC3 [89], AC4 [103] and AC6 [8]. Another distributed algorithm DisAC9 [52], which is also a distributed version of AC6, is currently the state-of-the-art.

Although privacy is one main motivation and a major concern of solving distributed constraint satisfaction problems (DisCSPs) [44, 50, 124, 131], no distributed AC algorithms so far have considered the *communication address*[1] privacy of individual agents. Indeed, the distributed AC algorithms mentioned above either assume a complete agent communication graph, which reveals the *communication address*, thus the *identity*, of every agent, or broadcast deleted values of variable domains, revealing the existence of variables and their domains.

More precisely, the termination procedure of DisAC3, DisAC4, DisAC6 and DisAC9 assumes that the agent communication graph is complete[2], i.e., any two agents know the communication address of each other, which implies that they know the existence of each other and can directly send messages to each other. Also, whenever an agent deletes a value from one of its local domains, the agent broadcasts this information to all other agents immediately. This setting has the following drawbacks: (i) the algorithm may need to send unnecessarily many messages; (ii) the identities of agents and deleted domain values are revealed to irrelevant agents.

In this chapter we propose a new distributed algorithm for enforcing AC and the *first* distributed algorithm for enforcing GAC on DisCSPs, which allows the agents to

[1] The address of an agent, as defined in [128], represents the unique identity of the agent in the agent communication graph.

[2] Although not explicitly stated in [52], DisAC9 implicitly assumes a complete agent communication graph, as it uses the distributed snapshot algorithm [19, 107].

share domains, constraints and communication addresses only with relevant agents. It is worthwhile to note that the aspect of parallel processing (cf. [32, 51, 108, 132]) is not considered in this chapter, as it requires global knowledge about the input problem, which is often not available when the knowledge about the problem (i.e., domains and constraints) is distributed among autonomous agents. Because of privacy reasons, collecting all such knowledge from the individual agents is undesirable or impossible [128, 129]. Also we assume that each agent owns a local CSP including variables and domains, and not just a single constraint as discussed in [54, 55].

In this chapter we first propose a new distributed AC algorithm, called DisAC3.1, which is based on the optimal AC algorithm AC2001/3.1 [12] and avoids the aforementioned issues. DisAC3.1 uses a novel termination determination mechanism, where a dedicated agent determines the termination of the algorithm by comparing the timestamps of each message that the sender and the recipient report separately. This termination mechanism does not require a complete agent communication graph, as agents send messages to relevant agents only when necessary. As a result, DisAC3.1 does not reveal more information of each agent than necessary.

Moreover, DisAC3.1 has a low time complexity $O(ed^2)$ and a low space complexity $O(ed)$, where e is the number of edges and d is the largest domain size in the constraint graph of the input DisCSP. Moreover, our experiments also show that DisAC3.1 outperforms the state-of-the-art distributed arc-consistency algorithm DisAC9.

Furthermore, we extend algorithm DisAC3.1 to the *first* distributed algorithm, called DisGAC3.1, that enforces generalized arc-consistency on k-ary ($k \geq 2$) CSPs. Experiments show that the performance of DisGAC3.1 scales linearly in the number of agents.

The remainder of the chapter is organized as follows. After a short introduction

Algorithm 6.1: AC2001/3.1

Input : A binary CSP $\mathcal{N} = \langle V, D, C \rangle$.
Output: The AC-closure of $\mathcal{N}$, or "inconsistent".

```
1 Q ← {(v, w) | R_vw ∈ C, v ≠ w}
2 while Q ≠ ∅ do
3     (v, w) ← Q.POP()
4     if REVISE(v, w) then
5         if D_v = ∅ then
6             return "inconsistent"
7         Q ← Q ∪ {(u, v) | R_uv ∈ C, u ≠ v}
8 return D
```

of necessary background knowledge in Section 2, we describe in Section 6.4 our new distributed AC algorithm DisAC3.1 and, in Section 4 give theoretical analyses of DisAC3.1. Then, we extend DisAC3.1 to the first distributed GAC algorithm DisGAC3.1 in Section 6.6, and evaluate our algorithms empirically in Section 6.7. The last section concludes the chapter.

6.3 Preliminaries

Basic notations and results about CSP, DisCSP, AC and GAC can be found in section 1.3.

Several AC algorithms for building the AC-closures of binary CSPs have been proposed in the past decades [8, 12, 89, 100, 101]. In this chapter, we will extend AC2001/3.1 [12], which is known to be optimal, to a distributed algorithm.

The AC2001/3.1 algorithm, presented in Algorithm 6.1, is similar to the classical AC3 algorithm [89] with the following differences in the REVISE function: in AC2001/3.1 we assume that for any domain D_v, there is an arbitrary ordering imposed on values in D_v. For each constraint R_{vw} and for each value $a \in D_v$, the *smallest support* of a on R_{vw} is stored in $SS_{vw}(a)$. Therefore, if the smallest support

Function REVISE(v, w)

```
1  revised ← false
2  foreach a ∈ D_v do
3  |   b ← SS_vw(a)
4  |   if b ∉ D_w then
5  |   |   b ← NextValue(b, D_w)
6  |   |   while b ≠ NIL and (b ∉ D_w or (a, b) ∉ R_vw) do
7  |   |   |   b ← NextValue(b, D_w)
8  |   |   if b ≠ NIL then
9  |   |   |   SS_vw(a) ← b
10 |   |   else
11 |   |   |   delete a from D_v
12 |   |   |   revised ← true
13 return revised
```

of a, say b, is deleted, then the algorithm only needs to search for the next smallest support $b' > b$ of a and does not need to search the whole domain every time as is the case of AC3. For example, suppose that the binary CSP in Figure 1.1 is an input to AC2001/3.1, then we have that $SS_{21}(f) = b$. If b is removed from D_1 and the ordering is $a < b < g$, AC2001/3.1 searches a new support for f by starting from g instead of from a. It turns out that introducing "smallest support" makes AC2001/3.1 an optimal AC algorithm.

Theorem 6.1 ([12]). AC2001/3.1 *has the optimal worst case time complexity* $O(ed^2)$ *with space complexity* $O(ed)$, *where* e *is the number of edges in the constraint graph of the input binary CSP and* d *is the largest domain size.*

6.4 A New Distributed Arc-Consistency Algorithm

In this section we extend AC2001/3.1 to a new distributed AC algorithm DisAC3.1. In the distributed setting, agents pass messages to communicate with each other. We follow the communication model of [128]:

- Agents communicate by sending messages. Agent i can send messages to agent j iff agent i knows the *communication address* of agent j. Agent i calls function Send(i, j, *msgType*, *msgContent*) to send a message with content *msgContent* and of type *msgType* to agent j.

- For each agent i, there is a message queue associated with it. Messages sent to agent i are stored in the queue. Agent i calls function Receive() to receive all messages stored in the queue. Once Receive() is called, agent i waits until at least one message is received.

- The delay of delivering a message is finite. For the transmission between any pair of agents, messages are received in the order in which they were sent.

Note that this model does not require a physical communication graph to be fully connected (i.e., a complete graph). This model only assumes the existence of a reliable underlying communication structure and is independent of the implementation of the physical communication graph.

Definition 6.1 (agent communication graph)**.** *Given a distributed binary CSP* $\mathcal{M} = \langle \mathcal{P}, C^X \rangle$*, an* agent communication graph *of* $\mathcal{M}$ *is a pair* (A, E_A)*, where* $A = \{1, 2, ..., p\}$ *is the set of agents in* $\mathcal{M}$*, and* E_A *is a set of undirected edges over* A*:* $e_{ij} = \{i, j\} \in E_A$ *iff agents* i *and* j *know the communication address of each other. An agent communication graph is called* standard *if it further satisfies the condition that agents* i *and* j *know the communication address of each other iff they share at least one external constraint.*

In this chapter, we assume that a distributed binary CSP cannot be split into two or more disjoint distributed binary CSPs; in other words, the standard agent communication graph of a distributed binary CSP is connected. The standard agent communication graph of the distributed binary CSP in Figure 1.2 is given in Figure 6.1.

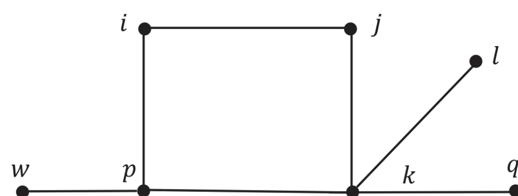


Figure 6.1 : The standard agent communication graph of the distributed binary CSP in Figure 1.2.

In order to handle termination of our distributed AC algorithm, we select a *dedicated* agent who decides when to terminate, and allow other agents to communicate with the dedicated agent through a path in the standard agent communication graph. To this end, we build a spanning tree of the standard agent communication graph by using the *echo algorithm* [22]. The root of the spanning tree becomes then the dedicated agent, called the *root agent*. In the echo algorithm, no agent knows the configuration or extent of the communication graph or the addresses of non-neighboring agents, and thus, privacy is not violated. The algorithm runs in $O(\hat{D})$ time and $O(p)$ space, and sends $O(\hat{e})$ messages, where $\hat{D}$, p and $\hat{e}$ are the diameter, number of nodes and number of edges of the standard agent communication graph, respectively.

Given the standard agent communication graph G of a distributed binary CSP $\mathcal{M}$, we call the spanning tree obtained by the echo algorithm the *termination tree* or the *T-tree* of G. We can build a T-tree (cf. Figure 6.2) of the standard agent communication graph in Figure 6.1. An agent can send messages to the root agent via its path to the root agent, where agents in the middle of the path need to help forwarding messages. For example, in Figure 6.2 agent k can send messages to agent i via the path (k, p, i).

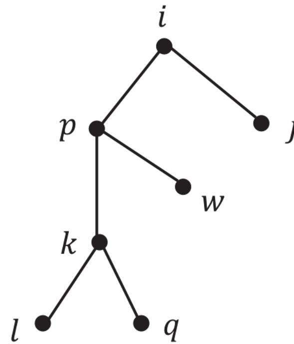


Figure 6.2 : A T-tree for the standard agent communication graph in Figure 6.1.

6.4.1 The Algorithm

The new distributed AC algorithm is presented as Algorithm 6.2. Given a distributed binary CSP $\mathcal{M} = \langle \mathcal{P}, C^X \rangle$, each agent i takes its portion of $\mathcal{M}$ as an input and runs its own copy of Algorithm 6.2 separately and concurrently. Agent i's portion of $\mathcal{M}$ includes:

- $\mathcal{N}_i = \langle V_i, \mathcal{D}_i, C_i \rangle$.
- For each i's neighbor agent j we have as inputs:
  - a set C_{ij} of all external constraints between i and j,
  - a set V_j^i of all variables of agent j that are related to i through an external constraint in C_{ij}, and
  - a set $\mathcal{D}_j^i$ of domains D_w^i for each $w \in V_j^i$, where each $D_w^i \in \mathcal{D}_j^i$ is a copy of agent j's domain D_w.

Example 6.1. *In Figure 1.2 agent i's portion of $\mathcal{M}$ is as follows:*

- $\mathcal{N}_i = \langle V_i = \{v_1, v_2\}, \mathcal{D}_i = \{D_{v_1}, D_{v_2}\}, C_i = \{R_{12}\} \rangle$,
- $V_j^i = \{v_3, v_4\}$, $\mathcal{D}_j^i = \{D_{v_3}^i, D_{v_4}^i\}$, $C_{ij} = \{R_{23}, R_{24}\}$,

Algorithm 6.2: DisAC3.1

Input : Agent i's portion of a distributed binary CSP $\mathcal{M}$.
A: a map that assigns to each variable $v \in V_i$ the set of all neighbor agents to which v is related through an external constraint.
Output: $\mathcal{D}_i$ or "inconsistent".

1 Run the echo algorithm to build a T-tree.
2 **if** parent $= i$ **then**
3   **foreach** $k, j \in \{1, 2, \ldots, p\}$ **do**
4         $s_{kj} \leftarrow -\infty, r_{kj} \leftarrow -\infty, \mathit{isIdle}[k] \leftarrow$ `false`
5 **foreach** $j = 1, 2, \ldots, p$ **do** $r_j \leftarrow -\infty$
6 $Q \leftarrow \left\{(v, w) \,\middle|\, v \in V_i, R_{vw} \in C_i \cup \bigcup_j C_{ij}, v \neq w\right\}$
7 **while** `true` **do**
8   **while** $Q \neq \varnothing$ **do**
9         $(v, w) \leftarrow Q.\textsc{pop}()$
10   **if** $\textsc{Revise}(v, w)$ **then**
11   **if** $D_v = \varnothing$ **then**
12   Send to all neighbors messages of type "inconsistent".
13   **return** "inconsistent"
14   **if** $A(v) \neq \varnothing$ **then**
15   **foreach** $j \in A(v)$ **do**
16   **foreach** $u \in V_j^i$ **s.t.** $R_{uv} \in C_{ji}$ **do**
17   **if** $\textsc{Revise}(u, v)$ **then**
18                             $s \leftarrow \textsc{CurrentTime}()$
19                             $\textsc{Send}(i, j,$ "domain update", $(v, D_v, s))$
20                             $\textsc{Send}(i,$ parent, "message sent", $(i, j, s))$
21   **break**
22             $Q \leftarrow Q \cup \{(u, v) \mid u \in V_i \text{ s.t. } R_{uv} \in C_i, u \neq v\}$
23     $\textsc{Send}(i,$ parent, "up to date", $(i, (r_j)_{j=1,\ldots,p}))$
24     $\mathit{messages} \leftarrow \textsc{Receive}()$
25   **while** $\mathit{messages} \neq \varnothing$ **do**
26         $(\mathit{msgType}, \mathit{msgContent}) \leftarrow \mathit{messages}.\textsc{pop}()$
27   **if** $\mathit{msgType} =$ "arc-consistent" **then**
28   Forward the message to all of its children.
29   **return** $\mathcal{D}_i$
30   **else if** $\mathit{msgType} =$ "inconsistent" **then**
31   Forward the message to the neighbors.
32   **return** "inconsistent"
33   **else if** $\mathit{msgType} =$ "domain update" **then**
34             $(w, D_w^i, r_j) \leftarrow \mathit{msgContent}$
35             $Q \leftarrow Q \cup \{(v, w) \mid v \in V_i \text{ s.t. } R_{vw} \in C_{ij}, v \neq w\}$
36   **else if** parent $= i$ **then**
37   **if** $\mathit{msgType} =$ "message sent" **then**
38                 $(k, j, s_{kj}) \leftarrow \mathit{msgContent}$
39   **if** $s_{kj} > r_{jk}$ **then** $\mathit{isIdle}[j] \leftarrow$ `false`
40   **if** $(s_{\ell j})_{\ell=1,\ldots,p} = (r_{j\ell})_{\ell=1,\ldots,p}$ **then** $\mathit{isIdle}[j] \leftarrow$ `true`
41   **if** $\mathit{msgType} =$ "up to date" **then**
42                 $(j, (r_k)_{k=1,\ldots,p}) \leftarrow \mathit{msgContent}$
43   **if** $(r_{jk})_{k=1,\ldots,p} = (s_{kj})_{k=1,\ldots,p}$ **then** $\mathit{isIdle}[j] \leftarrow$ `true`
44   **if** $\mathit{isIdle}[k] =$ `true` *for all* $k = 1, \ldots, p$ **then**
45   Send each of its children a message with type "arc-consistent".
46   **return** $\mathcal{D}_k$
47   **else** Forward the message to its parent.

- $V_p^i = \{v_6\}$, $\mathcal{D}_p^i = \{D_{v_6}^i\}$, $C_{ip} = \{R_{26}\}$.

In Algorithm 6.2, lines 2–4 and lines 36–46 are only run by the root agent, who is the dedicated agent to handle the termination. (Note that in our algorithm we distinguish the root agent from non-root agents by setting the parent of the root agent to itself.) These extra codes are used for handling termination of the algorithm, which are explained in details in Section 6.4.2.

In Algorithm 6.2, a queue Q is first initialized, which stores all arcs (v, w) with $R_{vw} \in C_i \cup \bigcup_j C_{ij}$ (line 6). The queue Q includes arcs (v, w) that correspond to constraints $R_{vw} \in C_i$ as is the case in AC2001/3.1. In addition, Q also includes arcs (v, w) that correspond to external constraints $R_{vw} \in C_{ij}$ with $v \in V_i$; excluded in the queue are arcs (w, v) that correspond to constraints $R_{wv} \in C_{ji}$ with $w \in V_j^i$ and $v \in V_i$.

Example 6.2. *Let the distributed binary CSP in Figure 1.2 be an input of* DisAC3.1*. Then Q in line 6 includes arcs $(v_1, v_2), (v_2, v_3), (v_2, v_4)$ and (v_2, v_6) but not arcs $(v_3, v_2), (v_4, v_2)$ and (v_6, v_2).*

Then the agent iteratively takes an arc (v, w) from Q, and revises D_v to make R_{vw} arc-consistent (lines 8–22). We will consider two cases: (i) If a domain D_v becomes empty, then the agent broadcasts “inconsistent” to its neighbors, and then returns “inconsistent” as its output (lines 11–13). Its neighbors will further broadcast “inconsistent” to their neighbors (in the standard agent communication graph) who have not yet received an “inconsistent” message (lines 30–31) and so on, until every agent has received an “inconsistent” message (c.f. Figure 6.4a). (ii) If a domain D_v is revised but is not empty, then there are arcs (u, v) that are potentially affected by the revision of D_v so that R_{uv} is no longer arc-consistent. Therefore, arcs (u, v) with $u \in V_i$ are added to Q (line 22), if they were not included already in Q. Arcs (u, v) with $u \in V_j^i$ for an agent j are dealt with slightly differently: since the

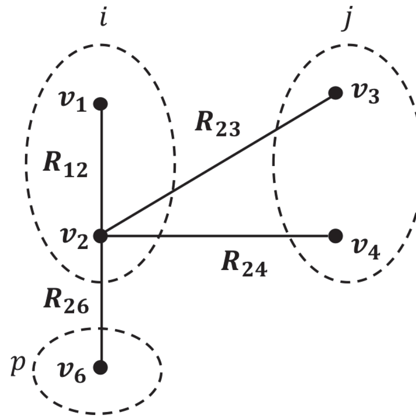


Figure 6.3 : The relevant parts of variable v_2 of the distributed binary CSP in Figure 1.2.

domain D_u of u is maintained by agent j in this case, agent i reports to its neighbor agent j about the revision of D_v so that agent j can revise D_u only when necessary (lines 14–19).[1] Here agent i sends for each revised domain at most one message to agent j, because there is no need to report to agent j about the same revision more than once (cf. line 21). The following example illustrates the ideas mentioned in this paragraph.

Example 6.3. *We consider the variable v_2 as well as other variables, domains and relations that are relevant to v_2, of the distributed binary CSP in Figure 1.2, which is illustrated in Figure 6.3. Let $D_{v_2} = \{a, b\}$, $D_{v_3} = \{c\}$, $D_{v_4} = \{d\}$, $D_{v_6} = \{e\}$ and $R_{23} = \{(a, c)\}$, $R_{24} = \{(a, d)\}$, $R_{26} = \{(a, e), (b, e)\}$. Then if a is removed from D_{v_2}, agent i will add arc (v_1, v_2) to Q if arc (v_1, v_2) was not already in Q, because constraint R_{12} may be no longer arc-consistent. Also, constraints R_{32}, R_{42} and R_{62} may be no longer arc-consistent. So agent i will notify the revision of D_{v_2} to agents j and p when necessary. Suppose i first checks whether it should inform p about the revision of D_{v_2} by calling* Revise(v_6, v_2). *Recall that i owns a copy $D^i_{v_6}$ of the original domain D_{v_6} of v_6. In this case, $D^i_{v_6} = D_{v_6} = \{e\}$. We know that*

[1]This technique was first introduced in [52].

$\text{REVISE}(v_6, v_2)$ *will return* false*, because although* a *is removed from* D_{v_2}*, we can still find another support* b *for* e *on* R_{62}*. In the process of running* $\text{REVISE}(v_6, v_2)$*, the algorithm only changes* e*'s smallest support on* R_{62} *from* a *to* b*. In other words, if* $\text{REVISE}(v_6, v_2)$ *returns* false*, we know that every element in* $D^i_{v_6}$ *is still supported on* R_{62}*. Thus,* R_{62} *is arc-consistent w.r.t.* $D^i_{v_6}$ *and* D_{v_2}*. Since we always have* $D^i_{v_6} \supseteq D_{v_6}$ *(cf. Lemma 6.2), we also have that* R_{62} *is arc-consistent w.r.t.* D_{v_6} *and* D_{v_2}*. Therefore,* i *finds that there is no need to notify the revision of* D_{v_2} *to* p*. However, with a similar analysis,* i *will find that* R_{32} *is not arc-consistent w.r.t.* $D^i_{v_3}$ *and* D_{v_2}*, and* R_{42} *is not arc-consistent w.r.t.* $D^i_{v_4}$ *and* D_{v_2}*, so* i *needs to notify the revision of* D_{v_2} *to the owners of* v_3 *and* v_4*. Because both* D_{v_3} *and* D_{v_4} *are owned by* j*,* i *only needs to inform the revision of* D_{v_2} *to* j *once.*

The root agent uses the T-tree to collect timestamps of each message that the sender and the recipient report. If an agent i sends a "domain update" message to one of its neighbors j, i will notify the root agent that a message was sent to agent j. To this end, agent i will first send a "message sent" message to its parent, and then the parent will forward the message to its parent (line 47) and so on, until the message is received by the root agent.[1] Note that in both of the messages to agent j and the root agent, we add the same timestamp s to the messages, which serves as a means for the root agent to confirm later that the message from agent i is received and processed by agent j.

Example 6.4. *Suppose that we have the T-tree in Figure 6.2 for the standard agent communication graph in Figure 6.1. Suppose that* k *sends a "domain update" message* m_{kj} *to* j *at time* s*. Then* k *sends a message* $(k, p, \text{"message sent"}, (k, j, s))$ *(line 20) with timestamp* s *to* p *and* p *will forward the message to its parent* i *by*

[1]Note that the sender's "id" remains in the forwarded message, but this "id" does not need to be the real id of the sender, it can be a *code name* (cf. [79]), e.g., a randomly generated character string. In this case, two neighbours need to exchange their code names with each other before running the algorithm.

sending another message $(p, i, \text{"message sent"}, (k, j, s))$ *(line 47). See Figure 6.4c for an illustration.*

Once agent i is done with adding new arcs to Q and sending messages to other agents, it repeats the whole process with the updated Q until there are no more arcs in Q (lines 8–22), i.e., Q is empty.

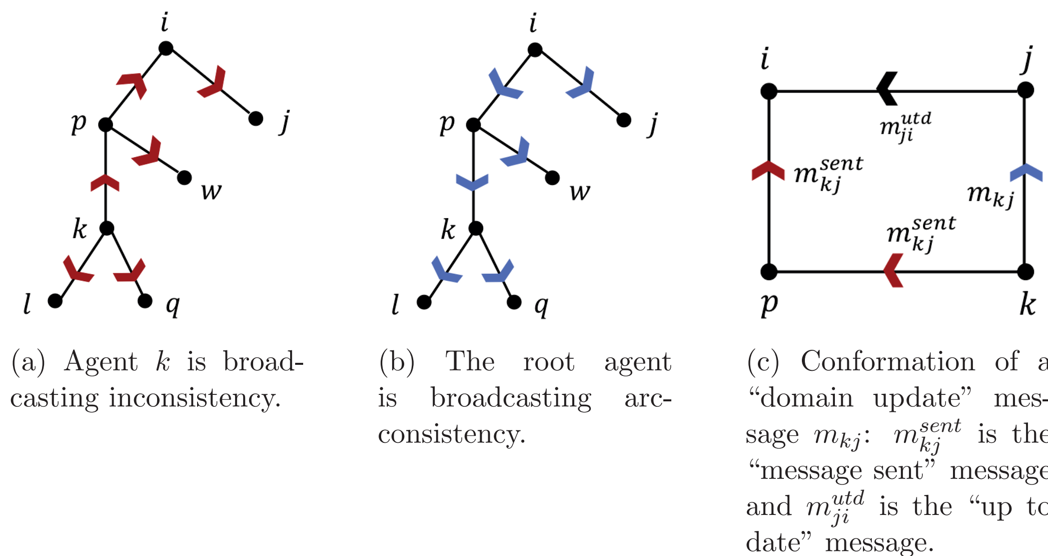
When there are no more arcs in Q to be processed, agent i reports to the root agent that its state is up to date (line 23), which is forwarded by the ancestors of agent i (cf. line 47), and shares with the root agent the latest timestamps $(r_{ij})_{j=1,\ldots,p}$ of all incoming messages of its neighbor agents j; these timestamps allow the root agent to determine whether the last message agent i received from agent j is the one that agent j reported to the root agent about.

Afterwards, agent i processes the messages that have been waiting to be processed in its message queue one by one in a FIFO manner (line 24). If the type of a message is "arc-consistent" or "inconsistent", then i will forward the message to relevant neighbors and then the algorithm terminates and returns $\mathcal{D}_i$ or the value "inconsistent", respectively (lines 27–32). If the type of a message is "domain update" sent from another agent j, which prompts agent i to update a domain D^i_w when $w \in V^i_j$, then agent i updates D^i_w and adds all arcs (v, w) that are potentially affected by the update to Q (lines 33–35).

When there are no more messages to be processed, then the algorithm repeats again with processing the arcs in the queue Q (line 8–22).

6.4.2 Handling Termination

A distributed algorithm should terminate when there are no messages in transit and all agents are waiting for new messages (i.e., idle). In fact, this is a well studied problem in the field of distributed systems, called *distributed termination detection*



(a) Agent k is broadcasting inconsistency.

(b) The root agent is broadcasting arc-consistency.

(c) Conformation of a "domain update" message m_{kj}: m_{kj}^{sent} is the "message sent" message and m_{ji}^{utd} is the "up to date" message.

Figure 6.4 : Illustrations of messages flow.

(DTD) [107]. DTD is a difficult problem as there is no simple way of gathering global knowledge of the distributed system. A number of algorithms to solve the DTD problem have been proposed in the literature (cf. [95]). These algorithms can be categorized based on eight features: algorithm type (wave, parental responsibility, credit-recovery, or ad-hoc), required topology of the agent communication graph, algorithm symmetry, required process knowledge (e.g., upper bound on the diameter of the agent communication graph or identity of the central agent), communication protocol (synchronous or asynchronous), communication channel behavior (first-in first-out (FIFO) or non-FIFO), message optimality, and fault tolerance [95].

Most DTD algorithms, like the one by Chandy and Misra [20], are wave algorithms and typically repetitive. These wave algorithms send wave after wave until termination is detected, which causes it to send up to $O(M \times p)$ control messages, where M is the number of basic messages and p is the number of agents. This repetitive property makes it unattractive for implementation. The algorithm proposed by Shavit and Francez [110] combines the algorithm of Dijkstra and Scholten [42] with a cycle-based repetitive wave algorithm.

One may integrate one of these DTD algorithms into DisAC3.1 to handle the its termination, but there are several aspects we need to bear in mind.

- Requirement on the topology of the agent communication graph and process knowledge should not be too strict. For example, we cannot require the agent communication graph to be a Hamiltonian cycle agent network (such as Dijkstra et al. [41], Shavit and Francez [110] and Mayo and Kearns [97]), or require agents to know the upper bound on agent network diameter (such as Skyum and Eriksen [43]), or require agents to know which one is the central agent (i.e., initiator and/or detector) (such as Huang [56]).
- Concerning performance, we would not consider algorithms that have very high message complexity (such as the repetitive wave algorithms [119, 20, 96] which require a large number of messages in the worst case) and algorithms that have high memory requirement (such as [121]).

In this chapter, we choose to develop a customized termination handling approach for algorithm DisAC3.1, which is not a wave algorithm, not parental responsible or credit-recovery. In our approach, agents report timestamps of both sent and received "domain update" messages to the root agent (lines 20 and 23). See Figure 6.5 for an illustration. The root agent runs additional codes (lines 2–4 and lines 36–43) to handle these timestamps.

There are two conditions to determine the termination of a distributed AC algorithm:

1. For each agent $i = 1, \ldots, p$ all "domain update" messages sent from other agents to agent i are received and processed by agent i and agent i is waiting for new messages;
2. An agent sent "inconsistent" messages to its neighbors, which indicates that

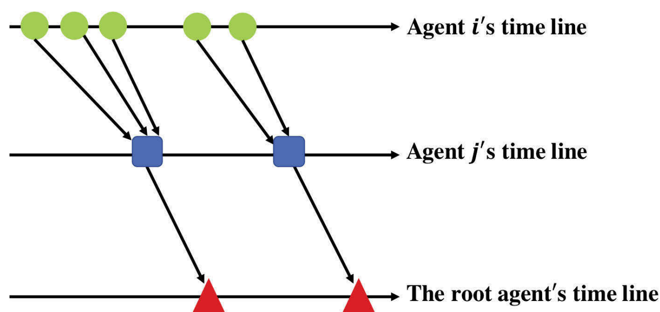


Figure 6.5 : Time lines of sending and receiving messages, where circles stand for the time points that agent i sends "domain update" messages to agent j, squares stand for the time points that agent j reports the latest received time stamps of "domain update" messages to the root agent via "up to date" messages, and triangles stand for the time points that the root agent receives "up to date" messages from agent j.

some local domain has become empty and the distributed CSP is thus inconsistent.

It is straightforward to check the satisfiability of the second condition. Once an agent finds out that one of its variable domains is empty, it sends an "inconsistent" message to each of its neighbors (line 12) and then other agents will help to broadcast inconsistency (lines 30–31). See Figure 6.4a for an example.

To check the satisfiability of the first condition, the root agent receives for each "domain update" message m_{ij} from sender i to recipient j two reports—one from the sender and one from the recipient, where the sender's report has type "message sent" and includes the same timestamp of m_{ij} and the recipient's report has type "up to date" and includes a timestamp not earlier than that of m_{ij}.[1] The root agent stores the timestamp reported by the sender i in variable s_{ij} (line 38) and the timestamp reported by the recipient j is stored in variable r_{ji} (line 42).

In what follows, we write m_{ij}^{sent} for the corresponding "message sent" message sent from agent i to the root agent which bears the same timestamp as m_{ij}; and

[1]This is because when several messages from agent i are received, the "up to date" message will only bear the timestamp of the latest one.

write m_{ij}^{utd} for the first "up to date" message sent from agent j to the root agent after it has received and processed m_{ij}.

For a message m_{ij} of type "domain update" we say m_{ij} is *confirmed* by the root agent, if (cf. Figure 6.4c)

- the root agent has received m_{ij}^{sent} (of type "message sent") from i;
- the root agent has received m_{ij}^{utd} (of type "up to date") sent from j.

When confirming a message we have to consider two cases. Since the communication speeds may be different among different pair of agents, m_{ij}^{sent} from agent i to the root agent may arrive earlier or later than m_{ij}^{utd} from agent j to the root agent. It is also worth reminding that every agent processes the messages that have been waiting to be processed in its message queue one by one in a FIFO manner.

Given a "domain update" message m_{ij} sent from agent i to agent j, if m_{ij}^{sent} arrives *earlier* at root agent than m_{ij}^{utd}, then the confirmation of message m_{ij} is accomplished when the root agent receives m_{ij}^{utd} (lines 41–43). On the other hand, if m_{ij}^{sent} arrives *later* at root agent than m_{ij}^{utd}, then the confirmation of message m_{ij} is accomplished when the root agent receives m_{ij}^{sent} (lines 37–40).

Whenever all messages sent to an agent i have been confirmed by the root agent (i.e., $s_{ji} = r_{ij}$ for all $1 \leq j \leq p$), then variable *isIdle*[i] is set to `true` (lines 40 and 43). When *isIdle*[i] = `true` for all agents i, then the root agent sends "arc-consistent" messages to all of its children (lines 44–45) and other agents further help broadcast the arc-consistent messages (lines 27–28). See Figure 6.4b for an example.

6.5 Analysis of DisAC3.1

6.5.1 Privacy of Individual Agents

For a distributed algorithm it is desirable that the algorithm does not give away the privacy of the involved agents, i.e., information of an individual agent is shared with other agents only when it is necessary to achieve a common goal. In the case of distributed binary CSPs such information could include information about the variables, domains, constraints and the communication address of each agent.

In this chapter, we assume that agents are *honest*, in the sense that no agent reveals its and/or its neighbors' private information (e.g., its variables, domains, constraints, or communication addresses) to irrelevant agents. With this assumption, in this subsection we show that DisAC3.1 leaks less private information of agents than existing distributed arc-consistency algorithms.

The following notion of *semi-private information* is adapted from [79].

Definition 6.2 (semi-private information)**.** *A piece of information about a DisCSP is regarded as* semi-private *if one agent or several agents might consider it private, but it can be leaked to irrelevant agents by their local knowledge of the AC-closure of the DisCSP.*

In this chapter, the semi-private information of a DisCSP concerns mainly about privacy of one agent (e.g., its variables, domains, and constraints) that can be inferred by another agent (or several agents jointly) by comparing the change of domains they know before and after enforcing a distributed arc-consistency algorithm.

Example 6.5. *Consider a DisCSP which has three variables x, y, z and two agents A, B such that A owns x, y and B owns z. Suppose there is a shared constraint R_{xz} that is AC in the beginning. Note that A knows the domain of z and B knows the domain of x, but it does not know the existence of y (let alone its domain) in the*

beginning. After enforcing AC, if D_z is changed, then B can infer from this change that A must own another variable which is connected to x. The information that "A has another variable which is connected to x" is thus a piece of semi-private information. This information will be leaked to B inevitably (even if we use encryption).

Theorem 6.2. *Let $\mathcal{M} = \langle \mathcal{P}, C^X \rangle$ be an input to Algorithm 6.2 and $\mathcal{N}_i$ the local binary CSP of agent i. Algorithm 6.2 satisfies the following conditions modulo semi-private information:*[1]

1. *The existence of a variable v of agent i and its domain are only known to agents who share the variable.*
2. *An external constraint of agent i is only known to another agent who shares the constraint with agent i.*
3. *The communication address of agent i is only known to its neighbor agents.*
4. *Private variables and their domains, and private constraints are all hidden to other agents.*

Proof. In Algorithm 6.2, information about shared variables and external constraints is only shared by agents that are involved in the external constraints, thus conditions 1–2 are satisfied modulo semi-private information. Furthermore, Algorithm 6.2 only requires the standard agent communication graph, i.e., two agents know the communication address of each other iff they share at least one external constraint, thus condition 3 is satisfied. Since no other information is revealed to other agents, condition 4 is satisfied modulo semi-private information. □

The four conditions in Theorem 6.2 are closely related to three types of privacy proposed by Léauté and Faltings [79], namely, agent privacy, topology privacy and

[1] Here by 'modulo semi-private information', we mean that we do not consider leak of semi-private information as privacy loss.

constraint privacy and the type of domain privacy introduced by Grinshpoun [49]. Agent privacy relates to the identities of the participants. Topology privacy concerns the existence of topological constructs in the constraint graph, such as nodes (i.e., variables) and edges (i.e., constraints). Constraint privacy corresponds to the nature of a constraint and domain privacy concerns the content of the domain of a variable, which could also be regarded as constraint privacy related to a unary constraint. Clearly, condition 1 is related to both topology and domain privacy, condition 2 is related to constraint privacy, and condition 4 involves topology, domain, and constraint privacy. As mentioned in [79], a particular consequence of the agent privacy is that two agents should only be allowed to communicate directly if they share a constraint. Therefore, condition 3 is related to the agent privacy.

However, we note that DisAC3.1as presented in Algorithm 2 does not strictly protect privacy of individual agents. First, constraint privacy is not strictly maintained if we care about semi-private information. Because even though agents cannot learn directly about constraint information, they can implicitly learn some information regarding constraints due to values that disappear from the domains of their neighbors. If such information is crucial in applications, one must apply some privacy protection measure (such as encryption [131]) to address the issues.

Second, DisAC3.1 does not protect all information about the agent constraint graph. Actually, before running DisAC3.1, we assume each agent only knows its neighbors in the standard agent communication graph and its parent and children in the T-tree, in particular, each agent knows whether it is the root agent. When an "up to date" or "message sent" message is sent from an agent, passing through internal agents, to the root agent, these internal agents and the root agent may learn about partial topology information of the agent communication graph. Consider the example in Figure 3. Suppose at time s agent k sends to j a "domain update" message m and sends to its parent p (cf. Figure 5) a "message sent" message m_{kj}^{sent}.

Then, since k is not the root, p will forward m_{kj}^{sent} to root i after receiving it. Note that m_{kj}^{sent} carries the information (k, j, s_{kj}) (here we assume j, k are disguised in their code names without leaking their identities) and thus both p and i know that (i) k is a descendant and (ii) k and j share a constraint. In general, each agent will know, in the extreme case and when the algorithm terminates, all its descendants (disguised in their code names) in the T-tree and all neighbors of each of these descendants in the standard agent communication graph. But this does not mean it will know the whole agent communication graph: it is unlikely that, for example, k knows that i is the root of the T-tree in Figure 5.

Note that leaking topology information of the agent communication graph to intermediate agents can be avoided by encrypting the whole message such that only the root agent can see the content, i.e., all the agents know the public key of the root agent and encrypt the messages using that key and the root agent, once receiving a message, decrypts the message by using its private key. No other intermediate agents can read the messages' content, as they do not have the private key of the root agent.

Alternatively, we can also alleviate the second limitation by introducing a *trustworthy system agent* instead of using the root agent of a T-tree to detect termination. The idea is to ask each agent to send their "message sent" and "up to date" messages directly to the system agent; if it detects that every agent is idle, then it sends an "arc-consistent" message to each agent; and if there is an inconsistency detected by an agent, it sends an "inconsistent" message directly to the system agent and the latter forwards the message to all other agents. In this *system-agent version of* DisAC3.1,[1] only the system agent knows the standard agent communication graph but it does not know the other private information (i.e., variables, domains, and constraints information about the DisCSP). For all non-system agents, they

[1]In this chapter, unless otherwise stated, DisAC3.1 always refers to the original Algorithm 2.

know only their neighbors in the standard agent communication graph. Another advantage of the system-agent version is that it's termination detection procedure is also message optimal (see Corollary 6.1). The system-agent version of DisAC3.1 requires a trustworthy system agent connecting to all other agents, which is not always available.

Unlike DisAC3.1, none of DisAC3, DisAC4 and DisAC6 satisfies conditions 1, 3, and 4 in Theorem 6.2 (modulo semi-private information). This is because they broadcast deleted values of variable domains to all agents, which reveals the existence of agents' variables and domains, and they assume a complete agent communication graph, which reveals every agent's communication address. DisAC9 does not satisfy condition 3 either, because the termination protocol it employs also assumes a complete agent communication graph. Therefore, we can conclude that DisAC3.1leaks less information about individual agents' communication addresses than all previous distributed AC algorithms. Nevertheless, we note that privacy loss of a distributed algorithm is difficult to evaluate outside the perspective of its integration with final applications such as search [112]. Since we do not integrate our algorithm with search in this chapter, we will not evaluate privacy loss of distributed AC algorithms. Future research may use existing privacy measuring approaches such as [92, 109] to measure privacy loss of distributed AC algorithms integrated with search.

6.5.2 Termination and Correctness

In this subsection, we prove that DisAC3.1 terminates and is correct. The proof of the termination result uses the following lemma.

Lemma 6.1. *The following conditions are equivalent:*

1. *isIdle*$[i] = \texttt{true}$ *for all* $i = 1, \ldots, p$.

2. *The root agent has received from all agents their first "up to date" messages*

(cf. line 23) and $s_{ij} = r_{ji}$ *for all* $i, j = 1, \ldots, p$.

3. *All "domain update" messages are confirmed and no agent is running the first inner while-loop (cf. lines 7-22).*

Proof. See Appendix. □

Theorem 6.3. *Algorithm 6.2 terminates.*

Proof. Let $\mathcal{N}_i = \langle V_i, \mathcal{D}_i, C_i \rangle$ be agent i's local binary CSP of an input distributed binary CSP $\mathcal{M} = \langle \mathcal{P}, C^X \rangle$. We note first that after a certain number of iterations of the outer while-loop (lines 7–47) of Algorithm 6.2, either the algorithm exits and returns a value ($\mathcal{D}_i$ or "inconsistent") or waits for new messages (line 24). The outer while-loop cannot iterate forever, as this can only happen, if at each iteration: (i) there exists a neighbor agent j who sends "domain update" messages to agent i (line 33), i.e., the domain of some variable of j has been updated and its size decreased; however, because all domains of $\mathcal{M}$ are finite, the number of "domain update" messages is finite, or (ii) there exists some agent k sends "message sent" messages to the root agent; however, because one "domain update" message induces one "message sent" message, the number of "message sent" messages is also finite, or (iii) there exists some agent k sends "up to date" messages to the root agent; however, because one "domain update" message induces at most one "up to date" message, the number of "up to date" messages is also finite. Analogously, the two inner while-loops (lines 8–22 and lines 27–47) iterate only finitely many times.

Since i is chosen arbitrarily above, we can assume that all "domain update" messages that were sent have been confirmed by the root agent and no agent is running the first inner while-loop. Then, by Lemma 6.1, *isIdle*$[i]$ = `true` for all $i = 1, \ldots, p$ and the root agent sends "arc-consistent" messages to its children and exits (lines 44–46), and agent i, on receiving the message, forwards the message and

returns $\mathcal{D}_i$, and then Algorithm 6.2 terminates. □

To prove the correctness of DisAC3.1, we need the following two lemmas.

Lemma 6.2. *Let $\mathcal{M} = \langle \mathcal{P}, C^X \rangle$ be an input distributed binary CSP for Algorithm 6.2, and let $v \in V_i$ and $u \in V_j$ such that $R_{uv} \in C_{ji}$ for some agents i and j, $i \neq j$. Then, the algorithm terminates with $D_u^i \supseteq D_u$.*

Proof. See Appendix. □

Lemma 6.3. *Let $\mathcal{M} = \langle \mathcal{P}, C^X \rangle$ be an input distributed binary CSP for Algorithm 6.2. Then, each application of* Revise *does not change the AC-closure of $\mathcal{M}$.*

Proof. When popping an arc (v, w) out of Q (line 9), the application of function Revise(v, w) only removes values from D_v that have no supports w.r.t. R_{vw}. We know that those values cannot be parts of the AC-closure [89]. Thus, each application of Revise does not change the AC-closure of $\mathcal{M}$. □

Theorem 6.4. *Given an input distributed binary CSP $\mathcal{M} = \langle \mathcal{P}, C^X \rangle$, if Algorithm 6.2 returns a new set of domains $\mathcal{D}'_i$ for $i = 1, \ldots, p$, then $\mathcal{M}' = \langle \mathcal{P}', C^X \rangle$, where $\mathcal{P}' = \{\mathcal{N}'_1, \ldots, \mathcal{N}'_p\}$ and $\mathcal{N}'_i = \langle V_i, \mathcal{D}'_i, C_i \rangle$, is the AC-closure of $\mathcal{M}$. If the algorithm returns "inconsistent" then $\mathcal{M}$ is inconsistent.*

Proof. Suppose Algorithm 6.2 returns $\mathcal{D}'_i$ for $i = 1, \ldots, p$. It does so only if during the execution of the distributed algorithm there was a time (or a state) t_{AC} where *isIdle*$[i] =$ `true` for all $i = 1, \ldots, p$ (cf. Algorithm 6.2, lines 44–46). Thus to prove the claim we need to show that at time t_{AC} the outcome $\mathcal{M}' = \langle \mathcal{P}', C^X \rangle$ is AC. To this end, we show that at time t_{AC} for each agent $i = 1, \ldots, p$ and for each neighbor agent j of agent i, any constraint $R_{v_0 w_0} \in C_i \cup C_{ij}$ is AC.

Suppose $R_{v_0w_0}$ is from C_i. Due to Lemma 6.1 we can assume that at time t_{AC} all "domain update" messages that were sent have been confirmed by the root agent and no agent is running the first inner while-loop. We show inductively that $R_{v_0w_0}$ is AC.

1. After the algorithm initializing the queue Q (line 6), running the first inner while-loop (lines 8–22) for the first time and waiting for new messages (line 24), then $R_{v_0w_0}$ is AC. This is because all operations involved are exactly the same as those in AC2001/3.1 except for the operations in lines 14–21, which however do not affect the domains of v_0 and w_0.

2. After receiving new "domain update" messages (line 33), new pairs of variables are added to Q. When the algorithm enters the first inner while-loop again then the queue Q is processed as in AC2001/3.1 (except for the operations in lines 14–21), establishing arc-consistency of $R_{v_0w_0}$.

Thus, $R_{v_0w_0}$ is AC at time t_{AC}.

Now suppose j is a neighbor agent of i and $R_{v_0w_0}$ is from C_{ij}, i.e., variable w_0 does not belong to agent i but to agent j. Looking at the algorithm from agent j's perspective, we have by Lemma 6.2 that $D^j_{v_0} \supseteq D_{v_0}$ at time t_{AC}. We also observe that due to line 17 at time t_{AC} constraint $R_{v_0w_0}$ is AC w.r.t. $D^j_{v_0}$ and D_{w_0}. Thus, at time t_{AC} for any $a \in D_{v_0} \subseteq D^j_{v_0}$ there exists $b \in D_{w_0}$ such that $(a, b) \in R_{v_0w_0}$, i.e., $R_{v_0w_0}$ is AC at time t_{AC}.

Thus $\mathcal{M}'$ is an AC-subnetwork of $\mathcal{M}$. By Lemma 6.3 it is also the AC-closure of $\mathcal{M}$.

If the algorithm returns "inconsistent", then it follows by Lemma 6.3 that the AC-closure of $\mathcal{M}$ has an empty domain. Thus, $\mathcal{M}$ is inconsistent. □

6.5.3 Time and Space Complexities

When analyzing the time complexity of a distributed algorithm, one usually assumes that the delay of sending a message is zero.

Theorem 6.5. *The time complexity of Algorithm 6.2 is $O(ed^2)$, where e is the number of edges in the constraint graph of the input distributed binary CSP and d is the largest domain size.*

Proof. In the analysis of time complexity we are interested in the number of increment operations (lines 16–17 of Algorithm 6.2) on the elements of each domain as this dominates all other numbers of operations. These increment operations take place in lines 5 and 7 of Function Revise. To this end, we need to first determine the number of calls to Revise. We first note that Revise is applied to a pair (v, w) of variables only in one of the two cases: (i) the pair is from Q, in particular, the first variable of the pair is from V_i (lines 9–10 of Algorithm 6.2) (ii) the first variable of the pair is from V_j^i, the set of all variables from agent j that are related to i through an external constraint (lines 16–17 of Algorithm 6.2). In either of the mentioned two cases, Revise is applied to a pair of variables, which was added to Q not because of the initialization of Q in line 6, if and only if the domain of the second variable was revised. Except the first revision, since each revision of a domain reduces its size, each ordered pair of variables are revised at most $d + 1$ times. Let (v, w) be a pair of variables and t_ℓ $(1 \leq \ell \leq d + 1)$ be the number of increment operations that takes place at the ℓth call to Revise(v, w). Then $\sum_1^{d+1} t_\ell \leq d^2$, because for each $a \in D_v$ only d many increments of $SS_{vw}(a)$ is possible. Thus, for each agent i there are altogether $T_i = O((|C_i| + |C_{ij}|) \cdot \sum_1^{d+1} t_\ell) = O((|C_i| + |C_{ij}|) \cdot d^2)$ number of increment operations. In the worst case, DisAC3.1 proceeds with a sequential

behavior and the time complexity of DisAC3.1 is

$$\sum_{i=1}^{p} T_i = \sum_{i=1}^{p} O((|C_i| + |C_{ij}|) \cdot d^2) \le d^2 \sum_{i=1}^{p} O(|C_i| + |C_{ij}|)$$
$$\le d^2 \cdot O(|C|) = O(ed^2).$$

Since running the echo algorithm to build a T-tree only takes $O(\hat{D})$ time, where $\hat{D} \le p \le n$ is the diameter of the standard agent communicate graph, it does not affect the time complexity of DisAC3.1. □

Theorem 6.6. *The space complexity of Algorithm 6.2 is $O(ed)$.*

Proof. For each variable $w \in V_j^i$ of some agent j, agent i keeps a copy of D_w, which requires $O(ed)$ space for all agents. Since the other used data structures of DisAC3.1 are the same as that of AC2001/3.1, the space complexity is the same as that of AC2001/3.1, i.e., $O(ed)$. Since the echo algorithm only takes $O(p)$ space with $p \le n$, it does not affect the space complexity of DisAC3.1. □

Number of Message Passing Operations

We analyze the number of messages sent in DisAC3.1.

Theorem 6.7. *Let $\mathcal{M}$ be a distributed binary CSP. Then, on input $\mathcal{M}$,* DisAC3.1 *sends at most $O(nd\alpha h)$ messages, where α is the largest vertex degree of the standard agent communication graph G of $\mathcal{M}$ and h is the height of the T-tree of G.*

Proof. DisAC3.1 send at most $nd\alpha$ “domain update” messages, because there are at most nd domain elements in total and each deletion of the elements in a domain leads to the dispatch of at most α “domain update” messages. Each dispatch of a “domain update” message is followed by the dispatch of a “message sent” message as well as of at most one “up to date” message. Because a “message sent” or an “up

to date” message may need to be forwarded at most h times until it reaches the root agent, there are at most $O(nd\alpha h)$ messages that Algorithm 6.2 sends. Since the echo algorithm sends $O(\hat{e})$ messages, where $\hat{e} \leq e \leq n\alpha$ is the number of edges of the standard agent communication graph, it does not affect the message complexity of DisAC3.1. □

Consider the system-agent version of DisAC3.1. For each distributed binary CSP $\mathcal{M}$, after introducing a system agent A_0, the standard communication graph of G of $\mathcal{M}$ has a T-tree with height 1, in which the root is A_0 and all regular agents are children of A_0. By Theorem 6.7, the system-agent version of DisAC3.1 sends at most $O(nd\alpha)$ messages on input $\mathcal{M}$. Regarding “domain update” messages as basic messages and “message sent’ and “up to date” messages as control messages, from the proof of Theorem 6.7, it is easy to see that the system-agent version of DisAC3.1 sends at most $O(M)$ control messages in total, where M is the number of basic messages it sends. Because the number of control messages to detect the termination of a distributed system is at least M [21], the following corollary follows directly from Theorem 6.7.

Corollary 6.1. *The termination detection procedure of the system-agent version of* DisAC3.1 *is message optimal.*

6.6 Non-binary Constraints

In this section we extend our results to k-ary CSPs.

6.6.1 Generalized Arc-Consistency

Definition 6.3 (generalized arc-consistency [100])**.** *Let $\mathcal{N} = \langle V, D, C\rangle$ be a k-ary CSP. Let (s, R) be a constraint of C with $s = (v_1, \dots, v_\ell)$. Given a value $a \in D_{v_i}$ with $1 \leq i \leq \ell$, a tuple $t = (t[v_1], \dots, t[v_\ell]) \in R$ is called a* candidate support

v_1	v_2	v_3
a	c	e
a	c	f
b	d	f

Figure 6.6 : A ternary constraint (s, R), where $s = (v_1, v_2, v_3)$ and $R = \{(a, c, e), (a, c, f), (b, d, f)\}$.

of a on R, if $t[v_i] = a$. If t is a candidate support of a on R and additionally $t \in D_{v_1} \times \ldots \times D_{v_\ell}$ then we call t a support *of a on R.*

A constraint (s, R) is said to be generalized arc-consistent (GAC) *iff for each variable v in s every value $a \in D_v$ has a support on R. CSP $\mathcal{N} = \langle V, D, C\rangle$ is called GAC iff every constraint of C is GAC.*

We note that for binary CSPs GAC coincides with AC.

Definition 6.4 (GAC-closure)**.** *Given a network $\mathcal{N} = \langle V, D, C\rangle$, let $\mathcal{N}' = \langle V, \mathcal{D}', C'\rangle$ be a subnetwork of $\mathcal{N}$. We call $\mathcal{N}'$ an GAC-subnetwork of $\mathcal{N}$, if $C' = C$ and $\mathcal{N}'$ is generalized arc-consistent and not empty. We call $\mathcal{N}'$ the* GAC-closure *of $\mathcal{N}$, if $\mathcal{N}'$ is the* largest *GAC-subnetwork of $\mathcal{N}$ that is equivalent to $\mathcal{N}$, in the sense that every other GAC-subnetwork $\mathcal{N}'' = \langle V, \mathcal{D}'', C\rangle$ of $\mathcal{N}$ that is equivalent to $\mathcal{N}$ is a subnetwork of $\mathcal{N}'$.*

Every k-ary constraint (s, R) can be represented as a table, where entries of the table are the tuples of R. We use $\prec_R$ to denote the ordering of entries of the table representation of R. For each constraint (s, R) and for each variable $v \in s$ and each value $a \in D_v$, the *smallest support* of a on R w.r.t. $\prec_R$ is stored in $SS_R(a)$.

Example 6.6. *A ternary constraint (s, R) is represented as a table in Figure 6.6. Suppose $D_{v_1} = \{a, b\}, D_{v_2} = \{c, d\}$ and $D_{v_3} = \{e, f, g\}$. Then the tuple (a, c, e) is the smallest support of $a \in D_{v_1}$ on R. However, $g \in D_{v_3}$ has no support on R, so this constraint is not GAC.*

In [12], algorithm AC2001/3.1 is extended to a generalized arc-consistency (GAC) algorithm, called GAC2001/3.1. The main change is the extension of function REVISE to n-ary constraints. In this extended REVISE, finding a support for $a \in D_{v_i}$ on a given constraint (s, R) is realized by checking for each $t \in D_{v_1} \times \ldots \times D_{v_\ell}$ first whether $t[v_i] = a$ and then whether $t \in R$. This requires to check $O(d^{k-1})$ tuples, where d is the maximum number of elements in a domain and k is the arity of R.

In the following we present function GREVISE, which is a modification of the extended REVISE. Here, we check first whether $t \in R$ following the ordering $\prec_R$ and then check whether $t[v_i] = a$ and $t[v_j] \in D_{v_j}$ for all $v_j \in s \setminus \{v_i\}$. Since membership queries can be done in $O(1)$ using hash-tables, this modification allows for reduced number of checks when R is sparse.

Function GREVISE(v, s, R)

```
1  revised ← false
2  foreach a ∈ D_v do
3      t ← SS_R(a)
4      if ∃v' ∈ s with t[v'] ∉ D_v' then
5          t ← NextTuple(t, R)
6          while t ≠ NIL and (t[v] ≠ a or ∃v' ∈ s with t[v'] ∉ D_v') do
7              t ← NextTuple(t, R)
8          if t ≠ NIL then
9              SS_R(a) ← t
10         else
11             delete a from D_v
12             revised ← true
13 return revised
```

Next we present a GAC algorithm as Algorithm 6.3, called GAC2001/3.1$*$. It is different from AC2001/3.1 in that it stores in Q triples (v, s, R). It is also different from GAC2001/3.1 in that it calls the modified function GREVISE (line 4). GAC2001/3.1$*$ shares the nice property with AC2001/3.1, i.e., for each value $a \in D_v$ and each constraint (s, R) with $v \in s$, a tuple $t \in R$ is only visited once.

Algorithm 6.3: GAC2001/3.1∗

Input : A CSP $\mathcal{N} = \langle V, D, C \rangle$.
Output: The GAC-closure of $\mathcal{N}$, or "inconsistent".

```
1 Q ← {(v, s, R) | (s, R) ∈ C, v ∈ s}
2 while Q ≠ ∅ do
3     (v, s, R) ← Q.POP()
4     if GREVISE(v, s, R) then
5         if D_v = ∅ then
6             return "inconsistent"
7         Q ← Q ∪ {(w, s, R) | (s, R) ∈ C, v, w ∈ s, w ≠ v}
8 return N
```

Theorem 6.8. *The time complexity of algorithm* GAC2001/3.1∗ *is* $O(ek^2d\beta)$ *where* e *is the number of constraints of the input CSP,* k *is the arity of the network,* d *is the largest domain size and* β *is the largest number of candidate supports of a value on a relation.*

Proof. We first analyze the cost of enforcing GAC on a single constraint (s, R). For any variable $v \in s$ and any value $a \in D_v$, there are at most β candidate supports needed to be confirmed (line 6 of Function GRevise). Confirming a candidate costs $O(k-1)$ time. Because there are rd values needed to be processed per constraint, enforcing GAC on a single constraint costs $O(rd \cdot \beta \cdot (k-1)) = O(k^2d\beta)$. Since there are at most e constraints, the time complexity of GAC2001/3.1∗ is $O(ek^2d\beta)$. □

Note that we must have $\beta \leq d^{k-1}$, so in the worst case the time complexity of GAC2001/3.1∗ is $O(ek^2d^k)$, which is the same as that of GAC2001/3.1.

6.6.2 A Distributed Generalized Arc-Consistency Algorithm

We now extend GAC2001/3.1∗ to a distributed generalized arc-consistency algorithm, called DisGAC3.1.

Definition 6.5 (distributed k-ary CSP)**.** *A distributed* k-ary CSP (DisCSP) *is defined as a pair* $\langle \mathcal{P}, C^X \rangle$*, where*

- $\mathcal{P} = \{\mathcal{N}_1, \ldots, \mathcal{N}_p\}$ *is a set of* k*-ary CSPs with* $\mathcal{N}_i = \langle V_i, \mathcal{D}_i, C_i \rangle$ $(1 \leq i \leq p)$*;*
- C^X *is a set of* external *constraints, where each* $(s, R) \in C^X$ *is shared by at least two different agents, i.e., there exist two different variables* $v, w \in s$ *s.t.* $v \in V_i, w \in V_j, i \neq j$*.*

The definitions of (standard) agent communication graphs of distributed binary CSPs (cf. Definition 6.1), GAC (cf. Definition 6.3) and GAC-closure (cf. Definition 6.4) naturally carry over to distributed k-ary CSPs.

The distributed GAC algorithm is presented as Algorithm 6.4. Given a DisCSP $\mathcal{M} = \langle \mathcal{P}, C^X \rangle$ $(1 \leq i \leq p)$ each agent i takes its portion of $\mathcal{M}$ as an input and runs its own copy of Algorithm 6.4 separately and concurrently. Agent i's portion of $\mathcal{M}$ includes:

- $\mathcal{N}_i = \langle V_i, \mathcal{D}_i, C_i \rangle$,
- $C^X(V_i) = \big\{(s, R) \in C^X \bigm| s \cap V_i \neq \varnothing\big\}$,
- a set $\mathcal{D}^i$ of D^i_w, where D^i_w is a copy D_w and w belongs to a neighbor agent $j \neq i$, i.e., there is an external constraint $(s, R) \in C(V_i)$ such that w is in scope s.

Example 6.7. *Suppose we have a DisCSP* $\mathcal{M} = \langle \mathcal{P} = \{\mathcal{N}_1, \mathcal{N}_2\}, C^X \rangle$*, where* $\mathcal{N}_1 = \langle V_1 = \{v_1\}, \mathcal{D}_1 = \{D_{v_1}\}, C_1 = \varnothing \rangle$*,* $\mathcal{N}_2 = \langle V_2 = \{v_2, v_3\}, \mathcal{D}_2 = \{D_{v_2}, D_{v_3}\},\ C_2 = \varnothing \rangle$ *and* $C^X = \{(s_1 = (v_1, v_2, v_3), R_1), (s_2 = (v_1, v_2), R_2)\}$ *as an input to Algorithm 6.4. Agent* 1*'s portion of* $\mathcal{M}$ *includes:*

- $\mathcal{N}_1$,

- $C^X(V_1) = \{(s_1, R_1), (s_2, R_2)\}$,
- $\mathcal{D}^1 = \{D^1_{v_2} = D_{v_2}, D^1_{v_3} = D_{v_3}\}$.

Algorithm 6.4 largely overlaps with Algorithm 6.2. Differences are highlighted in light gray background. In Algorithm 6.4, a queue Q is first initialized to store all the tuples (v, s, R) with $(s, R) \in C_i \cup C^X$ and $v \in s$ (line 7), and line 24 adds all new tuples (w, s', R') to Q with $w \in V_i, v, w \in s'$ and $w \neq v$ because of the revision of D_v. Also, if a new domain D_w is received from some other agent, agent i will add new tuples (v, s, R) to Q, where (s, R) is an external constraint, $v \in V_i$ and both $v, w \in s$ (line 37). Lines 16, 18 and 20 are to guarantee that if a domain D_v is revised, D_v is sent to relevant agents at most once, just like line 21 of Algorithm 6.2.

The correctness of DisGAC3.1 follows from the correctness of DisAC3.1 and GAC2001/3.1$*$, and we have the following:

Theorem 6.9. *Given an input DisCSP $\mathcal{M} = \langle \mathcal{P}, C^X \rangle$, Algorithm 6.4 terminates. If Algorithm 6.4 returns a new set of domains $\mathcal{D}'_i$ for $i = 1, \ldots, p$, then $\mathcal{M}' = \langle \mathcal{P}', C^X \rangle$, where $\mathcal{P}' = \{\mathcal{N}_1, \ldots, \mathcal{N}'_p\}$ and $\mathcal{N}'_i = \langle V_i, \mathcal{D}'_i, C_i \rangle$, is the GAC-closure of $\mathcal{M}$. If the algorithm returns "inconsistent" then $\mathcal{M}$ is inconsistent.*

Theorem 6.10. *The time and space complexities of algorithm* DisGAC3.1 *are $O(ek^2\beta)$ and $O(ndke)$, respectively, where e is the number of constraints of the input network, k is the arity of the network, d is the largest domain size and β is the largest number of candidate supports of a value on a relation.*

Proof. In the worse case, algorithm DisGAC3.1 proceeds with a sequential behavior, so constraints will be enforced GAC one by one and DisGAC3.1 has the same worse case time complexity of algorithm GAC2001/3.1$*$, which is $O(ek^2\beta)$.

Because every constraint can have a scope containing up to k variables, Q takes $O(ke)$ space. Since, in the worst case, each agent can share all the variables of other

Algorithm 6.4: DisGAC3.1

Input : i's portion of a DisCSP $\mathcal{M}$.
Output: $\mathcal{D}_i$ or "inconsistent".

```
1  Run the echo algorithm to build a T-tree.
2  if parent = i then
3      foreach k, j ∈ {1, 2, ..., p} do
4          s_kj ← −∞, r_kj ← −∞
5          isIdle[k] ← false
6  foreach j = 1, 2, ..., p do r_j ← −∞
7  Q ← {(v, s, R) | v ∈ V_i, (s, R) ∈ C_i ∪ C^X, v ∈ s}
8  while true do
9      while Q ≠ ∅ do
10         (v, s, R) ← Q.pop()
11         if GRevise(v, s, R) then
12             if D_v = ∅ then
13                 Send each of its neighbors a message with type "inconsistent".
14                 return "inconsistent"
15             if C^X(v) ≠ ∅ then
16                 L ← ∅
17                 foreach (s, R) ∈ C^X(v) do
18                     foreach u ∈ s s.t. u ∉ V_i, Owner(u) ∉ L do
19                         if GRevise(u, s, R) then
20                             L.append(Owner(u))
21                             s ← CurrentTime()
22                             Send(i, j, "domain update", (v, D_v, s))
23                             Send(i, parent, "message sent", (i, j, s))
24             Q ← Q ∪ {(w, s, R) | w ∈ V_i, (s, R) ∈ C_i ∪ C^X, v, w ∈ s, w ≠ v}
25     Send(i, parent, "up to date", (i, (r_j)_{j=1,...,p}))
26     messages ← Receive()
27     while messages ≠ ∅ do
28         (msgType, msgContent) ← messages.pop()
29         if msgType = "arc-consistent" then
30             Forward the message to all of its children.
31             return D_i
32         else if msgType = "inconsistent" then
33             Forward the message to the neighbors.
34             return "inconsistent"
35         else if msgType = "domain update" then
36             (w, D^i_w, r_j) ← msgContent
37             Q ← Q ∪ {(v, s, R) | v ∈ V_i, (s, R) ∈ C^X, v, w ∈ s}
38         else if parent = i then
39             if msgType = "message sent" then
40                 (j, s_kj) ← msgContent
41                 if s_kj > r_jk then isIdle[j] ← false
42                 if (s_ℓj)_{ℓ=1,...,p} = (r_jℓ)_{ℓ=1,...,p} then isIdle[j] ← true
43             if msgType = "up to date" then
44                 (j, (r_k)_{k=1,...,p}) ← msgContent
45                 if (r_jk)_{k=1,...,p} = (s_kj)_{k=1,...,p} then isIdle[j] ← true
46             if disIdle[k] = true for all k = 1, ..., p then
47                 Send each of its children a message with type "arc-consistent".
48                 return D_k
49         else Forward the message to its parent.
```

agents by means of external constraints, copied domains $\mathcal{D}^i$, $(i = 1, \ldots, p)$ takes in total $O(ndp)$ space. The data structure SS_R of Function GRevise associated with a constraint (s, R) takes $O(ndk)$ space, because there are at most $O(nd)$ domain values for a constraint (s, R) and we store for each value its smallest support, which takes $O(k)$ space. Since there are e constraints SS takes $O(ndre)$ space. By adding all this together we obtain $O(ne + ndp + ndke) = O(nd(p + ke))$. Since we assume that the standard agent communication graph is connected, for each agent there is an external constraint (s_i, R_i) shared by at most $k_i - 1$ other agents where k_i is the arity of (s_i, R_i) and we have $p \leq \sum_{i=1}^{e} k_i \leq ke$. Thus $O(ndke)$ is the space complexity of DisGAC3.1.

Similar to the cases of DisAC3.1, running the echo algorithm does not affect the time and space complexities of DisGAC3.1. □

Since the termination handling mechanism of DisGAC3.1 is same as that of DisAC3.1, we have the following result.

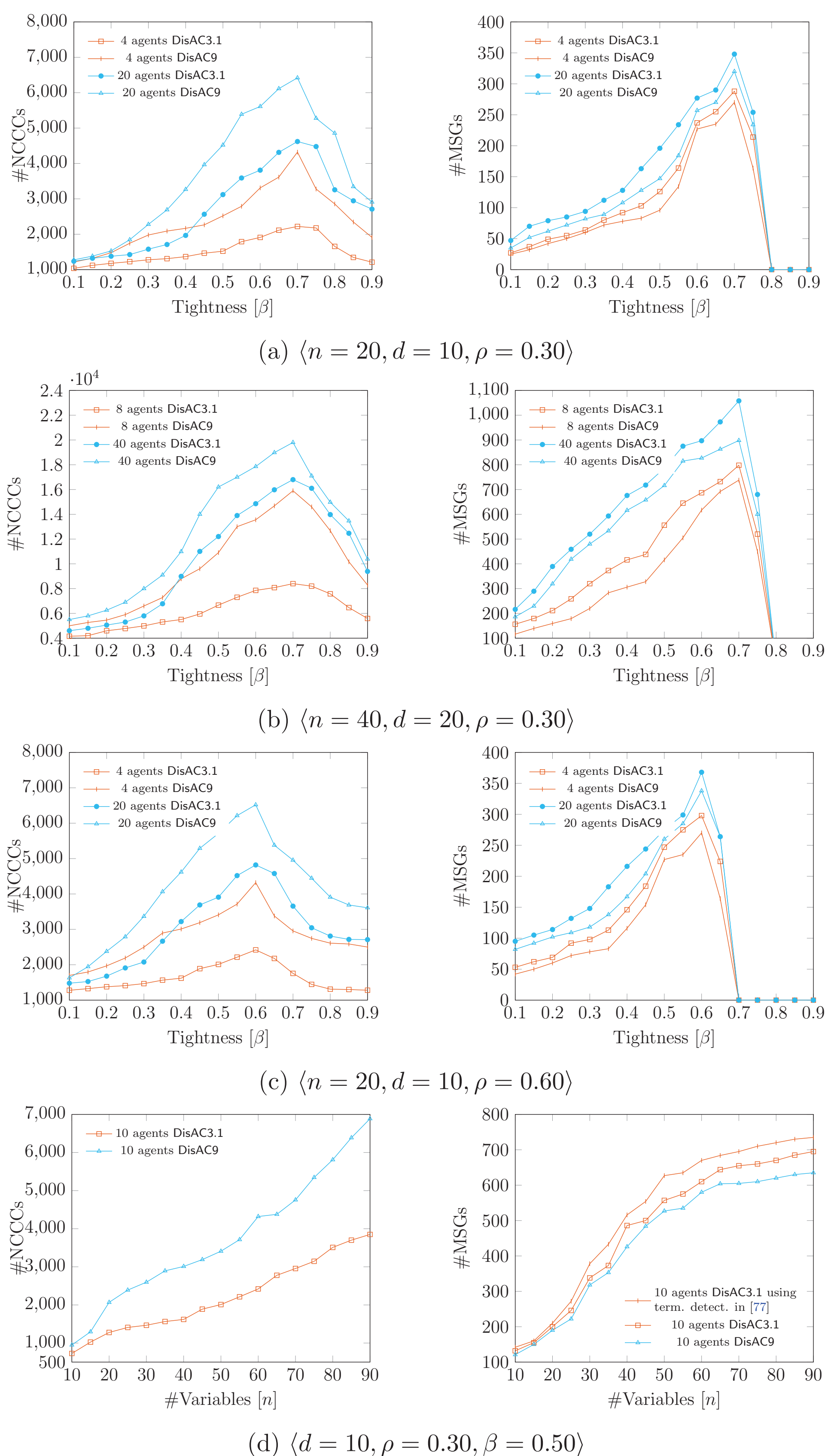
Theorem 6.11. *Let $\mathcal{M}$ be an input DisCSP for* DisGAC3.1*. Then* DisGAC3.1 *send at most $O(nd\alpha h)$ messages, where α is the largest vertex degree of the standard agent communication graph G of $\mathcal{M}$ and h is the height of the T-tree of G.*

Similar to the system-agent version of DisAC3.1 discussed before, we can also introduce a system-agent version of DisGAC3.1[1], which sends at most $O(nd\alpha)$ messages.

6.7 Evaluation

In this section we experimentally compare DisAC3.1 against the state-of-the-art distributed AC algorithm DisAC9. Comparison of theoretical time and space complexities between DisAC3.1 and DisAC9 is also given in Table 6.1. We also evaluate

[1] In this chapter, unless otherwise stated, DisGAC3.1 always refers to the original Algorithm 4.



(a) $\langle n = 20, d = 10, \rho = 0.30 \rangle$

(b) $\langle n = 40, d = 20, \rho = 0.30 \rangle$

(c) $\langle n = 20, d = 10, \rho = 0.60 \rangle$

(d) $\langle d = 10, \rho = 0.30, \beta = 0.50 \rangle$

Figure 6.7 : Performance comparisons between DisAC3.1 and DisAC9 on random instances.

Algorithm	DisAC3.1	DisAC9
Time	$O(ed^2)$	$O(n^2d^3)$
Space	$O(ed)$	$O(n^2d)$

Table 6.1 : Comparison of time and space complexities between DisAC3.1 and DisAC9.

our GAC algorithm DisGAC3.1 by comparing it against its centralized counterpart GAC2001/3.1∗, owing to lack of other existing distributed GAC algorithm. All experiments used an asynchronous simulator in which agents are simulated by processes which communicate only through message passing. Two independent measures of performance are commonly used for evaluating the performance of distributed algorithms: (i) the number of *non-concurrent constraint checks* (#NCCCs) (cf. [14, 98]) and (ii) the total number of messages sent (#MSGs) (cf. [88]).[1] All experiments for distributed algorithms used an asynchronous simulator in which (i) agents are simulated by processes which communicate only through message passing and (ii) default communication latency is assumed to be zero. Our experiments were implemented in Python 3.6 and carried out on a computer with an Intel Core i5 processor with a 2.9 GHz frequency per CPU, 8 GB memory.

6.7.1 DisAC3.1 vs. DisAC9

To evaluate the two distributed AC algorithms, we consider both benchmark problems and randomly generated binary CSPs whose AC-closures are not empty. The random instances were generated by using the random model proposed in [57]. The generator involves four parameters: (1) the number of variables n, (2) the largest domain size d, (3) the density $\rho = 2|C|/n(n+1)$ of the input binary CSP,

[1]Note that these two measures do not reflect the cost of the echo algorithm used in our distributed algorithms. However, since the number of agents are relatively small comparing to the number of variables of the input and the time complexity of the echo algorithm is only $O(\hat{D})$, where $\hat{D} \leq p \leq n$ is the diameter of the standard agent communication graph, the cost of the echo algorithm is negligible.

Instances $\langle n, d, m, p \rangle$	DisAC3.1 #NCCCs	DisAC3.1 #Msgs	DisAC9 #NCCCs	DisAC9 #Msgs
$\langle 1000, 10, 1000, 20 \rangle$	1.031e+4	964	2.137e+4	923
$\langle 500, 100, 500, 25 \rangle$	3.013e+5	3027	5.854e+5	3078
$\langle 300, 300, 300, 30 \rangle$	1.364e+6	7318	4.565e+6	7271

Table 6.2 : Performance comparisons between DisAC3.1 and DisAC9 on the DOMINO problem. Note that m is the number of constraints and p is the number of agents used by both the algorithms.

Instances $\langle n, d, m, p \rangle$	DisAC3.1 #NCCCs	DisAC3.1 #Msgs	DisAC9 #NCCCs	DisAC9 #Msgs
SCEN#01 $\langle 916, 44, 5548, 35 \rangle$	4.411e+5	5690	1.123e+6	5223
SCEN#11 $\langle 680, 44, 4103, 30 \rangle$	2.762e+6	8703	2.292e+7	9132
GRAPH#09 $\langle 916, 44, 5246, 35 \rangle$	4.735e+5	5350	1.391e+6	5378
GRAPH#10 $\langle 680, 44, 3907, 30 \rangle$	7.156e+5	6467	2.007e+6	6718
GRAPH#14 $\langle 916, 44, 4638, 35 \rangle$	4.094e+5	4779	1.227e+6	5137

Table 6.3 : Performance comparisons between DisAC3.1 and DisAC9 on the Radio Link Frequency Assignment Problem. Note that m is the number of constraints and p is the number of agents used by both the algorithms.

and (4) the tightness of constraints $\beta = M/d^2$, where M is the number of allowed tuples of a binary constraint.

Following the convention of [8, 12, 52, 98], experimental results with random instances are presented as figures where the x-axis represents the constraint tightness. The experiments presented here were performed on four groups of instances, where group A has instances with $n = 20, d = 10, \rho = 0.3$, group B has instances with $n = 40, d = 20, \rho = 0.3$, group C has instances with $n = 20, d = 10, \rho = 0.6$ and group D has instances with $d = 10, \rho = 0.3, \beta = 0.5$. Note that we have conducted the experiments on a large number of different classes of CSPs and the results were similar. The results reported here present a summary of all the results we have observed.

Experimental results with groups A, B, C and D are summarized in Figure 6.7a, Figure 6.7b, Figure 6.7c and Figure 6.7d, respectively, where every data point in each graph is obtained by averaging the results of 20 instances. We can observe from the figures that, if the number of agents is moderate (e.g. $n/5$), then DisAC3.1 only requires approximately half #NCCCs of DisAC9, but DisAC3.1 sends slightly more messages than DisAC9. We also notice that if too many agents are used (e.g. n), performances of the distributed algorithms become worse. This phenomenon owes to the fact that too many agents would cause overloaded inter-agent communications and thus decrease the concurrency of the distributed algorithms.

In the results of groups A and B, we observe that #NCCCs and #Msgs phase transitions happen at tightness = 0.7. When constraint tightness is larger than 0.8, #Msgs drops down to zero, because with increasing constraint tightness the need for deleting domain values when enforcing AC dramatically decreases, which results in decrease of inter-agent communications.

We notice that the constraint tightness at which #NCCCs and #Msgs phase transitions happen is lower for group C than groups A and B: In group C the phase transitions are at tightness=0.6, whereas in groups A and B the phase transitions are at tightness=0.7. This phenomenon owes to the fact that higher network density implies higher number of constraints, which in turn increases the number of constraint checks and inter-agent communications. Therefore, when the network density is higher as is the case of group C, then the distributed algorithms require lower constraint tightness to reach the phase transitions.

Furthermore, we observe that the distributed algorithms require higher #NCCCs and #Msgs for group B, which is not a surprise considering the networks in group B have larger number of variables and domain size.

We now compare the scalability of DisAC3.1 with DisAC9 (cf. Figure 6.7d). From

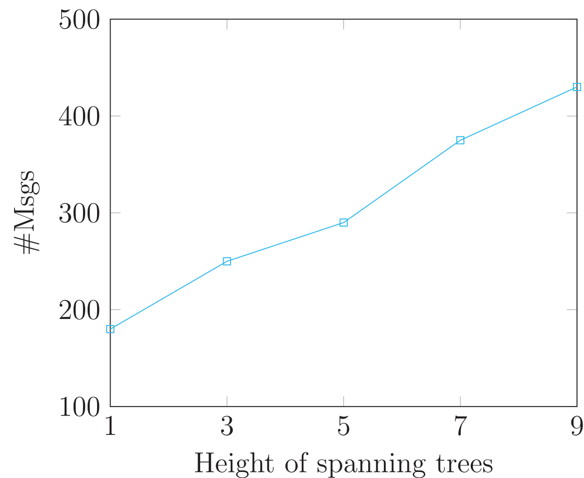


Figure 6.8 : Number of messages sent by agents w.r.t. height of spanning trees. We set $n = 10, d = 10, \rho = 1.0, \beta = 0.5$ and #agent = 10.

the left subfigure, we observe that both algorithms show a linear behaviour w.r.t. #variables, but #NCCCs of DisAC3.1 grows slower than that of DisAC9. We also evaluated Lai and Wu's termination detection algorithm [77] for DisAC3.1, which is based on Dijkstra-Scholten termination detector and has an *optimal* message complexity.[1] In order to compare the performance of our termination detection method with Lai and Wu's termination detection algorithm, we modified DisAC3.1 by replacing our termination detection procedure with Lai and Wu's algorithm.[2] Result shows that all three algorithms show a sub-linear behavior w.r.t. the number of variables. The modified DisAC3.1 requires more agent messages than the other two, and DisAC9 is the best.

The main factor that affects the performance of our termination detection method is the height of spanning trees of agent communication graphs. We evaluated the #Msgs of DisAC3.1 w.r.t. the height of spanning trees. In order to generate span-

[1]Note that Lai and Wu's algorithm assumes fully-connected agent communication graphs, and every agent knows the identifications of all agents in the system.

[2]Note that we do not compare #NCCCs between the modified DisAC3.1 and DisAC3.1, as termination detector is *superimposed* on the underling computation, i.e., it cannot intervene in the underling computation [42] and thus does not affect #NCCCs.

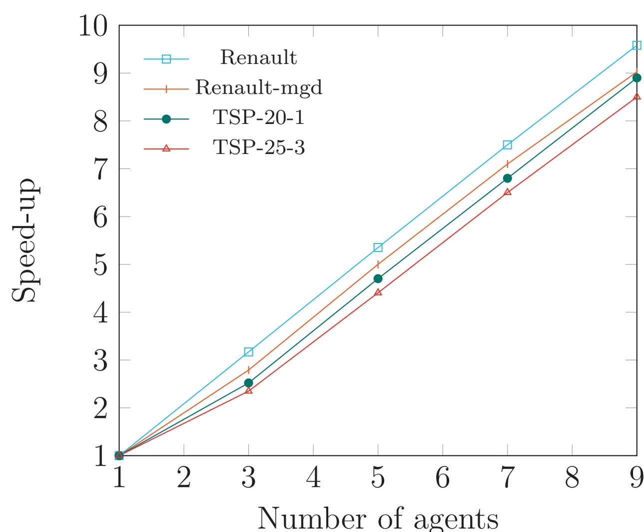


Figure 6.9 : Speed-up vs. number of agents on benchmark problems.

ning trees with height that we need, we set $\rho = 1$ for all random generated CSP instances, so that all agent communication graphs are complete. The result is shown in Figure 6.8, where each data point is averaged over 20 random instances. As expected, the larger the height of spanning trees, the larger the #Msgs required by the algorithm. If the root is fixed, distributed breadth-first search [1] can be used to find the spanning trees with minimum height.

We also consider benchmark problems that are commonly used for the evaluating AC algorithms, i.e., the DOMINO problem [12, 136], which is designed to study the worst case performance of AC3 and the Radio Link Frequency Assignment Problem [11, 12]. [1] Results are summarized in Tables 6.2 and 6.3. In Table 6.2, we can observe that DisAC3.1 only needs 43.2% of #NCCCs of DisAC9 on average, while #Msgs of both algorithms are comparable. Similarly, in Table 6.3, we can observe that DisAC3.1 only needs 30.9% of #NCCCs of DisAC9 on average, while #Msgs of both algorithms are comparable.

[1] Website https://www.cril.univ-artois.fr/~lecoutre/benchmarks.html

6.7.2 DisGAC3.1 vs. GAC2001/3.1

To evaluate our DisGAC3.1 algorithm, we compare it with GAC2001/3.1 by using benchmark problems from the fourth international constraint solver competition.[1] We have extracted instances of the Renault Megane configuration problem (Renault) and the traveling salesman problem (TSP) for experiments, where the Renault instances have arity of 10 and the TSP instances have arity of 3. Results are presented in Figure 6.9. Here, the performance of GAC2001/3.1 is chosen as the baseline (number of agents = 1) and the speed-up[2] is measured relative to the baseline. As one can expect DisGAC3.1 is more efficient than GAC2001/3.1 and we observe that it has a linear speed-up in the number of agents. We can also conclude that the performance of DisAC3.1 increases linearly in the number of agents, since DisAC3.1 is a special case of DisGAC3.1.

6.8 Conclusion

In this chapter we have presented new distributed algorithms DisAC3.1 and DisGAC3.1 for efficient AC and GAC propagations. These algorithms do not assume a complete agent communication graph and release less private information of individual agents when enforcing AC and GAC. More precisely, an agent i only shares information about its communication address, its domain D_u, and its external constraint R_{uv} with another agent j, if the variable v is owned by j.

Our theoretical analysis shows that our algorithms are efficient in both time and space. Our experiments on both randomly generated instances and benchmark problems show that (i) DisAC3.1 requires significantly less number of non-concurrent constraint checks than the state-of-the-art distributed AC algorithm DisAC9, and (ii)

---

[1] Website http://www.cril.univ-artois.fr/CSC09/

[2] In this experiment, we measured the performance of the algorithm using the *simulated time metric* [116] where message latency is simulated by a random time from 0–10ms per message.

the performance of DisGAC3.1 scales linearly in the number of agents, maximizing the merit of the distributed setting.

As technological advancements allow for solving larger problems that are highly interwoven and dependent on each other, efficiency and privacy have become critical requirements. For problems that can be modelled as distributed CSPs, we have shown that DisAC3.1 and DisGAC3.1 are efficient algorithms that can serve as good candidates for search space pruning.

Methods developed in this chapter also can be adapted to solve the multiagent STP (MaSTP) studied in Chapter 5.

Appendix

Proof of Lemma 6.1

Proof. (1) $\Rightarrow$ (2): Given an agent i, the property *isIdle*$[i]$ is set to `true` only in line 40 and line 43. If *isIdle*$[i]$ was set to `true` in line 40, then $s_{ji} = r_{ij}$ for all $j = 1, \ldots, p$ and there is an agent k who sent a "domain update" message m_{ki} with timestamp s_{ki} to agent i, and the corresponding "message sent" message m_{ki}^{sent} from agent k to the root agent has been received. Since $s_{ki} = r_{ik}$, the root agent has also received the message m_{ki}^{utd}, and thus, the root agent has received agent i's first "up to date" message.

The other case, i.e., *isIdle*$[i]$ was set to `true` in line 43, immediately implies that the root agent have received an "up to date" message from agent i and $s_{ji} = r_{ij}$ for all $j = 1, \ldots, p$.

(2) $\Rightarrow$ (3): We prove this by contradiction.

First, suppose there is at least one "domain update" message that

has not been confirmed by the root agent and let m_{ij} be such a message (sent

from agent i to agent j) which bears the *earliest* timestamp s and let m_{ij}^{sent} and m_{ij}^{utd} be the corresponding "message sent" message and "up to date" message, respectively (cf. Section 6.4.2).

We first note that according to our assumption (2) the root agent already received agent i's first "up to date" message m^0 and $s_{ik} = r_{ki}$ for all $k = 1, \ldots, p$. Furthermore, because m_{ij} is the earliest message that has not been confirmed yet and the root agent receives messages in FIFO manner, no other messages from agent i to j sent later than m_{ij} have been confirmed by the root agent. Then it follows that both m_{ij}^{sent} and m_{ij}^{utd} have not been received by the root agent; otherwise, we would have either $s_{ij} > r_{ji}$ or $s_{ij} < r_{ji}$, contradicting our assumption.

Furthermore, agent i must have sent m_{ij} after sending m^0 to the root agent; otherwise, as the root agent receives messages from agent i in a FIFO manner, it would have received m_{ij}^{sent} before receiving m^0. Now that agent i sent m_{ij} after sending m^0, it must have sent m_{ij} when it re-entered the first inner while-loop, i.e., it received a "domain update" message m_{ki} from an agent k after sending m^0. Note that this message m_{ki} is not confirmed either, because the corresponding "up to date" message m_{ki}^{utd} from agent i to the root agent must be sent after m_{ij}^{sent}, but m_{ij}^{sent} has not been received by the root agent. As m_{ki} bears an earlier timestamp than m_{ij}, we have the contradiction that m_{ij} is an unconfirmed "domain update" message with the earliest timestamp. Thus, we can assume that all "domain update" messages have been confirmed.

Now suppose there is at least one agent running the first inner while-loop and let i be such an agent. Then, since agent i sent already its first "up to date" message to the root agent, agent i must have received a new message of type "domain update" so that it could re-enter the first inner while-loop. Let this "domain update" message m_{ki} be from an agent k. Then agent i has not yet sent the corresponding "up to

date" message m_{ki}^{utd} to the root agent, as it has not finished the first inner while-loop. Thus m_{ki} has not been confirmed. This is a contradiction to our assumption that all "domain update" messages have been confirmed.

(3) $\Rightarrow$ (1): We prove this by contradiction.

We note first that the state of *isIdle*$[i]$ ($i = 1, \ldots, p$) will remain fixed in the future, because all "domain update" messages have been confirmed and no agent is running the first inner while-loop, and thus, no new message of type "message sent" or "up to date" will be received by the root agent in the future. Suppose there is an agent i such that *isIdle*$[i]$ = `false`. Then the agent must have received at least one "domain update" message from an agent k, otherwise $s_{ji} = r_{ij} = -\infty$ for all $j = 1, \ldots, p$ and as a consequence *isIdle*$[i]$ had to be set to `true` in line 43 after the root agent received the last "up to date" message of agent i.

Now suppose that m_{ki} is the last message that agent i received, which was sent from agent k, and let m_{ki}^{sent} and m_{ki}^{utd} be the corresponding "message sent" message and "up to date" message, respectively. Then, because m_{ki} has been confirmed, both m_{ki}^{sent} and m_{ki}^{utd} must have been received by the root agent. If m_{ki}^{sent} was received later, then the root agent must set *isIdle*$[i]$ to `true` in line 40, and if m_{ki}^{utd} was received later, the root agent must set *isIdle*$[i]$ to `true` in line 43. Both cases contradict our assumption that *isIdle*$[i]$ = `false`. Hence, *isIdle*$[i]$ = `true` for all $i = 1, \ldots, p$. $\square$

Proof of Lemma 6.2

Proof. We note first that at the beginning of the algorithm we have vacuously $D_u^i \supseteq D_u$, as D_u^i is a copy of D_u.

Now suppose $D_u^i \supseteq D_u$ during the process of the algorithm. Then we note that $D_u^i \not\supseteq D_u$ can only happen in line 17 of the algorithm, where there exists a variable $v \in V_i$ with $R_{uv} \in C_{ji}$ such that $\textsc{Revise}(u, v)$ is true, i.e., there exists $b \in D_u^i$ for

which no $a \in D_v$ exists with $(b, a) \in R_{uv}$ in which case we remove b from D_u^i. Agent i then prompts agent j to replace D_v^j with the content of D_v (line 19). Agent j—*we now switch the perspective and look at the algorithm from agent j's point of view*—then replaces D_v^j with the content of D_v (line 34) and adds among other arcs (u, v) to its queue (line 35), and applies function Revise to (u, v) (line 10). This deletes b from D_u, as there is no $a \in D_v^j$ with $(b, a) \in R_{uv}$. Thus, we have $D_u^i \supseteq D_u$. □

Chapter 7

Conclusion

The research presented in this dissertation is focused on exploring local consistency methods for solving constraint satisfaction problems (CSPs) in several directions.

In the first part we explored centralized local consistency algorithms to identify more general tractable constraint classes and to solve tractable constraint classes more efficiently. Most of our works are related to the class of connected row-convex (CRC) constraints, which can model problems in domains such as temporal reasoning, VLSI design, geometric reasoning, scene labelling as well as logical filtering.

We first generalized the class of CRC constraints to the class of tree-preserving constraints. We proved that enforcing strong PC decides the consistency of a tree-preserving constraint network and, if no inconsistency is detected, transforms the network into a globally consistent constraint network. As an application, we showed that a large subclass of the scene labelling problem can be modelled as tree-preserving constraint networks.

We then characterized CSPs that are solvable with directional PC. We showed that **DPC** (the DPC enforcing algorithm of Dechter and Pearl) decides the constraint satisfaction problem (CSP) of a constraint language if it is complete and has the variable elimination property (VEP). However, we also showed that no complete VEP constraint language can have a domain with more than 2 values. We then presented a simple variant of the **DPC** algorithm, called **DPC***, and showed that the CSP of a constraint language can be decided by **DPC*** if it is closed under a majority operation. In fact, **DPC*** is sufficient for guaranteeing backtrack-free

search for such constraint networks.

In the second part we designed more efficient distributed local consistency algorithms to solve tractable constraint classes and to filter inconsistent tuples for distributed CSP solvers. We presented the first distributed deterministic algorithm for connected row-convex (CRC) constraints. Our distributed partial path consistency algorithm efficiently transforms a CRC constraint network into an equivalent constraint network, where all constraints are minimal (*i.e.*, they are the tightest constraints) and generating all solutions can be done in a backtrack-free manner. When compared with the state-of-the-art distributed algorithm for CRC constraints, which is a randomized one, our algorithm guarantees to generate a solution for satisfiable CRC constraint networks and it is applicable to solve large networks in real distributed systems. The experimental evaluations showed that our algorithm outperforms the state-of-the-art algorithm in both practice and theory.

We then considered the class of simple temporal constraints, which is closely related to the class of CRC constraints and is widely used in temporal planning and scheduling. In fact, discretized simple temporal constraints over finite domains are exactly CRC constraints. We showed that the AC-based approach is sufficient for solving both the simple temporal problem (STP) and Multiagent STP (MaSTP) and provide efficient algorithms for them. As our AC-based approach does not impose new constraints between agents, it does not violate the privacy of the agents and is superior to the state-of-the-art approach to MaSTP. Empirical evaluations on diverse benchmark datasets also showed that our AC-based algorithms for STP and MaSTP are significantly more efficient than existing approaches.

Finally we proposed a new distributed arc-consistency algorithm, called DisAC3.1, which leaks less private information of agents than existing distributed arc-consistency algorithms. In particular, DisAC3.1 uses a novel termination determination mech-

anism, which allows the agents to share domains, constraints and communication addresses only with relevant agents. We further extended DisAC3.1 to DisGAC3.1, which is the first distributed algorithm that enforces generalized arc-consistency on k-ary ($k \geq 2$) constraint satisfaction problems. Theoretical analyses showed that our algorithms are efficient in both time and space. Experiments also demonstrated that DisAC3.1 outperforms the state-of-the-art distributed arc-consistency algorithm and that DisGAC3.1's performance scales linearly in the number of agents.

7.1 Directions for Future Research

The work presented here leaves room for additional improvements that can be pursued in the future, some of which has been already mentioned in the respective chapters.

Recently, Bulatov [16] and Zhuk [138] have confirmed the long-standing *CSP dichotomy conjecture*, which was first proposed in [45]. The conjecture states that, given any set of constraint relations Γ, the set of all possible CSPs defined over Γ, C_Γ, is either in P or NP-complete. It would be intriguing to develop efficient methods to decide whether C_Γ for a given set of relations Γ is in P, and if the answer is positive, then develop efficient algorithms to solve all problems in C_Γ.

Although Kozik had shown that singleton arc-consistency (SAC) algorithms solve the same family of problems that are solvable with local consistency [71]. However, it remains unclear whether backtrack-free search can be used to extract a solution for such a problem after enforcing SAC. It would be interesting to see whether it is possible to develop an algorithm which can extract a solution in a backtrack-free manner.

It would be also interesting to see how the results in Chapter 5 can be used for solving the general disjunctive temporal problems [115]. Potential extensions also

include adapting our AC algorithms to incremental algorithms for the STP [105], dynamic situations [102] and uncertainty [120].

Bibliography

[1] Awerbuch, B., Gallager, R.: A new distributed algorithm to find breadth first search trees. IEEE Transactions on Information Theory 33(3), 315–322 (1987)

[2] Barták, R., Morris, R.A., Venable, K.B.: An introduction to constraint-based temporal reasoning. Synthesis Lectures on Artificial Intelligence and Machine Learning 8(1), 1–121 (2014)

[3] Barták, R., Salido, M.A.: Constraint satisfaction for planning and scheduling problems. Constraints 16(3), 223 (2011)

[4] Barto, L.: The collapse of the bounded width hierarchy. Journal of Logic and Computation 26(3), 923–943 (2016)

[5] Barto, L., Kozik, M.: Constraint satisfaction problems solvable by local consistency methods. Journal of the ACM 61(1), 3:1–3:19 (2014)

[6] Baudot, B., Deville, Y.: Analysis of distributed arc-consistency algorithms. Tech. Rep. 97-07, Université catholique de Louvain (1997)

[7] van Beek, P., Dechter, R.: On the minimality and global consistency of row-convex constraint networks. Journal of the ACM 42(3), 543–561 (1995)

[8] Bessière, C.: Arc-consistency and arc-consistency again. Artificial Intelligence 65(1), 179 – 190 (1994)

[9] Bessière, C., Cardon, S., Debruyne, R., Lecoutre, C.: Efficient algorithms for singleton arc consistency. Constraints 16(1), 25–53 (2011)

[10] Bessière, C., Debruyne, R.: Theoretical analysis of singleton arc consistency and its extensions. Artificial Intelligence 172(1), 29–41 (2008)

[11] Bessière, C., Freuder, E.C., Régin, J.C.: Using constraint metaknowledge to reduce arc consistency computation. Artificial Intelligence 107(1), 125 – 148 (1999)

[12] Bessière, C., Régin, J., Yap, R.H.C., Zhang, Y.: An optimal coarse-grained arc consistency algorithm. Artificial Intelligence 165(2), 165–185 (2005)

[13] Bliek, C., Sam-Haroud, D.: Path consistency on triangulated constraint graphs. In: IJCAI. pp. 456–461 (1999)

[14] Boerkoel Jr, J.C., Durfee, E.H.: Distributed reasoning for multiagent simple temporal problems. Journal of Artificial Intelligence Research 47, 95–156 (2013)

[15] Boros, E., Hammer, P.L., Minoux, M., Rader, D.J.: Optimal cell flipping to minimize channel density in vlsi design and pseudo-boolean optimization. Discrete Applied Mathematics 90(1), 69–88 (1999)

[16] Bulatov, A.A.: A dichotomy theorem for nonuniform csps. In: 58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017. pp. 319–330 (2017)

[17] Bulatov, A.A., Jeavons, P.: An algebraic approach to multi-sorted constraints. In: CP. pp. 183–198 (2003)

[18] Carbonnel, C., Cooper, M.C.: Tractability in constraint satisfaction problems: A survey. Constraints 21(2), 115–144 (2016)

[19] Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states of distributed systems. ACM Transactions on Computer Systems (TOCS) 3(1),

63–75 (1985)

[20] Chandy, K.M., Misra, J.: A paradigm for detecting quiescent properties in distributed computations. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems. pp. 325–341. Springer Berlin Heidelberg (1985)

[21] Chandy, K.M., Misra, J.: How processes learn. Distributed Computing 1(1), 40–52 (1986)

[22] Chang, E.J.H.: Echo Algorithms: Depth Parallel Operations on General Graphs. IEEE Transactions on Software Engineering SE-8(4), 391–401 (1982)

[23] Chen, H., Dalmau, V., Grußien, B.: Arc consistency and friends. Journal of Logic and Computation 23(1), 87–108 (2011)

[24] Chmeiss, A., Jégou, P.: Efficient path-consistency propagation. International Journal on Artificial Intelligence Tools 7(02), 121–142 (1998)

[25] Clowes, M.B.: On seeing things. Artificial Intelligence 2(1), 79–116 (1971)

[26] Cohen, D., Jeavons, P.: The complexity of constraint languages. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, pp. 245–280. Elsevier, Amsterdam (2006)

[27] Conitzer, V., Derryberry, J., Sandholm, T.: Combinatorial auctions with structured item graphs. In: AAAI. pp. 212–218 (2004)

[28] Conry, S.E., Kuwabara, K., Lesser, V.R., Meyer, R.A.: Multistage negotiation for distributed constraint satisfaction. IEEE Transactions on Systems, Man, and Cybernetics 21(6), 1462–1477 (1991)

[29] Cooper, M.C.: Linear-time algorithms for testing the realisability of line drawings of curved objects. Artificial Intelligence 108(1), 31–67 (1999)

[30] Cooper, M.C., Jeavons, P.G., Salamon, A.Z.: Hybrid tractable csps which generalize tree structure. In: ECAI. pp. 530–534 (2008)

[31] Cooper, M.C., Mouelhi, A.E., Terrioux, C., Zanuttini, B.: On broken triangles. In: CP. pp. 9–24 (2014)

[32] Cooper, P.R., Swain, M.J.: Arc consistency: parallelism and domain dependence. Artificial Intelligence 58(1-3), 207–235 (1992)

[33] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, 3rd edn. (2009)

[34] Cox, J., Durfee, E.: Efficient and distributable methods for solving the multiagent plan coordination problem. Multiagent and Grid Systems 5(4), 373–408 (2009)

[35] Debruyne, R., Bessière, C.: Some practicable filtering techniques for the constraint satisfaction problem. In: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, Nagoya, Japan, August 23-29. pp. 412–417. Morgan Kaufmann, Burlington (1997)

[36] Dechter, R.: From local to global consistency. Artificial Intelligence 55(1), 87–107 (1992)

[37] Dechter, R.: Constraint processing. Morgan Kaufmann (2003)

[38] Dechter, R., Meiri, I., Pearl, J.: Temporal constraint networks. Artificial intelligence 49(1-3), 61–95 (1991)

[39] Dechter, R., Pearl, J.: Network-based heuristics for constraint-satisfaction problems. Artificial Intelligence 34(1), 1–38 (1987)

[40] Deville, Y., Barette, O., van Hentenryck, P.: Constraint satisfaction over connected row-convex constraints. Artificial Intelligence 109(1), 243–271 (1999)

[41] Dijkstra, E.W., Feijen, W.H., Van Gasteren, A.M.: Derivation of a termination detection algorithm for distributed computations. In: Control Flow and Data Flow: concepts of distributed programming, pp. 507–512. Springer (1986)

[42] Dijkstra, E.W., Scholten, C.S.: Termination detection for diffusing computations. Information Processing Letters 11(1), 1–4 (1980)

[43] Eriksen, O., Skyum, S.: Symmetric distributed termination. DAIMI Report Series 14(189) (1985)

[44] Faltings, B., Léauté, T., Petcu, A.: Privacy guarantees through distributed constraint satisfaction. In: 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology. vol. 2, pp. 350–358 (2008)

[45] Feder, T., Vardi, M.Y.: The computational structure of monotone monadic snp and constraint satisfaction: A study through datalog and group theory. SIAM Journal on Computing 28(1), 57–104 (1998)

[46] Fikes, R.E.: Ref-arf: A system for solving problems stated as procedures. Artificial Intelligence 1(1), 27 – 120 (1970)

[47] Freuder, E.C.: Synthesizing constraint expressions. Communications of the ACM 21(11), 958–966 (1978)

[48] Freuder, E.C.: A sufficient condition for backtrack-free search. Journal of the ACM 29(1), 24–32 (1982)

[49] Grinshpoun, T.: When you say (DCOP) privacy, what do you mean? - categorization of DCOP privacy and insights on internal constraint privacy. In: ICAART 2012 - Proceedings of the 4th International Conference on Agents and Artificial Intelligence, Volume 1 - Artificial Intelligence, Vilamoura, Algarve, Portugal, 6-8 February, 2012. pp. 380–386 (2012)

[50] Grinshpoun, T., Tassa, T.: P-syncbb: A privacy preserving branch and bound DCOP algorithm. Journal of Artificial Intelligence Research 57(1), 621–660 (2016)

[51] Gu, J., Sosic, R.: A parallel architecture for constraint satisfaction. In: International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems. pp. 229–237 (1991)

[52] Hamadi, Y.: Optimal distributed arc-consistency. Constraints 7(3-4), 367–385 (2002)

[53] Han, C.C., Lee, C.H.: Comments on Mohr and Henderson's path consistency algorithm. Artificial Intelligence 36(1), 125 – 130 (1988)

[54] Hassine, A.B., Ghedira, K.: How to establish arc-consistency by reactive agents. In: Proceedings of the15th European Conference on Artificial Intelligence. pp. 156–160 (2002)

[55] Hassine, A.B., Ghedira, K., Ho, T.B.: New distributed filtering-consistency approach to general networks. In: Proceedings of the 17th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems. pp. 708–717 (2004)

[56] Huang, S.T.: Detecting termination of distributed computations by external agents. In: The 9th International Conference on Distributed Computing Systems. pp. 79–84 (1989)

[57] Hubbe, P.D., Freuder, E.C.: An efficient cross product representation of the constraint satisfaction problem search space. In: Proceedings of the Tenth National Conference on Artificial Intelligence. pp. 421–427 (1992)

[58] Huffman, D.A.: Impossible objects as nonsense sentences. Machine Intelligence 6(1), 295–323 (1971)

[59] Huhns, M.N., Bridgeland, D.M.: Multiagent truth maintenance. IEEE Transactions on Systems, Man, and Cybernetics 21(6), 1437–1445 (1991)

[60] Hunsberger, L.: Algorithms for a temporal decoupling problem in multi-agent planning. In: Eighteenth National Conference on Artificial Intelligence. pp. 468–475. AAAI Press, Menlo Park, CA, USA (2002)

[61] Jeavons, P., Cohen, D., Cooper, M.C.: Constraints, consistency and closure. Artificial Intelligence 101(1), 251 – 265 (1998)

[62] Jeavons, P., Cohen, D.A., Gyssens, M.: Closure properties of constraints. Journal of the ACM 44(4), 527–548 (1997)

[63] Jeavons, P.G., Cooper, M.C.: Tractable constraints on ordered domains. Artificial Intelligence 79(2), 327–339 (1995)

[64] Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of computer computations, pp. 85–103. Springer (1972)

[65] Kirousis, L.M.: Effectively labelling planar projections of polyhedra. IEEE Transactions on Pattern Analysis and Machine Intelligence 12(2), 123–130 (1990)

[66] Kirousis, L.M., Papadimitriou, C.H.: The complexity of recognizing polyhedral scenes. In: FOCS. pp. 175–185 (1985)

[67] Kjærulff, U.: Triangulation of graphs – algorithms giving small total state space. Tech. rep., Aalborg University (1990)

[68] Kong, S., Li, S., Li, Y., Long, Z.: On tree-preserving constraints. In: Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming, Cork, Ireland, 31 August - 4 September. pp. 244–261. Springer, Berlin (2015)

[69] Kong, S., Li, S., Li, Y., Long, Z.: On tree-preserving constraints. Annals of Mathematics and Artificial Intelligence 81(3-4), 241–271 (2017)

[70] Korte, B., Vygen, J.: Combinatorial Optimization, Algorithms and Combinatorics, vol. 21. Springer Berlin Heidelberg (2012)

[71] Kozik, M.: Weak consistency notions for all the CSPs of bounded width. In: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, New York, USA. pp. 633–641. ACM, New York (2016)

[72] Kumar, T.K.S.: On geometric CSPs with (near)-linear domains and max-distance constraints. In: Workshop on Modelling and Solving Problems with Constraints. pp. 125–138 (2004)

[73] Kumar, T.K.S.: On the tractability of restricted disjunctive temporal problems. In: ICAPS. pp. 110–119 (2005)

[74] Kumar, T.K.S.: Simple randomized algorithms for tractable row and tree convex constraints. In: AAAI. pp. 74–79 (2006)

[75] Kumar, T.K.S., Nguyen, D.T., Yeoh, W., Koenig, S.: A simple polynomial-time randomized distributed algorithm for connected row convex constraints. In: AAAI. pp. 2308–2314 (2014)

[76] Kumar, T.K.S., Russell, S.J.: On some tractable cases of logical filtering. In: ICAPS. pp. 83–92 (2006)

[77] Lai, T.H., Wu, L.F.: An (n -1)-resilient algorithm for distributed termination detection. IEEE Transaction on Parallel Distributed System 6(1), 63–78 (1995)

[78] Lam, E., Hentenryck, P.V.: A branch-and-price-and-check model for the vehicle routing problem with location congestion. Constraints 21(3), 394–412 (2016)

[79] Léauté, T., Faltings, B.: Protecting privacy through distributed computation in multi-agent decision making. Journal of Artificial Intelligence Research 47, 649–695 (2013)

[80] Léauté, T., Ottens, B., Szymanek, R.: Frodo 2.0: An open-source framework for distributed constraint optimization. In: Proceedings of the IJCAI Distributed Constraint Reasoning Workshop. pp. 160–164 (2009)

[81] Lecoutre, C., Cardon, S., Vion, J.: Conservative dual consistency. In: AAAI. pp. 237–242 (2007)

[82] Lecoutre, C., Cardon, S., Vion, J.: Path consistency by dual consistency. In: CP. pp. 438–452 (2007)

[83] Lecoutre, C., Cardon, S., Vion, J.: Second-order consistencies. Journal of Artificial Intelligence Research 40, 175–219 (2011)

[84] Li, S., Liu, W., Wang, S.: Qualitative constraint satisfaction problems: An extended framework with landmarks. Artificial Intelligence 201, 32–58 (2013)

[85] Liu, J.S., Sycara, K.P.: Multiagent coordination in tightly coupled task scheduling. In: Proceedings of the Second International Conference on Multi-Agent Systems. pp. 181–188 (1996)

[86] Long, Z., Li, S.: On distributive subalgebras of qualitative spatial and temporal calculi. In: COSIT 2015. pp. 354–374 (2015)

[87] Long, Z., Sioutis, M., Li, S.: Efficient path consistency algorithm for large qualitative constraint networks. In: IJCAI 2016,. pp. 1202–1208 (2016)

[88] Lynch, N.A.: Distributed algorithms. Morgan Kaufmann (1996)

[89] Mackworth, A.K.: Consistency in networks of relations. Artificial intelligence 8(1), 99–118 (1977)

[90] Mackworth, A.K.: On reading sketch maps. In: Proceedings of the 5th International Joint Conference on Artificial Intelligence. pp. 598–606 (1977)

[91] Mackworth, A.K., Freuder, E.C.: The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. Artificial Intelligence 25(1), 65 – 74 (1985)

[92] Maheswaran, R.T., Pearce, J.P., Bowring, E., Varakantham, P., Tambe, M.: Privacy loss in distributed constraint reasoning: A quantitative framework for analysis and its applications. Autonomous Agents and Multi-Agent Systems 13(1), 27–60 (2006)

[93] Maruyama, H.: Structural disambiguation with constraint propagation. In: ACL. pp. 31–38 (1990)

[94] Mason, C.L., Johnson, R.R.: Datms: A framework for distributed assumption based reasoning. Tech. rep., Lawrence Livermore National Lab., CA (USA) (1988)

[95] Matocha, J., Camp, T.: A taxonomy of distributed termination detection algorithms. Journal of Systems and Software 43(3), 207–221 (1998)

[96] Mattern, F.: Algorithms for distributed termination detection. Distributed computing 2(3), 161–175 (1987)

[97] Mayo, J., Kearns, P.: Distributed termination detection with roughly synchronized clocks. Information Processing Letters 52(2), 105–108 (1994)

[98] Meisels, A., Zivan, R.: Asynchronous forward-checking for discsps. Constraints 12(1), 131–150 (2007)

[99] Miyashiro, R., Matsui, T.: A polynomial-time algorithm to find an equitable home–away assignment. Operations Research Letters 33(3), 235–241 (2005)

[100] Mohr, R., Henderson, T.C.: Arc and path consistency revisited. Artificial Intelligence 28(2), 225 – 233 (1986)

[101] Montanari, U.: Networks of constraints: Fundamental properties and applications to picture processing. Information sciences 7, 95–132 (1974)

[102] Morris, P., Muscettola, N., Vidal, T.: Dynamic control of plans with temporal uncertainty. In: Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 1. pp. 494–499. IJCAI'01, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)

[103] Nguyen, T., Deville, Y.: A distributed arc-consistency algorithm. Science of Computer Programming 30(1-2), 227–250 (1998)

[104] Planken, L., de Weerdt, M., van der Krogt, R.: P^{3}C: A new algorithm for the simple temporal problem. In: Proceedings of the Eighteenth International Conference on International Conference on Automated Planning and Scheduling. pp. 256–263. ICAPS'08, AAAI Press, Sydney, Australia (2008)

[105] Planken, L., de Weerdt, M., Yorke-Smith, N.: Incrementally solving stns by enforcing partial path consistency. In: Proceedings of the Twentieth International Conference on International Conference on Automated Planning and Scheduling. pp. 129–136. ICAPS'10, AAAI Press (2010)

[106] Planken, L.R., de Weerdt, M.M., van der Krogt, R.P.: Computing all-pairs shortest paths by leveraging low treewidth. Journal of Artificial Intelligence Research 43, 353–388 (2012)

[107] Raynal, M.: Distributed Algorithms for Message-Passing Systems. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

[108] Samal, A., Henderson, T.: Parallel consistent labeling algorithms. International Journal of Parallel Programming 16(5), 341–364 (1987)

[109] Savaux, J., Vion, J., Piechowiak, S., Mandiau, R., Matsui, T., Hirayama, K., Yokoo, M., Elmane, S., Silaghi, M.: DisCSPs with privacy recast as planning problems for self-interested agents. In: 2016 IEEE/WIC/ACM International Conference on Web Intelligence (WI). pp. 359–366 (2016)

[110] Shavit, N., Francez, N.: A new approach to detection of locally indicative stability. In: International Colloquium on Automata, Languages, and Programming. pp. 344–358. Springer (1986)

[111] Shostak, R.: Deciding linear inequalities by computing loop residues. Journal of the ACM 28(4), 769–779 (1981)

[112] Silaghi, M.c.: A comparison of distributed constraint satisfaction approaches with respect to privacy. In: DCR. Citeseer (2002)

[113] Singh, M.: Path consistency revisited. International Journal on Artificial Intelligence Tools 5(1-2), 127–142 (1996)

[114] Sioutis, M., Long, Z., Li, S.: Efficiently reasoning about qualitative constraints through variable elimination. In: Proceedings of the 9th Hellenic Conference on Artificial Intelligence, Thessaloniki, Greece,18-20 May. pp. 1:1–1:10. ACM, New York (2016)

[115] Stergiou, K., Koubarakis, M.: Backtracking algorithms for disjunctions of temporal constraints. Artificial Intelligence 120(1), 81–117 (2000)

[116] Sultanik, E.A., Lass, R.N., Regli, W.C.: Dcopolis: A framework for simulating and deploying distributed constraint optimization algorithms. In: Proceedings of the Workshop on Distributed Constraint Reasoning (2007)

[117] Sycara, K., Roth, S.F., Sadeh, N., Fox, M.S.: Distributed constrained heuristic search. IEEE Transactions on Systems, Man, and Cybernetics 21(6), 1446–1461 (1991)

[118] Tarjan, R.E.: Data Structures and Network Algorithms. No. 44 in CBMS-NSF Regional Conference Series in Applied Mathematics, Society for Industrial and Applied Mathematics (1983)

[119] Topor, R.W.: Termination detection for distributed computations. Information Processing Letters 18(1), 33 – 36 (1984)

[120] Venable, K.B., Yorke-Smith, N.: Disjunctive temporal planning with uncertainty. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence. pp. 1721–1722 (2005)

[121] Venkatesan, S.: Reliable protocols for distributed termination detection. IEEE Transactions on Reliability 38(1), 103–110 (1989)

[122] Vidal, V., Geffner, H.: Branching and pruning: An optimal temporal pocl planner based on constraint programming. Artificial Intelligence 170(3), 298 – 335 (2006)

[123] Wallace, R.J.: Why AC-3 is almost always better than AC-4 for establishing arc consistency in csps. In: IJCAI. pp. 239–245 (1993)

[124] Wallace, R.J., Freuder, E.C.: Constraint-based reasoning and privacy/efficiency tradeoffs in multi-agent problem solving. Artificial Intelligence 161(1), 209 – 227 (2005)

[125] Waltz, D.L.: Generating semantic descriptions from drawings of scenes with shadows. Tech. Rep. AITR-271, MIT (1972)

[126] Wilcox, R., Shah, J.: Optimization of multi-agent workflow for human-robot collaboration in assembly manufacturing. In: Infotech@Aerospace 2012. American Institute of Aeronautics and Astronautics (2012)

[127] Xu, L., Choueiry, B.Y.: A new efficient algorithm for solving the simple temporal problem. In: TIME. pp. 212–222 (2003)

[128] Yokoo, M., Durfee, E.H., Ishida, T., Kuwabara, K.: The distributed constraint satisfaction problem: Formalization and algorithms. IEEE Transactions on Knowledge and Data Engineering 10(5), 673–685 (1998)

[129] Yokoo, M., Hirayama, K.: Algorithms for distributed constraint satisfaction: A review. Autonomous Agents and Multi-Agent Systems 3(2), 185–207 (2000)

[130] Yokoo, M., Ishida, T., Durfee, E.H., Kuwabara, K.: Distributed constraint satisfaction for formalizing distributed problem solving. In: Proceedings of the 12th International Conference on Distributed Computing Systems. pp. 614–621 (1992)

[131] Yokoo, M., Suzuki, K., Hirayama, K.: Secure distributed constraint satisfaction: reaching agreement without revealing private information. Artificial Intelligence 161(1), 229 – 245 (2005)

[132] Zhang, Y., Mackworth, A.K.: Parallel and distributed algorithms for finite constraint satisfaction problems. In: Proceedings of the 3th IEEE Symposium on Parallel and Distributed Processing. pp. 394–397 (1991)

[133] Zhang, Y., Freuder, E.C.: Tractable tree convex constraint networks. In: AAAI. pp. 197–203 (2004)

[134] Zhang, Y., Freuder, E.C.: Properties of tree convex constraints. Artificial Intelligence 172(12-13), 1605–1612 (2008)

[135] Zhang, Y., Marisetti, S.: Solving connected row convex constraints by variable elimination. Artificial Intelligence 173(12), 1204–1219 (2009)

[136] Zhang, Y., Yap, R.H.C.: Making AC-3 an optimal algorithm. In: Proceedings of the 17th International Joint Conference on Artificial Intelligence. pp. 316–321 (2001)

[137] Zhang, Y., Yap, R.H.C.: Consistency and set intersection. In: IJCAI. pp. 263–270 (2003)

[138] Zhuk, D.: A proof of CSP dichotomy conjecture. In: 58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017. pp. 331–342 (2017)