ON PATH CONSISTENCY FOR
BINARY CONSTRAINT SATISFACTION PROBLEMS

by

Christopher G. Reeson

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Berthe Y. Choueiry

Lincoln, Nebraska

December, 2016

ON PATH CONSISTENCY FOR
BINARY CONSTRAINT SATISFACTION PROBLEMS

Christopher G. Reeson, M. S.
University of Nebraska, 2016

Adviser: Berthe Y. Choueiry

Constraint satisfaction problems (CSPs) provide a flexible and powerful framework for modeling and solving many decision problems of practical importance. Consistency properties and the algorithms for enforcing them on a problem instance are at the heart of Constraint Processing and best distinguish this area from other areas concerned with the same combinatorial problems. In this thesis, we study path consistency (PC) and investigate several algorithms for enforcing it on binary finite CSPs. We also study algorithms for enforcing consistency properties that are related to PC but are stronger or weaker than PC.

We identify and correct errors in the literature and settle an open question. We propose two improvements that we apply to the well-known algorithms PC-8 and PC-2001, yielding PC-8$^+$ and PC-2001$^+$. Further, we propose a new algorithm for enforcing partial path consistency, σ-Δ-PPC, which generalizes features of the well-known algorithms DPC and PPC. We evaluate over fifteen different algorithms on both benchmark and randomly generated binary problems to empirically demonstrate the effectiveness of our approach.

ACKNOWLEDGMENTS

I would like to thank my adviser, Professor Choueiry, without whom this work would not have been possible. She went above and beyond to make herself available to complete this work. Her insight and advise have been essential, and I very much appreciate her efforts.

I would also like to thank Robert Woodward who created much of the infrastructure needed to run the empirical evaluations. His help troubleshooting problems as they arose was invaluable.

Thanks are also due to the members of my examination committee, Professor Wei and Professor Revesz. Also, Tony Schneider, who has already taken the first step towards implementing these ideas in search.

Finally, I would like to thank my parents for their support.

This research was partially supported by a National Science Foundation (NSF) grant No. RI-111795 and No. RI-1619344. Experiments were conducted on the equipment of the Holland Computing Center at UNL.

Contents

Contents **iv**

List of Figures **viii**

List of Tables **xi**

1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 3

1.3 Outline of Thesis 4

2 Background **5**

2.1 Constraint Satisfaction Problem: Problem Definition 5

2.2 Solving CSPs 7

2.3 Graphical Representation 8

2.4 Global Consistency Properties 9

2.5 Local Consistency Properties 10

2.5.1 Basic Local Consistency Properties for Binary CSPs 11

2.5.1.1 Arc Consistency (AC) 11

2.5.1.2 Path Consistency (PC) 11

2.5.1.3 k-Consistency and Strong k-Consistency 13
2.5.1.4 Directional Path Consistency (DPC) 14
2.5.2 Other Local Consistency Properties for Binary CSPs 15
2.5.2.1 Conservative Path Consistency (CPC) 15
2.5.2.2 Partial Path Consistency (PPC) 15
2.5.2.3 Singleton Arc Consistency (SAC) 16
2.5.2.4 Dual Consistency (DC) and Conservative Dual Consistency (CDC) 17
2.5.3 Comparing Local Consistency Properties 18
2.6 Algorithms for Local Consistency 19
2.7 An Algorithm for Arc Consistency: AC-2001 20
2.8 Algorithms Related to Path Consistency 23
2.8.1 Basic Path Consistency Algorithm (PC-2) 27
2.8.2 Directional Path Consistency (DPC) 30
2.8.3 Iterating Over Variables' Domains (PC-8) 32
2.8.4 Introducing Support Bookkeeping (PC-2001) 35
2.8.5 Partial Path Consistency (BSH-PPC) 37
2.8.6 Strong Conservative Dual Consistency (sCDC1) 40
2.8.7 Strong Dual Consistency (sDC2) 43
2.9 Path Consistency Algorithms in Practice 46

3 Gaps in the Literature **48**
3.1 On the Definition of PPC 48
3.2 CPC and PPC on Triangulated Graphs 50
3.2.1 Incorrect Example 50
3.2.2 Incorrect Proposition 51

3.3 Imprecision in the Pseudocode of BSH-PPC 51
3.4 Comparing PC and PPC 53

4 New Algorithms for (Partial) Path Consistency 56

4.1 Correcting BSH-PPC (PPC+AP) 57
4.2 Exploiting Triangles (ΔPPC) 60
4.3 σ-ΔPPC 63
4.3.1 Idea and Data Structures 63
4.3.2 The σ-ΔPPC Algorithm 64
4.3.3 Support Structures in σ-ΔPPC 67
4.3.4 Reflections on PPC 69
4.4 PC-8^{+} and PC-2001^{+} 69
4.4.1 PC-8-Flag 70
4.4.2 PC-8-Ordering 72
4.4.3 PC-8^{+} 74
4.5 Time and Space Complexities 75

5 Empirical Evaluations 78

5.1 Benchmark Problems 78
5.1.1 Problem Characteristics 78
5.1.2 Experimental Set-Up 81
5.1.3 Pre-processing with Arc Consistency 83
5.1.4 Propagation Queue: Edges versus Triangles 85
5.1.5 Propagating Along a PEO 86
5.1.6 Comparing Types of Supports 86
5.1.7 Benefits of Using Supports 88
5.1.8 Improving PC-8 88

5.1.9 Improving PC-2001 89

5.1.10 Comparing Main Algorithms 90

5.2 Randomly Generated Problems 92

5.2.1 Problem Characteristics 92

5.2.2 Results 93

5.2.2.1 $\langle 50, 25, t, 20\% \rangle$ 93

5.3 Discussion 97

6 Conclusions and Future Work **99**

A Data Structures **101**

B Results of Benchmark Problems **104**

B.1 Pre-processing with Arc Consistency 104

B.2 Propagation Queue: Edges versus Triangles 117

B.3 Propagating Along a PEO 119

B.4 Comparing Types of Supports 121

B.5 Benefits of Using Supports 123

B.6 Improving PC-8 126

B.7 Improving PC-2001 128

B.8 Comparing Main Algorithms 130

C Results of Randomly Generated Problems **132**

C.1 $\langle 50, 25, t, 40\% \rangle$ 132

C.2 $\langle 50, 50, t, 20\% \rangle$ 137

C.3 $\langle 50, 50, t, 40\% \rangle$ 141

Bibliography **145**

List of Figures

2.1 A simple, binary CSP with three variables . . . 8

2.2 The constraint network of a simple binary CSP and a PEO of this network . . 9

2.3 Two simple CSPs that are PC, but not AC . . . 13

2.4 Illustrating DPC: for every tuple $(a, b) \in R_{i,j}$ (the dashed line), there is a value $c \in D_k$ so that $(a, c) \in R_{i,k}$ and $(b, c) \in R_{j,k}$ (the solid lines) . . . 14

2.5 Partial order of some consistency properties [Lecoutre *et al.*, 2011] . . . 19

2.6 A three-variable example . . . 22

2.7 Processing $\mathcal{Q}$, updated values are shown in gray . . . 22

2.8 *Left*: Dashed lines in the constraint graph denote universal constraints. *Right*: The relations of the CSP. PC-2 removes the tuples are shown in gray . . . 28

2.9 A five-variable example . . . 31

2.10 A triangulation of the constraint graph . . . 31

2.11 The ordering used . . . 32

2.12 After running DPC . . . 32

2.13 *Left*: A CSP where the dashed line denotes a universal constraint *Right*: Relations of the CSP with tuples removed by PC-8's lines 1–6 shown in gray . . . 34

2.14 *Left*: Seeking supports for $(2, *) \in R_{1,3}$ in D_2. *Right*: Seeking supports for $(2, *) \in R_{1,4}$ in D_2 . . . 35

2.15 PC-8 seeks supports for $(*, 3) \in R_{1,4}$ in D_2. The tuple $(1, 3)$ is removed from $R_{1,4}$ because no support exists in D_2 35

2.16 Path-consistent relations of the CSP in Figure 2.8 after execution of PC-8 . . 35

2.17 *Left*: A triangulated graph, the dashed line is a universal constraint. *Right*: The relations of the CSP. BSH-PPC removes the tuples shown in gray 38

2.18 *Left*: The CSP before processing. *Middle*: Relations as revised by sCDC1, removed tuples are shown in gray. *Right*: The CSP after processing 42

3.1 A CSP that is PPC because every *closed* graph-path is PC 49

3.2 An incorrect example with the inconsistent two-length path (V_2, V_3, V_5) [Lecoutre *et al.*, 2011] 50

3.3 A triangulated CSP that is CPC, but not PPC 50

3.4 BSH-PPC cannot detect the inconsistency of the instantiation $\{(V_2, 1), (V_1, 1)\}$ along the path (V_2, V_1, V_3, V_1) if it considers only triangles with distinct vertices 52

3.5 A binary CSP that is PPC but not CDC 53

3.6 Enforcing PC uncovers the inconsistency of the CSP, but enforcing PPC yields no filtering 54

3.7 Relations of the CSP in Figure 3.6 54

4.1 A triangulated graph, the dashed line denotes a universal constraint 62

4.2 The relations. Gray tuples are removed by ΔPPC 62

4.3 A triangulated graph, a *peo*, *tri*$[\cdot]$, and *TrianglesEdge*$[\cdot]$ 64

4.4 The support structure for tuples in σ-ΔPPC 67

4.5 Three variables of a CSP 70

5.1 $\langle 50, 25, t, 20\% \rangle$: Comparing the best previously known algorithms 94

5.2 $\langle 50, 25, t, 20\% \rangle$: Comparing the most competitive algorithms 94

5.3 $\langle 50, 25, t, 20\% \rangle$: Improving PC-8 95

5.4 $\langle 50, 25, t, 20\% \rangle$: Improving PC-2001 95

5.5 $\langle 50, 25, t, 20\% \rangle$: Impact of the propagation queue and PEO 96

5.6 $\langle 50, 25, t, 20\% \rangle$: Impact of support structures 96

A.1 The data structures. 101

C.1 $\langle 50, 25, t, 40\% \rangle$: Comparing the best previously known algorithms 134

C.2 $\langle 50, 25, t, 40\% \rangle$: Comparing the most competitive algorithms 134

C.3 $\langle 50, 25, t, 40\% \rangle$: Improving PC-8 135

C.4 $\langle 50, 25, t, 40\% \rangle$: Improving PC-2001 135

C.5 $\langle 50, 25, t, 40\% \rangle$: Impact of the propagation queue and PEO 136

C.6 $\langle 50, 25, t, 40\% \rangle$: Impact of support structures 136

C.7 $\langle 50, 50, t, 20\% \rangle$: Comparing the best previously known algorithms 138

C.8 $\langle 50, 50, t, 20\% \rangle$: Comparing the most competitive algorithms 138

C.9 $\langle 50, 50, t, 20\% \rangle$: Improving PC-8 139

C.10 $\langle 50, 50, t, 20\% \rangle$: Improving PC-2001 139

C.11 $\langle 50, 50, t, 20\% \rangle$: Impact of the propagation queue and PEO 140

C.12 $\langle 50, 50, t, 20\% \rangle$: Impact of support structures 140

C.13 $\langle 50, 50, t, 40\% \rangle$: Comparing the best previously known algorithms 142

C.14 $\langle 50, 50, t, 40\% \rangle$: Comparing the most competitive algorithms 142

C.15 $\langle 50, 50, t, 40\% \rangle$: Improving PC-8 143

C.16 $\langle 50, 50, t, 40\% \rangle$: Improving PC-2001 143

C.17 $\langle 50, 50, t, 40\% \rangle$: Impact of the propagation queue and PEO 144

C.18 $\langle 50, 50, t, 40\% \rangle$: Impact of support structures 144

List of Tables

2.1 Summary of the studied consistency properties 18
2.2 List of the studied algorithms 23
2.3 Properties of the studied algorithms 25
2.4 Data structures and propagation queues for the studied algorithms 26
2.5 PC-2's first 12 steps on the example in Figure 2.8 29

4.1 Summary of the introduced algorithms 56
4.2 Data structures and propagation queues of the introduced algorithms 57
4.3 Properties of the introduced algorithms 76

5.1 Characteristics of the benchmark problems (table 1 of 3) 79
5.2 Characteristics of the benchmark problems (table 2 of 3) 80
5.3 Characteristics of the benchmark problems (table 3 of 3) 81
5.4 Summary of effect of pre-processing with AC 84
5.5 Enforcing AC before DPC 84
5.6 A propagation queue made of edges (AC+PPC+AP) vs triangles (AC+ΔPPC) 85
5.7 Propagating along a PEO: AC+ΔPPC vs AC+σ-ΔPPC 86
5.8 Comparing the two types of support structures 87
5.9 Impact of using support structures 88
5.10 Improvements to PC-8 89

5.11 Improvements to PC-2001 90

5.12 Comparing the best performing algorithms 91

5.13 Comparing CPU times on random problems $\langle 50, 25, t, 20\% \rangle$ 93

B.1 Enforcing arc consistency as a pre-processing step 105

B.2 Enforcing AC before DPC. AC+DPC and DPC complete the same number of instances 106

B.3 Enforcing AC before PC-2. AC+PC-2 completes more instances than PC-2 and consistently saves time 107

B.4 Enforcing AC before PC-8. AC+PC-8 completes more instances than PC-8 and generally saves time 108

B.5 Enforcing AC before PC-8-Ordering. AC+PC-8-Ordering completes more instances than PC-8-Ordering and consistently saves time 109

B.6 Enforcing AC before PC-2001. AC+PC-2001 completes more instances than AC+PC-2001 and consistently saves time 110

B.7 Enforcing AC before PC-2001-Ordering. AC+PC-2001-Ordering completes more instances than PC-2001-Ordering and consistently saves time 111

B.8 Enforcing AC before PPC+AP. Both algorithms complete 1,837 instances but pre-processing with AC generally saves time 112

B.9 Enforcing AC before ΔPPC. Both algorithms complete 1,837 instances but pre-processing with AC generally saves time 112

B.10 Enforcing AC before σ-ΔPPC. Both algorithms complete 1,837 instances but pre-processing with AC generally saves time 113

B.11 Enforcing AC before σ-ΔPPC^{sup2001}. Pre-processing with AC allows us to complete more instances and saves time 114

B.12 Enforcing AC before σ-ΔPPC^{sup}. Pre-processing with AC allows us to complete more instances and generally saves time 115
B.13 Enforcing AC before sDC2. Pre-processing with AC allows us to complete more instances and consistently saves time 116
B.14 AC+PPC+AP vs AC+ΔPPC (table 1 of 2) 117
B.15 AC+PPC+AP vs AC+ΔPPC (table 2 of 2) 118
B.16 Following a PEO: AC+ΔPPC vs AC+σ-ΔPPC (table 1 of 2) 119
B.17 Following a PEO: AC+ΔPPC vs AC+σ-ΔPPC (table 2 of 2) 120
B.18 Comparing the two styles of support structures when no filtering occurs . . . 121
B.19 Comparing the two styles of support structures on benchmarks when filtering occurs 122
B.20 Drawbacks of using support structures on benchmarks with no filtering 123
B.21 Impact of using support structures on benchmarks with filtering (table 1 of 2) 124
B.22 Impact of using support structures on benchmarks with filtering (table 2 of 2) 125
B.23 Assessing improvements to PC-8 (table 1 of 2) 126
B.24 Assessing improvements to PC-8 (table 2 of 2) 127
B.25 Assessing improvements to PC-2001 (table 1 of 2) 128
B.26 Comparing AC+DPC, AC+PC-2001-Ordering, AC+PC-8-Ordering, σ-ΔPPC^{sup}, AC+sDC2, and sCDC1 (table 1 of 2) 130
B.27 Comparing AC+DPC, AC+PC-2001-Ordering, AC+PC-8-Ordering, σ-ΔPPC^{sup}, AC+sDC2, and sCDC1 (table 2 of 2) 131

C.1 Comparing CPU times on random problems $\langle 50, 25, t, 40\% \rangle$ 133
C.2 Comparing CPU times on random problems $\langle 50, 50, t, 20\% \rangle$ 137
C.3 Comparing CPU times on random problems $\langle 50, 50, t, 40\% \rangle$ 141

List of Algorithms

1 AC-2001($\mathcal{P}$) *Adapted from* [Bessière *et al.*, 2005] 20
2 AC-2001-CORE($\mathcal{P}, \mathcal{X}$) *Adapted from* [Bessière *et al.*, 2005] 21
3 REVISE-2001(V_i, V_j) *Adapted from* [Bessière *et al.*, 2005] 21
4 PC-2($\mathcal{P}$) *Adapted from* [Dechter, 2003] 27
5 REVISE-3(V_i, V_j, V_k) *Adapted from* [Dechter, 2003] 28
6 DPC($\mathcal{P}$, *ord*) *Adapted from* [Dechter and Pearl, 1988] 30
7 FILTER-DOM($D_i, \mathcal{C}_i$) 31
8 PC-8($\mathcal{P}$) *Adapted from* [Chmeiss and Jégou, 1998] 33
9 PC-2001($\mathcal{P}$) *Adapted from* [Bessière *et al.*, 2005] 36
10 BSH-PPC($\mathcal{P}$) *Adapted from* [Bliek and Sam-Haroud, 1999] . . . 38
11 sCDC1($\mathcal{P}$) *Adapted from* [Lecoutre *et al.*, 2007a] 40
12 sCDC1-CHECK($\mathcal{P}, V_x$) *Adapted from* [Lecoutre *et al.*, 2007a] . . . 41
13 sDC2($\mathcal{P}$) *Adapted from* [Lecoutre *et al.*, 2007b] 44
14 sDC2-CHECK($\mathcal{P}, V_x$, *count*) *Adapted from* [Lecoutre *et al.*, 2007b] . . 45
15 FC-2001($\mathcal{P}, V_x$) *Adapted from* [Lecoutre *et al.*, 2007b] 45
16 PPC+AP($\mathcal{P}$) 58
17 PROPAGATE-APS($\mathcal{A}_{points}, \mathcal{Q}_{AP}$) 59
18 FILTER-RELS($\mathcal{C}_i, D_i$) 59
19 REVISE-3+AP($V_i, V_j, V_k, \mathcal{A}_{points}$) 60

20 ΔPPC($\mathcal{P}$) 61

21 Revise-Triangle($V_i, V_j, V_k, \mathcal{A}_{points}$) 62

22 Init-Triangle-Vector($\mathcal{P}, peo, tri[\cdot]$) 65

23 σ-ΔPPC($\mathcal{P}, peo$) 66

24 PC-8-Flag($\mathcal{P}$) 71

25 Revise-PC-8-Flag($V_i, V_j, V_k, flag$) 72

26 PC-8-Ordering($\mathcal{P}$) 74

27 PC-8$^+$($\mathcal{P}$) 75

Chapter 1

Introduction

Constraint satisfaction problems (CSPs) provide a flexible and powerful framework for modeling many decision problems of practical importance. In order to find a solution to a CSP, a solver must, in general, incorporate search because CSPs are NP-complete. A solver will usually run consistency algorithms as a pre-processing step and/or during search in order to reduce the cost of the search. Consistency algorithms operate by removing values from the domains of the variables and/or tuples from the relations of the constraints. As a result, they reduce the size of the search space.

In this thesis, we study the recent developments of the algorithms for path consistency (PC) and propose new efficient variations of these algorithms, thus improving on the state of the art.

1.1 Motivation

While consistency algorithms are at the heart of constraint processing, most of the research has focused on developing efficient algorithms for enforcing arc consistency

on binary CSPs and generalized arc consistency (GAC) on non-binary CSPs. This situation remains the case even today with new generic algorithms for GAC being proposed in the main conferences. Algorithms for enforcing path consistency (PC) have been considered too expensive to use in practice. Three main trends have revived our interest in path consistency:

1. First, Bliek and Sam-Haroud [1999] proposed an algorithm for enforcing partial path consistency (PPC) that operates on arbitrary binary finite-constraints. Xu and Choueiry [2003] proposed the ΔSTP algorithm which applied the same idea as DPC to efficiently solve the simple temporal problem. Planken *et al.* [2008] proposed the P^{3}C algorithm, which improved the worst case performance of ΔSTP. Long *et al.* [2016] proposed DPC+, which specialized P^{3}C for distributive spacio-temporal constraints. They generalized the scalar operators to relational ones (i.e., addition and minimum to composition and intersection, respectively). ΔSTP and its improved versions have become a staple for processing time in planning problems [Yorke-Smith, 2005] and in multi-agent systems [Boerkoel Jr. and Durfee, 2013]. It is important to return to PPC, the original algorithm by Bliek and Sam-Haroud [1999] in order to correct it and improve its practical performance.

2. Second, in recent years, the Constraint Systems Laboratory has developed new algorithms for relational consistency that operate by filtering constraints (e.g., R($*$,m)C using PerTuple [Karakashian *et al.*, 2010; 2013], R($*$,m)C using PerFB [Schneider *et al.*, 2014], R($*$,m)C using AllSol [Geschwender *et al.*, 2016], RNIC [Woodward *et al.*, 2011; 2012], Living-STR [Woodward *et al.*, 2014]). Those algorithms were shown to effectively solve many difficult benchmark problems. However, the properties enforced by those algorithms are based

on inverse consistency and are, in general, not comparable to PPC. Before we can empirically compare the two 'flavors' of consistency property, we need to study the PPC property and its proposed algorithm in detail.[1]

3. Finally, directional path consistency can be viewed as an efficient approximation of the more general *variable-elimination* mechanism [Dechter, 2003]. In fact, resolution steps that generate *new* binary clauses have become a key component of successful SAT solvers [Eén and Biere, 2005].

The above three reasons are the main motivations that justify our efforts.

1.2 Contributions

Below, we list the contributions of this thesis:

1. We provide a concise and uniform description of the main known algorithms for path consistency and its approximations (both weak and strong).

2. We identify and correct errors that have appeared in the literature concerning the definition of the property of partial path consistency, the algorithm for enforcing it, and a proposition about it reported in the literature.

3. We settle an open question that appeared in the literature concerning whether or not partial path consistency and path consistency are equivalent with respect to detecting the insolubility of a CSP.

4. We introduce two improvements to the well-known PC-8 algorithm [Chmeiss and Jégou, 1998], yielding a new algorithm, which we call PC-8^+.

[1] The comparison between inverse consistency properties and PPC is beyond the scope of this thesis.

5. We study the known PPC algorithm [Bliek and Sam-Haroud, 1999], and propose two new variations, namely ΔPPC and σ-ΔPPC.

6. We conduct extensive empirical evaluations on both on randomly generated CSPs and on benchmark problems to compare the performance of over 15 algorithms: PC-2 [Mackworth and Freuder, 1984], DPC [Dechter and Pearl, 1988], PC-8 [Chmeiss and Jégou, 1998], PC-2001 [Bessière *et al.*, 2005], PPC+AP (a correction of PPC [Bliek and Sam-Haroud, 1999]), sCDC1 [Lecoutre *et al.*, 2007a], sDC2 [Lecoutre *et al.*, 2007b], PC-8-Flag, PC-8-Ordering, PC-8^+, PC-2001-Flag, PC-2001-Ordering, PC-2001^+, ΔPPC, σ-ΔPPC, and σ-ΔPPC^{sup}.

Finally, we identify directions for future research.

1.3 Outline of Thesis

This thesis is structured as follows. Chapter 2 introduces CSPs, discusses main consistency properties, and reviews in great detail various algorithms for enforcing local consistencies. Chapter 3 highlights errors that we have encountered in the literature. Chapter 4 discusses the new algorithms that we introduce. Chapter 5 discusses our empirical evaluations. Finally, Chapter 6 concludes this thesis.

Chapter 2

Background

In this chapter, we first define constraint satisfaction problems and review their representation in terms of constraint graphs. Then, we discuss local consistency properties and some selected consistency-enforcing algorithms. Finally, we give a comprehensive review of the main path-consistency algorithms that have appeared in the literature.

2.1 Constraint Satisfaction Problem: Problem Definition

A *constraint satisfaction problem* (CSP) is defined as $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, where $\mathcal{V}$ is a set of variables, $\mathcal{D}$ is a set of domains, where a domain is a set of values that a variable can take, and $\mathcal{C}$ is a set of constraints restricting the combinations of values that variables can take at the same time. A solution to a CSP is an assignment of values to variables such that all the constraints are satisfied. Deciding the existence of a solution is NP-complete.

Each variable $V_i \in \mathcal{V}$ has a finite domain $D_i \in \mathcal{D}$, which is a set of values that the variable can take. A *variable-value pair* (vvp) (V_i, a) is the association of a variable

V_i to a value $a \in D_i$. Each variable is constrained by a subset of the constraints in $\mathcal{C}$.

Each constraint $C_i \in \mathcal{C}$ is defined as $C_i = \langle \mathcal{S}_i, R_i \rangle$, where

- The *scope* of a constraint C_i, $scope(C_i) \subseteq \mathcal{V}$, is the set of variables to which C_i applies. The *arity* of C_i is the cardinality of the scope, $|scope(C_i)|$.
- The relation of C_i, $R_i = rel(C_i)$, is defined as a subset of the Cartesian product of the domains of the variables in $scope(C_i)$. Relations are defined either in intension (i.e., as a set builder) or in extension as a set of tuples (e.g., *table* constraints, tries, or multi-valued decision diagrams).

A tuple $t_i \in rel(C_i)$ specifies a combination of values that are either allowed (i.e., *support*) or forbidden (i.e., *conflict* or *no-good*) by C_i. A constraint is satisfied when every variable in its scope is assigned a value from its respective domain that is allowed by the constraint. A *universal binary* constraint is a constraint between two variables that allows all combinations of values in the domains of the two variables.

The *neighbors* of a variable are those variables that appear with the variable in the scope of some constraint.

In this thesis, we restrict ourselves to CSPs with binary constraints (i.e., arity 2) and to table constraints specified with supports (i.e., *positive* table constraints). Further, we denote the constraint defined over the two variables V_i and V_j as $C_{i,j}$ and assume that $C_{i,j} = C_{j,i}$.

Example 2.1.1 provides a simple example of a CSP instance.

Example 2.1.1. Consider the following CSP:

- $\mathcal{V} = \{V_1, V_2, V_3\}$
- $D_1 = D_2 = D_3 = \{1, 2, 3\}$

- $\forall\, i, j \in \{1, 2, 3\}, i \neq j, R_{i,j} = \{(1, 2), (2, 1), (1, 3), (3, 1), (2, 3), (3, 2)\}$ is the set of support tuples.
- A solution to this CSP is $\{(V_1, 1), (V_2, 3), (V_3, 2)\}$.

2.2 Solving CSPs

CSPs are typically solved with some combination of *backtrack search* and *constraint propagation*.

Backtrack search is a systematic, exhaustive exploration of the search space obtained by instantiating variables. Search proceeds by building a solution incrementally, instantiating one variable at a time and ensuring that the partial solution is consistent (i.e., does not break any of the constraints defined over the variables in the partial solution). When a partial solution cannot be extended to the next considered variable, the last assignment is undone by backtracking and a new instantiation is attempted. Because search proceeds in a depth-first manner, i.e., maintaining a single partial solution, it requires linear space. However, it requires exponential time in the number of variables of the CSP (i.e., $\mathcal{O}(d^n)$ where d is the maximum domain size and n the number of variables). The order in which the variables are considered for instantiation, the *instantiation order*, may greatly affect the performance of backtrack search. Conventional wisdom dictates to instantiate the 'most constrained' variable first.

Constraint propagation is the process of locally enforcing some level of consistency on combinations of variables (or constraints) and propagating the effect of such an operation across the network until reaching a fixpoint (see below for more details).

2.3 Graphical Representation

A CSP is often graphically represented as a *constraint network* (CN).[1] A constraint network represents the variables as vertices and the constraints as edges connecting the variables in their scopes. Figure 2.1 shows the constraint graph of the CSP from Example 2.1.1.

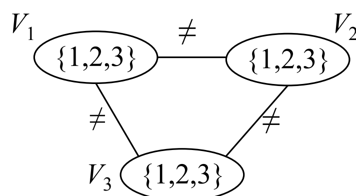


Figure 2.1: A simple, binary CSP with three variables

An *ordering* of a graph is a total ordering of the vertices of a graph. The *parents* of a vertex are the neighbors that appear before it in the ordering. The *width of a vertex* is the number of its parents. The *width of an ordering* is the maximum vertex width. The *width of a graph*, denoted w, is the minimum width of all its possible orderings, and can be found in quadratic time in the number of vertices in the graph [Freuder, 1982].

A graph is *triangulated*, or *chordal*, iff every cycle of length four or more in the graph has a chord, which is an edge between two non-consecutive vertices. To triangulate a graph, chords are added to chordless cycles of length four or more. The added chords are called fill edges. Minimizing the number of fill edges is NP-hard [Yannakakis, 1981]. A triangulated graph (i.e., moralized graph) is obtained when each pair of parents of every vertex are connected (i.e., moralizing the vertex by marrying its parents), whose width is called the *induced width*, denoted w^*, of the ordering used.

[1]The terms *constraint graph* and *constraint hypergraph* are also commonly used in the literature for binary and non-binary CSPs, respectively.

A *perfect elimination ordering* (PEO) of a graph is "an ordering of the vertices of the graph such that, for each vertex V, V and the neighbors of V that occur after V in the order form a clique. A graph is chordal iff it has a perfect elimination ordering" [Fulkerson and Gross, 1965]. Figure 2.2 shows a PEO of a graph with six vertices.

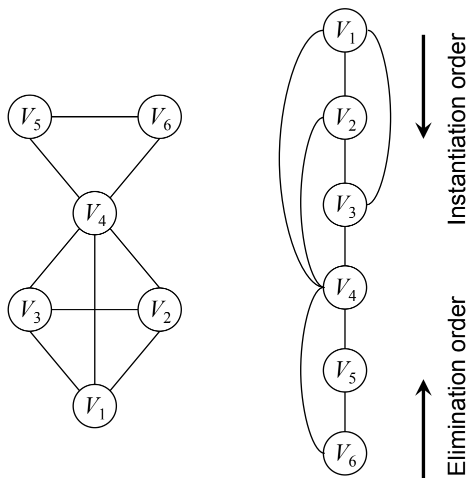


Figure 2.2: The constraint network of a simple binary CSP and a PEO of this network

The *induced width of a graph* is the minimum induced width of all orderings. Finding a graph's induced width and its ordering is NP-hard [Arnborg, 1985]. However, a good approximation of a graph's induced width is the width of a PEO of the graph triangulated using the MIN-FILL heuristic [Kjærulff, 1990]. Note that if a graph is triangulated, the width of the graph is the width any PEO of the graph.

2.4 Global Consistency Properties

The main global consistency properties are consistency, minimality, and decomposability.

- *Consistency* is simply another term for solvability of the CSP.

- Montanari [1974] introduced *Minimality* as the "central problem." Stated informally, the *minimal network* network is the one where the relations are as tight as can be, that is, each tuple in a relation can be extended to a solution to the CSP. We give the formal definition as formulated by Dechter [2003].

  Definition 2.4.1. *Minimality* [Dechter, 2003]. Given a CSP $\mathcal{P}_0$, let $\{\mathcal{P}_1, \ldots, \mathcal{P}_l\}$ be the set of all networks equivalent to $\mathcal{P}_0$. Then the minimal network M of $\mathcal{P}_0$ is defined by $M(\mathcal{P}_0) = \cap_{i=1}^{l} \mathcal{P}_i$.

  Gottlob [2011] showed that not only is enforcing minimality NP-complete but also that solving a minimal CSP remains NP-complete.

- Montanari [1974] also introduced *decomposability*, which guarantees that every assignment of any number of variables in the CSP that is consistent with the constraints defined over these variables can be extended to a complete solution of the CSP.

Decomposability guarantees backtrack-free search and it also guarantees minimality. In turn, minimality guarantees consistency, which is NP-complete.

2.5 Local Consistency Properties

A CSP with at least one solution is solvable (or consistent). Because determining the consistency of a CSP is NP-complete, and thus likely intractable, significant research has been invested in defining *local consistency* properties. Local consistency properties are defined on sub-problems of the original problem that have a pre-determined fixed size (e.g., two variables or two constraints). Because the size is fixed, enforcing local consistencies is usually tractable. Below, we review some basic local consistency

properties. Then, we recall more 'advanced' consistency properties that are relevant to our research.

2.5.1 Basic Local Consistency Properties for Binary CSPs

We first recall the most basic and common local consistency properties for binary CSPs, namely, arc consistency, path consistency, directional path consistency, and (strong) k-consistency.

2.5.1.1 Arc Consistency (AC)

A problem is *arc consistent* when every value in the domain of each variable is consistent with all of the variable's neighbors. Meaning that, given a variable that has taken a value, at least one value can be found in each of the variable's neighbors' domains that is allowed by the constraint between the two variables.

Definition 2.5.1. *Arc Consistency* (AC) [Mackworth, 1977]. Given a CSP $\mathcal{P}$, with $C_{i,j} \in \mathcal{C}$ a variable V_i is arc consistent relative to V_j iff for every value $a \in D_i$ there exists some value $b \in D_j$ such that $(a, b) \in R_{i,j}$. The arc (V_i, V_j) is arc consistent iff V_i is arc consistent relative to V_j and V_j is arc consistent relative to V_i. $\mathcal{P}$ is said to be arc consistent iff all of its arcs are arc consistent.

2.5.1.2 Path Consistency (PC)

Montanari [1974] originally introduced the property of *path consistency* as a tractable approximation of minimality (see Section 2.4). Mackworth [1977] later restated it as follows:

- A path in a CSP is a sequence $(V_i, \ldots, V_j)$ of variables of $\mathcal{P}$ s.t. $V_i \neq V_j$. Note that two adjacent variables in the path do not have to have an edge between them in the constraint network.

- An instantiation $\{(V_i, a), (V_j, b)\}$ is consistent along a path $(V_i, \ldots, V_j)$ if $(a, b) \in R_{i,j}$ and there exists a value for each variable in the path that satisfies the binary constraints along the path. Note that a variable can appear multiple times in the path and may take different values each time.[2] Further, if there is no constraint in $\mathcal{C}$ for two variables adjacent in the path, then the (binary) universal constraint applies.

- A path $(V_i, \ldots, V_j)$ is consistent iff $\forall\, (a, b) \in R_{i,j}, \{(V_i, a), (V_j, b)\}$ is a consistent instantiation.

- A CSP is path consistent iff every path in its constraint graph is consistent.

Because enumerating all paths is not practical, the following equivalent definition of path consistency is used. It requires that every arc consistent assignment to any two variables can be extended to every third variable in a consistent way.

Definition 2.5.2. *Path Consistency* (PC) [Dechter, 2003]. Given a CSP $\mathcal{P}$, the variables V_i and V_j are path consistent relative to a variable V_k iff for every consistent assignment $\{(V_i, a), (V_j, b)\}$ there is some value $c \in D_k$ such that both the assignments $\{(V_i, a), (V_k, c)\}$ and $\{(V_j, b), (V_k, c)\}$ are consistent. $\mathcal{P}$ is path consistent iff $\forall\, V_i, V_j, V_k \in \mathcal{V}$ with $k \neq i \neq j$, V_i and V_j are path consistent relative to V_k.

Given a consistent assignment $\{(V_i, a), (V_j, b)\}$ and a value $c \in D_k$ such that both the assignments $\{(V_i, a), (V_k, c)\}$ and $\{(V_j, b), (V_k, c)\}$ are consistent, we say that the

[2] See both the text and the footnote on page 109 [Montanari, 1974].

value $c \in D_k$ as well as the tuples $(a, c) \in R_{i,k}$ and $(b, c) \in R_{j,k}$ *support* the tuple $(a, b) \in R_{i,j}$

Example 2.1.1 (see also Figure 2.1) shows a CSP that is both arc consistent and path consistent. Note that path consistency does not imply arc consistency. The two examples in Figure 2.3 are path consistent but are not arc consistent.

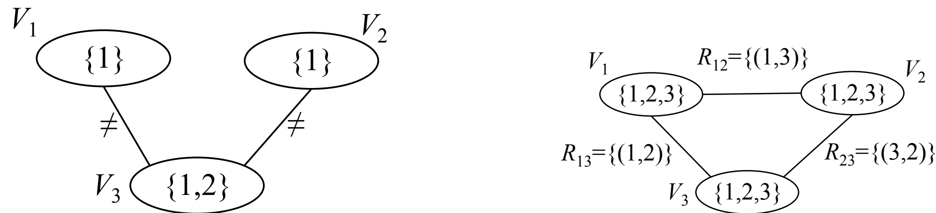


Figure 2.3: Two simple CSPs that are PC, but not AC

For the left example, the variable V_3 is not arc consistent relative to V_1 since there does not exist a value $a \in D_1$ such that $(a, 1) \in R_{1,3}$.[3] For the right example, all three variables are mutually arc inconsistent. For example, V_1 is not arc consistent relative to V_2 since there does not exist a value $a \in D_2$ such that $(2, a) \in R_{1,2}$.

2.5.1.3 k-Consistency and Strong k-Consistency

We recall the k-consistency property in a similar manner to arc consistency (which is 2-consistency) and path consistency (which is 3-consistency) [Freuder, 1978]. Basically, a problem is k-consistent if every consistent assignment to every combination of $k - 1$ variables can be consistently extended to any k^{th} variable.

Note that k-consistency does not guarantee $(k - 1)$-consistency (e.g., see Figure 2.3). The property *strong k-consistency* guarantees that the problem is j-consistent for all $j \leq k$. Further, strong n-consistency, where n is the number of variables in the CSP, is equivalent to decomposability, which was defined in Section 2.4.

[3] The same holds for V_3 relative to V_2.

Importantly, Freuder showed that any network with width j that is $(j+1)$-consistent can be solved in a backtrack-free manner [1982].

2.5.1.4 Directional Path Consistency (DPC)

Dechter and Pearl [1988] defined the notion of *directional k-consistency* along a given ordering of the variables as a restricted form of k-consistency. We recall the definition of directional path consistency, for which we will give an algorithm in Section 2.8.2.

Definition 2.5.3. *Directional Path Consistency* (DPC) [Dechter and Pearl, 1988]. A CSP is directional path consistent relative to order $ord = (V_1, V_2, \ldots, V_n)$, iff for every $k \geq i, j$, the two variables V_i and V_j are path consistent relative to V_k.

In Figure 2.4, we illustrate DPC, which guarantees that every consistent assignment to V_i and V_j can be extended to V_k.

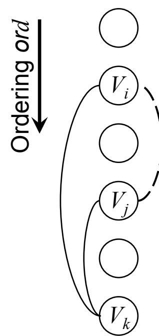


Figure 2.4: Illustrating DPC: for every tuple $(a, b) \in R_{i,j}$ (the dashed line), there is a value $c \in D_k$ so that $(a, c) \in R_{i,k}$ and $(b, c) \in R_{j,k}$ (the solid lines)

Importantly, Dechter and Pearl showed that, when given a CSP, $\mathcal{P}$, and an ordering, *ord*, of the variables of $\mathcal{P}$, if $\mathcal{P}$ is strong DPC along *ord* and the induced width of *ord* is two, then $\mathcal{P}$ can be solved in a backtrack-free manner along *ord* [1988].

2.5.2 Other Local Consistency Properties for Binary CSPs

In recent years, new local consistency properties have been introduced in the literature. We recall conservative path consistency [Debruyne, 1999] and partial path consistency [Bliek and Sam-Haroud, 1999], which were introduced the same year. We also recall singleton arc consistency [Debruyne and Bessière, 1997], dual consistency [Lecoutre *et al.*, 2007a], and conservative dual consistency [Lecoutre *et al.*, 2007a].

2.5.2.1 Conservative Path Consistency (CPC)

We first recall conservative path consistency, which is a restriction of path consistency to the existing constraints of a problem. If there is no $C_{i,j} \in \mathcal{C}$ then V_i and V_j are always conservative path consistent.

Definition 2.5.4. *Conservative Path Consistency* (CPC) [Debruyne, 1999]. An assignment to two variables V_i and V_j such that there is no constraint $C_{i,j} \in \mathcal{C}$ is conservative path consistent. If $\exists\, C_{i,j} \in \mathcal{C}$, the assignment $\{(V_i, a), (V_j, b)\}$ is conservative path consistent iff $(a, b) \in R_{i,j}$ and $\forall\, V_i, V_j, V_k \in \mathcal{V}$ with $k \neq i \neq j, C_{i,k}, C_{j,k} \in \mathcal{C} \Rightarrow \exists\, c \in D_k$ such that $(a, c) \in R_{i,k}$ and $(b, c) \in R_{j,k}$. A constraint $C_{i,j} \in \mathcal{C}$ is conservative path consistent iff for all the tuples $(a, b) \in R_{i,j}$, the assignment $\{(V_i, a), (V_j, b)\}$ is conservative path consistent. A CSP is conservative path consistent iff it is arc consistent and $\forall\, C_{i,j} \in \mathcal{C}, C_{i,j}$ is conservative path consistent.

2.5.2.2 Partial Path Consistency (PPC)

We now recall *partial path consistency* [Bliek and Sam-Haroud, 1999], which was introduced in the same year as CPC. As we argue in Section 3.1, the definition of PPC in the original paper is incomplete and likely flawed. Below, we report the

definition of PPC stated by Lecoutre *et al.* [2011]. Lecoutre *et al.* first introduce the definition of graph-path and closed (graph-)path respectively as follows:

- *A graph-path* of a constraint graph is a path $(V_1, \ldots, V_k)$ of the graph s.t. $\forall i \in \{1, \ldots, k-1\}, \exists\, C_{i,i+1} \in \mathcal{C}$. Informally, the graph-path can only use existing edges in the graph.

- *A closed (graph-)path* of a constraint graph is a (graph-)path $(V_1, \ldots, V_k)$ of the graph s.t. $\exists\, C_{1,k} \in \mathcal{C}$. Informally, there must be an edge between the two endpoints of the path. Note that a closed graph-path is a cycle in the constraint graph.

Definition 2.5.5. *Partial Path Consistency* (PPC) [Lecoutre *et al.*, 2011]. A CSP is partially path consistent iff every closed graph-path of its constraint graph is consistent.

We introduce the following proposition, not reported in the literature:

Proposition 2.5.1. *DPC defined along any ordering is strictly weaker than PPC.*

Sketch Proof: PPC guarantees that every closed path of any length is path consistent. DPC requires that the edges of the triangles along the ordering be consistent but not those along the reverse ordering. □

2.5.2.3 Singleton Arc Consistency (SAC)

Debruyne and Bessière [1997] introduced singleton consistency. It takes advantage of the fact that if a vvp (V_i, a) is consistent (i.e., appears in a solution), then the CSP obtained by restricting the domain of V_i to the singleton a is consistent (i.e., has a solution). We denote the CSP obtained by assigning the value a to the variable V_i

as $\mathcal{P}|_{D_i=\{a\}} = \mathcal{P}|_{V_i=a}$. Debruyne and Bessière [1997] noticed that if $\mathcal{P}|_{D_i=a}$ is inconsistent, then a can safely be removed from D_i. A *singleton test*[4] consists of assigning a value to a given variable and enforcing a consistency algorithm. If the algorithm fails, then the value is removed because it cannot appear in any solution. Singleton consistency can be combined with any second consistency property. Enforcing the second consistency property on every CSP obtained after the singleton assignment will not yield an inconsistent problem. For example, singleton arc consistency is defined relative to arc consistency:

Definition 2.5.6. *Singleton Arc Consistency* (SAC) [Bessiere, 2006]. A CSP is singleton arc consistent iff $\forall V_i \in \mathcal{V}, a \in D_i, P|_{D_i=\{a\}}$ is not arc inconsistent.

2.5.2.4 Dual Consistency (DC) and Conservative Dual Consistency (CDC)

Lecoutre *et al.* [2011] introduced dual consistency and conservative dual consistency. Dual consistency is defined using arc consistency:

Definition 2.5.7. *Dual Consistency* (DC) [Lecoutre *et al.*, 2011]. Given a CSP, $\mathcal{P}$, an assignment $\{(V_i, a), (V_j, b)\}$ is dual consistent iff $(V_j, b) \in AC(\mathcal{P}|_{V_i=a})$ and $(V_i, a) \in AC(\mathcal{P}|_{V_j=b})$. $\mathcal{P}$ is dual consistent iff every consistent assignment $\{(V_i, a), (V_j, b)\}$ is dual consistent.

Conservative dual consistency distinguishes between pairs of variables that are connected and those that are not:

Definition 2.5.8. *Conservative Dual Consistency* (CDC) [Lecoutre *et al.*, 2007a]. Given a CSP, $\mathcal{P}$, an assignment $\{(V_i, a), (V_j, b)\}$ is conservative dual consistent iff

[4] A singleton test is called probing in SAT solvers.

$(C_{i,j} \notin \mathcal{C}) \vee ((V_j, b) \in AC(\mathcal{P}|_{V_i=a}) \wedge (V_i, a) \in AC(\mathcal{P}|_{V_j=b}))$. $\mathcal{P}$ is conservative dual consistent iff every consistent assignment $\{(V_i, a), (V_j, b)\}$ is conservative dual consistent.

PC, DC, and CDC can be combined with AC to yield the corresponding *strong* consistency properties sPC, sDC, and sCDC.

Proposition 2.5.2. *Strong* Conservative Dual Consistency (sCDC) *is equivalent to* SAC+CDC [Lecoutre *et al.*, 2011].

2.5.3 Comparing Local Consistency Properties

Table 2.1 summarizes the consistency properties discussed in Sections 2.4 and 2.5.

Table 2.1: Summary of the studied consistency properties

Acronym	Consistency property	Original paper
	Minimality	[Montanari, 1974]
	Decomposability	[Montanari, 1974]
	(strong) k-consistency	[Freuder, 1978]
AC	Arc Consistency	[Mackworth, 1977]
PC	Path Consistency	[Montanari, 1974]
sPC (AC∧PC)	strong Path Consistency	[Freuder, 1978]
DPC	Directional Path Consistency	[Dechter and Pearl, 1988]
PPC	Partial Path Consistency	[Bliek and Sam-Haroud, 1999]
CPC	Conservative Path Consistency	[Debruyne, 1999]
DC	Dual Consistency	[Lecoutre *et al.*, 2011]∗
sDC (AC∧DC)	strong Dual Consistency	[Lecoutre *et al.*, 2011]∗
CDC	Conservative Dual Consistency	[Lecoutre *et al.*, 2011]∗
SAC	Singleton Arc Consistency	[Debruyne and Bessière, 1997]
sCDC (SAC∧CDC)	strong Conservative Dual Consistency	[Lecoutre *et al.*, 2011]∗

∗ Definitions were introduced by Lecoutre *et al.* [2007a; 2007b] and later corrected by Lecoutre *et al.* [2011].

Using the terminology introduced by Debruyne and Bessière [1997], we say that a local consistency property LC is *stronger* than another local consistency property LC' if in any CSP where LC holds, LC' also holds. Further, LC is *strictly stronger* than

LC' if LC is stronger than the LC' and there exists at least one CSP in which LC' holds but LC does not. Finally, LC and LC' are *equivalent* when LC is stronger than LC' and vice versa [Bessière *et al.*, 2008]. In practice, when a consistency property is stronger (respectively, weaker) than another, enforcing the former never yields less (respectively, more) pruning than enforcing the latter on the same problem does.

Figure 2.5 shows the relationships between the local consistency properties relevant to this thesis. The proofs can be found in [Lecoutre *et al.*, 2011]. Each arrow points from a property that is stronger to a property that is weaker.

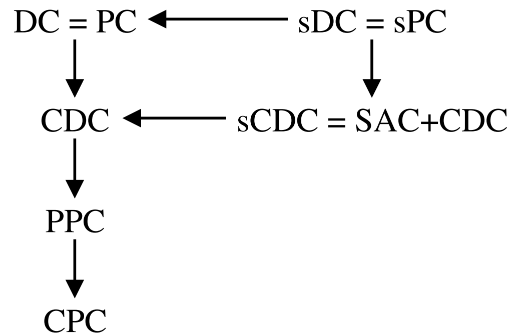


Figure 2.5: Partial order of some consistency properties [Lecoutre *et al.*, 2011]

2.6 Algorithms for Local Consistency

A consistency algorithm enforces a given consistency property on a CSP. Typically, a consistency enforcing algorithm operates by either removing values from the variables' domains (i.e., *domain* filtering) or by removing tuples from the constraints' relations (i.e., *relation* filtering). Some algorithms may do both domain and relation filtering. The local consistency property being enforced guarantees that any value (or tuple) removed does not participate in any solution to the CSP. Thus, enforcing a given consistency property never sacrifices solutions.

More than one algorithm may exist for enforcing a given consistency property. For example, arc consistency is enforced by AC-1 [Mackworth and Freuder, 1984], AC-3 [Mackworth and Freuder, 1984], AC-4 [Mohr and Henderson, 1986], and AC-2001 [Bessière *et al.*, 2005]. Similarly, path consistency is enforced by PC-1 [Montanari, 1974], PC-2 [Mackworth and Freuder, 1984], PC-8 [Chmeiss and Jégou, 1998], and PC-2001 [Bessière *et al.*, 2005].

Algorithms that enforce the same consistency property may use different book-keeping data-structures to avoid repeated consistency checks (e.g., whether or not a tuple appears in a relation), thus trading space for time.

2.7 An Algorithm for Arc Consistency: AC-2001

In this section, we discuss AC-2001 (Algorithm 1) [Bessière and Régin, 2001; Zhang and Yap, 2001; Bessière *et al.*, 2005], currently the best and most widely used generic algorithm for enforcing arc consistency.

Algorithm 1: AC-2001($\mathcal{P}$) *Adapted from* [Bessière *et al.*, 2005]

```
   Input: P = (V, D, C)
   Output: Arc consistent P
1  foreach V_i ∈ V do
2      foreach V_j ∈ V such that C_{i,j} ∈ C do
3          foreach a ∈ D_i do Last((V_i, a), V_j) ← ∅
4  AC-2001-CORE(P, V)
5  return P
```

This algorithm requires that the problem's domains be totally ordered. AC-2001 uses the function NEXT on the ordered domains. NEXT(D_i, a) yields the value after a in D_i. When a is *nil*, NEXT(D_i, *nil*) returns the first element in D_i. When a is the last element in D_i, NEXT(D_i, a) returns *nil*. Like its predecessors, (e.g., AC-1

Algorithm 2: AC-2001-CORE($\mathcal{P}, \mathcal{X}$) *Adapted from* [Bessière *et al.*, 2005]

```
  Input: P = (V, D, C), X ⊆ V
  Output: Arc consistent P
1 Q ← ∅
2 foreach V_i ∈ X do
3   foreach V_j ∈ V such that C_{i,j} ∈ C do Q ← Q ∪ {(V_j, V_i)}
4 while Q do
5   (V_i, V_j) ← Pop(Q)
6   if Revise-2001(V_i, V_j) then
7     foreach V_k ∈ V such that C_{i,k} ∈ C ∧ (k ≠ j) ∧ (k ≠ i) do
8       Q ← Q ∪ {(V_k, V_i)}
9 return P
```

Algorithm 3: REVISE-2001(V_i, V_j) *Adapted from* [Bessière *et al.*, 2005]

```
   Input: V_i, V_j ∈ V
   Output: true if D_i has been revised, false otherwise
1  revised ← false
2  foreach a ∈ D_i do
3    b ← Last((V_i, a), V_j)
4    if b ∉ D_j then
5      b ← Next(D_j, b)
6      while b do
7        if (a, b) ∈ R_{i,j} then
8          Last((V_i, a), V_j) ← b
9          break
10       b ← Next(D_j, b)
11     if not b then
12       D_i ← D_i \ {a}
13       revised ← true
14 return revised
```

and AC-3), AC-2001 loops through all ordered pairs of variables, (V_i, V_j), that share a constraint $C_{i,j}$ and revises the domain of V_i given $R_{i,j}$. REVISE-2001 (Algorithm 3) updates D_i by removing the values that have no supporting value in D_j given $R_{i,j}$.

The particularity of AC-2001 is that for every value a in D_i it keeps track of the

first value in D_j that *supports* (V_i, a). As long as the supporting value remains in the domain of D_j, the support is 'active,' and reused in future calls to REVISE-2001. The data structure used for this bookkeeping is $Last((V_i, a), V_j)$. In practice, this simple structure saves many constraint-checking operations. Its space complexity is $\mathcal{O}(end)$, where e is the number of binary constraints, n is the number of variables in the problem, and d is the maximal domain size.

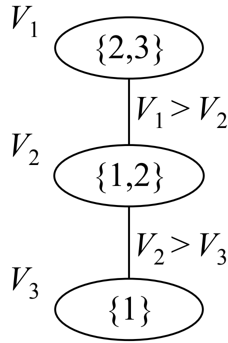


Figure 2.6: A three-variable example

		Revising ...				
		(V_2, V_1)	(V_1, V_2)	(V_3, V_2)	(V_2, V_3)	(V_1, V_2)
$((V_1,2),V_2)$	*nil*	*nil*	1	1	1	*nil*
$((V_1,3),V_2)$	*nil*	*nil*	1	1	1	2
$((V_2,1),V_1)$	*nil*	2	2	2	2	2
$((V_2,2),V_1)$	*nil*	3	3	3	3	3
$((V_2,1),V_3)$	*nil*	*nil*	*nil*	*nil*	*nil*	*nil*
$((V_2,2),V_3)$	*nil*	*nil*	*nil*	*nil*	1	1
$((V_3,1),V_2)$	*nil*	*nil*	*nil*	2	2	2

Figure 2.7: Processing $\mathcal{Q}$, updated values are shown in gray

We illustrate the operation of AC-2001 on the example shown in Figure 2.6. First, AC-2001 initializes the contents of *Last* to *nil*. The first column of Figure 2.7 shows the elements of *Last* and the second column shows the initial *nil* values. As the algorithm proceeds, REVISE-2001 updates *Last* elements to new values as illustrated in columns 3–7 of Figure 2.7. The figure highlights values that are updated by showing them in gray.

AC-2001-CORE (Algorithm 2) initializes the propagation queue, $\mathcal{Q}$, to: $\mathcal{Q} \leftarrow \{(V_2, V_1), (V_1, V_2), (V_3, V_2), (V_2, V_3)\}$. It then passes each queue element to REVISE-2001. Figure 2.7 has a column for each element of $\mathcal{Q}$ and the updates to *Last* are shown there.

When REVISE-2001 is passed (V_2, V_3), it finds no support for $(V_2,1)$ in D_3. As a

result, REVISE-2001 removes the value 1 from the domain of D_2 and returns *true*. AC-2001-CORE adds (V_1, V_2) to the queue.

For the last element of the queue, (V_1, V_2), *Last* has non-*nil* values for the two elements we are concerned with (see the top two rows of Figure 2.7). The *Last* value for the element $((V_1, 2), V_2)$ is 1, but $1 \notin D_2$. Thus, REVISE-2001 searches for a new support starting from NEXT$(D_2, 1)$, but none is found and 2 is removed from D_1. Next, REVISE-2001 seeks a support for $(V_1, 3)$ in V_2, but we already saw that $Last((V_1, 3), V_2) = 1 \notin D_2$. This time, it finds a new support, the value 2, and updates $Last((V_1, 3), V_2) \leftarrow 2$. At this point, the queue becomes empty and AC-2001 terminates successfully.

Throughout this thesis, we enforce AC using the AC-2001 algorithm.

2.8 Algorithms Related to Path Consistency

We now discuss several algorithms that enforce or approximate path consistency. The algorithms are listed in Table 2.2; their properties are summarized in Table 2.3; their data structures and propagation queues are given in Table 2.4.

Table 2.2: List of the studied algorithms

Algorithm	**Section**	**Original publication**
PC-2	Section 2.8.1	[Mackworth and Freuder, 1984]
DPC	Section 2.8.2	[Dechter and Pearl, 1988]
PC-8	Section 2.8.3	[Chmeiss and Jégou, 1998]
PC-2001	Section 2.8.4	[Bessière *et al.*, 2005]
BSH-PPC	Section 2.8.5	[Bliek and Sam-Haroud, 1999]
sCDC1	Section 2.8.6	[Lecoutre *et al.*, 2007a]
sDC2	Section 2.8.7	[Lecoutre *et al.*, 2007b]

Although more algorithms for path consistency are reported in the literature (e.g.,

PC-1 [Montanari, 1974], PC-3 [Mohr and Henderson, 1986], PC-4 [Han and Lee, 1988], PC-5 and PC-5++ [Singh, 1996], PC-6 [Chmeiss, 1996], PC-7 [Chmeiss and Jégou, 1996]), we choose to study only the latest ones as well as PC-2.[5]

sCDC1 and sDC2 were not formally introduced as path-consistency algorithms but instead as dual-consistency algorithms. In fact, they do not operate by composing constraints (like all the other PC-algorithms do), but instead they operate by running an arc-consistency algorithm.

In Table 2.3, we identify the following properties of the chosen algorithms: the constraint graph on which they operate, their pruning power with respect to PC-2, and their time and space complexity. Notice that sCDC1 is the only algorithm that does not modify the constraint network but instead operates on the original graph.

[5]Han and Lee [1988] found an error in PC-3 [Mohr and Henderson, 1986], which they fixed in PC-4. Chmeiss and Jégou [1996] showed that 'PC-4 is really inefficient in practice' [Chmeiss and Jégou, 1998]. Finally, Chmeiss and Jégou [1998] showed that 'PC-8 often outperforms' both PC-5 and PC-6.

Table 2.3: Properties of the studied algorithms

	Constraint graph	**Enforces consistency property**	**Pruning w.r.t. PC-2**	**Complexity**		
				Time	**Space**	
					Constraints	**Data structures**
PC-1	Completed	Strong PC	More	$\mathcal{O}(n^5d^5)$	$\mathcal{O}(n^2d^2)$	$\mathcal{O}(d^2)$
PC-2	Completed	PC	Same	$\mathcal{O}(n^3d^5)$	$\mathcal{O}(n^2d^2)$	$\mathcal{O}(n^3)$
DPC	Triangulated	Strong DPC	Less	$\mathcal{O}(n^3d^3)$ $\mathcal{O}(w_{peo}^2nd^3)$	$\mathcal{O}(e'd^2)$	$\mathcal{O}(1)$
PC-8	Completed	PC	Same	$\mathcal{O}(n^3d^4)$	$\mathcal{O}(n^2d^2)$	$\mathcal{O}(n^2d)$
PC-2001	Completed	PC	Same	$\mathcal{O}(n^3d^3)$	$\mathcal{O}(n^2d^2)$	$\mathcal{O}(n^3d^2)$
BSH-PPC	Triangulated	PPC	Less	$\mathcal{O}(\delta(e+e')d^5)$	$\mathcal{O}(e'd^2)$	$\mathcal{O}(n^3+e+e') = \mathcal{O}(n^3)$
sCDC1	Original	sCDC	Incomparable	$\mathcal{O}(\lambda end^3)$	$\mathcal{O}(1)$	$\mathcal{O}(ed^2)$
sDC2	Completed	Strong PC	More	$\mathcal{O}(\lambda n^3d^3)$	$\mathcal{O}(n^2d^2)$	$\mathcal{O}(n^2d^2)$

d: Maximum domain size, MAX($|D_i|$)
n: Number of variables, $|\mathcal{V}|$
e: Number of constraints before graph is altered, $|\mathcal{C}|$
e': Number of edges added by a triangulation of the graph
w_{peo}: Width of the triangulated graph in the *peo* ordering
δ: The maximum degree of the graph
λ: Number of allowed tuples in all the constraints of $\mathcal{P}$,
i.e., $\lambda = \sum_{C_i \in \mathcal{C}} |R_i|$ and is bounded by $\mathcal{O}(n^2d^2)$ and $\mathcal{O}(ed^2)$

In Table 2.4, we state whether or not each algorithm uses support structures and a propagation queue, and provide pointers to the constituent functions and their pseudocode.

Table 2.4: Data structures and propagation queues for the studied algorithms

	Supports	**Queue element**	**Pseudocode**
PC-1	None	None	-
PC-2	None	(V_i, V_j, V_k) with $i < k$, $i \neq j \neq k$	PC-2 (Algorithm 4) REVISE-3 (Algorithm 5)
PC-8	None	$((V_i, V_j), V_k)$	PC-8 (Algorithm 8)
PC-2001	$Last((V_i, a),(V_j, b),V_k) \leftarrow c$	$((V_i, V_j), V_k)$	PC-2001 (Algorithm 9)
BSH-PPC	None	$C_{i,j}$	BSH-PPC (Algorithm 10) REVISE-3 (Algorithm 5)
DPC	None	None	DPC (Algorithm 6) REVISE-3 (Algorithm 5) FILTER-DOM (Algorithm 7)
sCDC1	None, embedded AC may	None	sCDC1 (Algorithm 11) sCDC1-CHECK (Algorithm 12) REVISE-2001 (Algorithm 3) AC-2001 (Algorithm 1) AC-2001-CORE (Algorithm 2)
sDC2	None, embedded AC may	None	sDC2 (Algorithm 13) sDC2-CHECK (Algorithm 14) REVISE-2001 (Algorithm 3) AC-2001 (Algorithm 1) AC-2001-CORE(Algorithm 2) FC-2001 (Algorithm 15)

Below we review the operation of each of the studied algorithms.

2.8.1 Basic Path Consistency Algorithm (PC-2)

PC-2 is the first improvement on the first and most basic PC algorithm, PC-1, which was proposed by Montanari [1974]. The PC-2 algorithm proposed by Mackworth [1977] enforces strong path consistency (sPC). That is, in addition to updating relations to enforce path consistency it also updates the domains of the variables to enforce arc consistency. However, the PC-2 algorithm (Algorithm 4) that we discuss below is based on the version described by Dechter [2003], and enforces only path consistency.

Algorithm 4: PC-2($\mathcal{P}$) *Adapted from* [Dechter, 2003]

Input: $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ with a complete constraint graph
Output: Path consistent $\mathcal{P}$

1 $\mathcal{Q} \leftarrow \emptyset$
2 **foreach** $V_i, V_k, V_j \in \mathcal{V}, (i < j) \wedge (i \neq k) \wedge (j \neq k)$ **do**
3 $\quad \mathcal{Q} \leftarrow \mathcal{Q} \cup \{(V_i, V_k, V_j)\}$
4 **while** $\mathcal{Q}$ **do**
5 $\quad (V_i, V_k, V_j) \leftarrow$Pop$(\mathcal{Q})$
6 $\quad$ **if** Revise-3(V_i, V_j, V_k) **then**
7 $\quad\quad$ **foreach** $V_m \in \mathcal{V}, (m \neq i) \wedge (m \neq j)$ **do**
8 $\quad\quad\quad \mathcal{Q} \leftarrow \mathcal{Q} \cup \begin{cases} \textbf{if } m < i < j \textbf{ then } \{(V_m, V_i, V_j), (V_m, V_j, V_i)\} \\ \textbf{if } i < m < j \textbf{ then } \{(V_m, V_i, V_j), (V_i, V_j, V_m)\} \\ \textbf{if } i < j < m \textbf{ then } \{(V_j, V_i, V_m), (V_i, V_j, V_m)\} \end{cases}$
9 **return** $\mathcal{P}$

Like PC-1, PC-2 operates on a constraint graph that is complete, meaning that there is a constraint between every pair of variables. If the original constraint graph is not complete, the missing edges are added as universal binary constraints.

PC-2 improves the performance of PC-1 by using a more sophisticated propagation queue. The propagation queue begins with triplets of distinct variables:

$$\mathcal{Q} \leftarrow \{(V_i, V_k, V_j) \text{ s.t. } (i < j) \wedge (i \neq k) \wedge (j \neq k)\}$$

Algorithm 5: Revise-3(V_i, V_j, V_k) *Adapted from* [Dechter, 2003]

Input: $V_i, V_j, V_k \in \mathcal{V}$
Output: $\mathcal{T}_d$ the set of removed tuples from $R_{i,j}$

1 $\mathcal{T}_d \leftarrow \emptyset$
2 **foreach** $(a, b) \in R_{i,j}$ **do**
3 | **if** $\nexists c \in D_k$ such that $((a, c) \in R_{i,k}) \wedge ((b, c) \in R_{j,k})$ **then**
4 | | $R_{i,j} \leftarrow R_{i,j} \setminus \{(a, b)\}$
5 | | $\mathcal{T}_d \leftarrow \mathcal{T}_d \cup \{(a, b)\}$
6 **return** $\mathcal{T}_d$

For each element in $\mathcal{Q}$, Revise-3 (Algorithm 5) revises $R_{i,j}$ given $R_{i,k}$, $R_{k,j}$, and D_k. After any revision PC-2 adds only the triplets that may be affected by the revision back into the queue.[6]

We illustrate the operation of PC-2 on the example shown in Figure 2.8. We give the constraints in intension on the left and in extension on the right. On the left of Figure 2.8 the solid lines represent constraints with initial relations and the dashed lines represent two binary universal relations. PC-2 initializes the propagation queue

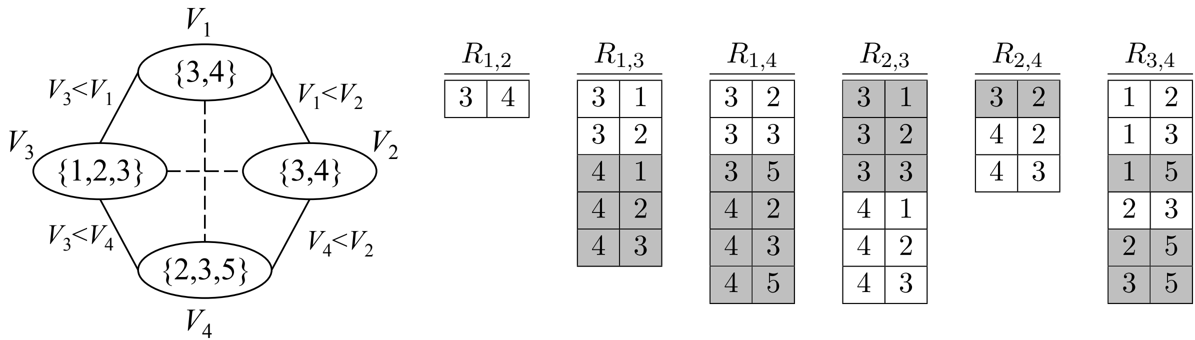


Figure 2.8: *Left*: Dashed lines in the constraint graph denote universal constraints. *Right*: The relations of the CSP. PC-2 removes the tuples are shown in gray

[6]Line 8 of PC-2 (Algorithm 4) is a correction of line 6 of the algorithm reported by Dechter [2003] (Figure 3.11 [2003]). The corrected line adds to the propagation queue triplets (i, k, j) exclusively with $i < j$, excluding triplets with $i > j$. Indeed, (i, k, j) and (j, k, i) effectuate the same consistency checking, and including both triplets in the propagation queue duplicates the constraint-checking effort.

with these 12 triplets (lines 1–3):

$$\mathcal{Q} \leftarrow \{(V_1, V_3, V_2), (V_1, V_4, V_2), (V_1, V_2, V_3), (V_1, V_4, V_3), (V_1, V_2, V_4), (V_1, V_3, V_4), (V_2, V_1, V_3), (V_2, V_4, V_3), (V_2, V_1, V_4), (V_2, V_3, V_4), (V_3, V_1, V_4), (V_3, V_2, V_4)\}$$

Table 2.5 summarizes the first 12 steps of PC-2's main loop (lines 1–8). Each row in this table shows:

1. The triplet (V_i, V_k, V_j) that is popped from the queue at line 5.
2. The revisions Revise-3 makes to $R_{i,j}$, when applicable.
3. The triplets PC-2 adds to the propagation queue at line 8.

Table 2.5: PC-2's first 12 steps on the example in Figure 2.8

Step	**Triplet**	**Relation update**	**Additions to $\mathcal{Q}$**
1	(V_1, V_3, V_2)	none	-
2	(V_1, V_4, V_2)	none	-
3	(V_1, V_2, V_3)	$R_{1,3} \leftarrow \{(3,1), (3,2)\}$	$\{(V_1, V_3, V_2)\}$
4	(V_1, V_4, V_3)	none	-
5	(V_1, V_2, V_4)	$R_{1,4} \leftarrow \{(3,2), (3,3)\}$	$\{(V_1, V_4, V_2), (V_1, V_4, V_3)\}$
6	(V_1, V_3, V_4)	none	-
7	(V_2, V_1, V_3)	$R_{2,3} \leftarrow \{(4,1), (4,2)\}$	$\{(V_1, V_2, V_3)\}$
8	(V_2, V_4, V_3)	none	-
9	(V_2, V_1, V_4)	$R_{2,4} \leftarrow \{(4,2), (4,3)\}$	$\{(V_1, V_2, V_4), (V_2, V_4, V_3)\}$
10	(V_2, V_3, V_4)	none	-
11	(V_3, V_1, V_4)	$R_{3,4} \leftarrow \{(1,2)(1,3), (2,3)\}$	$\{(V_1, V_3, V_4), (V_2, V_3, V_4)\}$
12	(V_3, V_2, V_4)	none	-

The fourth column in Table 2.5 indicates the only eight items in$\mathcal{Q}$ after the first 12 steps. PC-2 continues without yielding any further revisions or queue additions, at which point it terminates with an empty queue.

2.8.2 Directional Path Consistency (DPC)

Dechter and Pearl [1988] proposed the DPC algorithm to compute the DPC network given a CSP and an instantiation ordering of the variables.[7] As the algorithm proceeds, it adds edges to the graph (i.e., new constraints to the CSP), which corresponds to moralizing the graph along the given ordering.

The DPC algorithm (Algorithm 6) that we discuss below takes as input a CSP that has had its constraint graph triangulated and the instantiation ordering *ord* corresponding to the inverse of a perfect elimination ordering of the vertices of the graph.[8] Typically DPC considers fewer constraints than PC-2, because DPC operates on a triangulation of the constraint graph while PC-2 operates on the complete graph.

Algorithm 6: DPC($\mathcal{P}$, *ord*) *Adapted from* [Dechter and Pearl, 1988]

Input: $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ with a triangulated constraint graph,
ord: an instantiation ordering of $\mathcal{V}$
Output: Strong directional path consistent $\mathcal{P}$ in the direction of *ord*

```
1 for k ← |V| downto 2 by -1 do
2     for i ← 1 to k − 1 such that C_{i,k} ∈ C do REVISE(D_i, {C_{i,k}})
3     for i ← 1 to k − 2 such that C_{i,k} ∈ C do
4         for j ← i + 1 to k − 1 such that C_{i,j} ∈ C do REVISE-3(V_i, V_j, V_k)
5 return P
```

DPC loops through the variables starting with the last variable in the instantiation order *ord*. As it visits each variable V_k, it first enforces directional arc consistency on all the variables V_i preceding V_k in the ordering. That is, for all $i < k$ DPC uses FILTER-DOM (Algorithm 7) to revise D_i given the relation $R_{i,k}$. Then, for all

[7] In the instantiation ordering the variables are numbered from 1 to n.

[8] In order to triangulate the graph, we use the MIN-FILL heuristic, and add, as universal constraints, the fill edges generated by MIN-FILL [Kjærulff, 1990].

$i < j < k$, where $C_{i,k}$ and $C_{j,k}$ exist, DPC uses REVISE-3 (Algorithm 5) to revise $R_{i,j}$ given $R_{i,k}$, $R_{j,k}$, and D_k Figure 2.4 in Section 2.5.1.4 illustrates DPC's filtering.

Algorithm 7: FILTER-DOM($D_i, \mathcal{C}_i$)

Input: D_i: Domain of variable V_i,
$\mathcal{C}_i$: Set of constraints incident to variable V_i
Output: *true* if D_i was updated, *false* otherwise
1 $D_i^{old} \leftarrow D_i$
2 **foreach** $C \in \mathcal{C}_i$ **do** $D_i \leftarrow D_i \cap \pi_{\{V_i\}}(rel(C))$
3 **return** $D_i^{old} \neq D_i$

We illustrate the operation of DPC on the example shown in Figure 2.9. First, we triangulate the problem by adding a universal constraint between V_2 and V_3, which is shown as a dashed line in Figure 2.10.

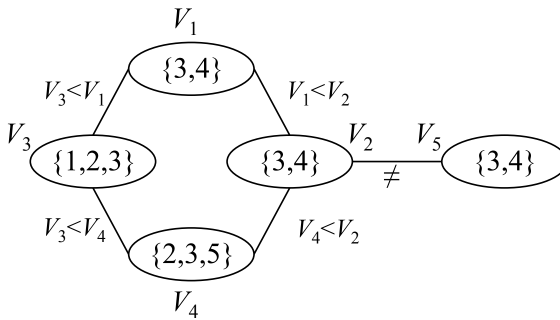


Figure 2.9: A five-variable example

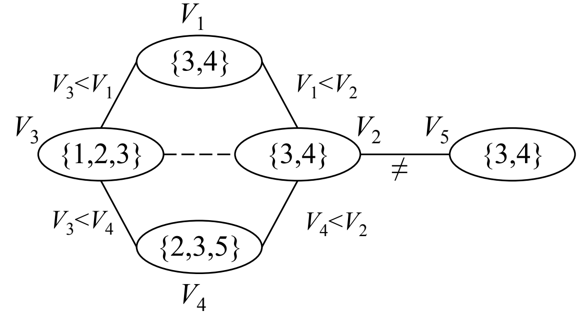


Figure 2.10: A triangulation of the constraint graph

DPC uses the ordering $V_1, V_2, \ldots, V_5$, as shown in Figure 2.11, and executes the following steps:

1. DPC first considers V_5, and updates the domain D_2 given $R_{2,5}$ (no filtering).

2. DPC then 'moves up to' V_4 by setting k to 4 at line 1. It updates D_2 and D_3 given $R_{2,4}$ and $R_{3,4}$ respectively (no filtering), and updates $R_{2,3} \leftarrow \{(3,1),(4,1),(4,2)\}$ given $R_{2,4}$ and $R_{3,4}$.

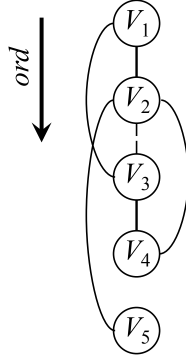


Figure 2.11: The ordering used

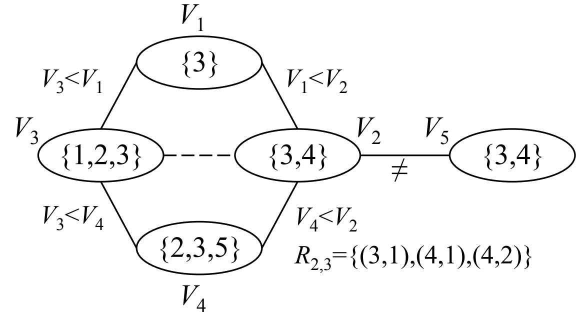


Figure 2.12: After running DPC

3. Moving up to V_3, it updates D_1 and D_2 given $R_{1,3}$ and $R_{2,3}$ respectively (no filtering). It updates $R_{1,2}$ given $R_{1,3}$ and $R_{2,3}$ (no filtering).

4. Finally, moving up to V_2, it updates $D_1 \leftarrow \{3\}$ given $R_{1,2}$.

DPC makes two updates to the problem, namely $D_1 \leftarrow \{3\}$ and $R_{2,3} \leftarrow \{(3,1),(4,1),(4,2)\}$. Figure 2.12 depicts the final constraint graph.

2.8.3 Iterating Over Variables' Domains (PC-8)

Chmeiss and Jégou [1998] proposed PC-8 (Algorithm 8) to enforce PC on a CSP's completed constraint graph. We compare PC-8 to PC-2. The consistency-checking operations of PC-8 (lines 3–5 and lines 10–12) operate similarly to Revise-3 (Algorithm 5), which is used in PC-2 (Algorithm 4). However, PC-8's propagation queue is different than PC-2's, which stores triplets of variables (V_i, V_k, V_j). PC-2 uses Revise-3 which iterates over *every* tuple in $R_{i,j}$ to ensure the existence of a supporting value in D_k with corresponding supporting tuples in $R_{i,k}$ and $R_{j,k}$. In contrast, PC-8's queue stores elements of the following form:

$$\mathcal{Q} \leftarrow \{((V_i, a), V_k) \text{ s.t. } (i \neq k)\}$$

Algorithm 8: PC-8($\mathcal{P}$) *Adapted from* [Chmeiss and Jégou, 1998]

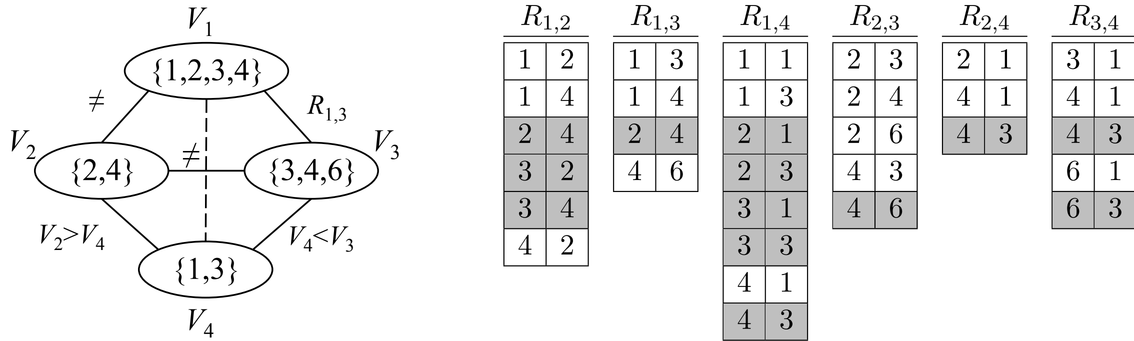
```
   Input: P = (V, D, C) with a complete constraint graph
   Output: Path consistent P
 1 Q ← ∅
 2 foreach V_i, V_j, V_k ∈ V such that (i < j) ∧ (i ≠ k) ∧ (j ≠ k) do
 3     foreach (a, b) ∈ R_{i,j} do
 4         if ∄ c ∈ D_k such that (a, c) ∈ R_{i,k} ∧ (b, c) ∈ R_{j,k} then
 5             R_{i,j} ← R_{i,j} \ {(a, b)}
 6             Q ← Q ∪ {((V_i, a), V_j), ((V_j, b), V_i)}
 7 while Q do
 8     ((V_i, a), V_k) ← Pop(Q)
 9     foreach V_j ∈ V such that (j ≠ i) ∧ (j ≠ k) do
10         foreach (a, b) ∈ R_{i,j} do
11             if ∄ c ∈ D_k such that (a, c) ∈ R_{i,k} ∧ (b, c) ∈ R_{j,k} then
12                 R_{i,j} ← R_{i,j} \ {(a, b)}
13                 Q ← Q ∪ {((V_i, a), V_j), ((V_j, b), V_i)}
14 return P
```

The elements are tuples $((V_i, a), V_k)$ where (V_i, a) is a variable-value pair and V_k is a variable. PC-8 considers every third variable, V_j (line 9), but instead of looping over every tuple in $R_{i,j}$ it only considers those where $V_i \leftarrow a$ (line 10). Every tuple (a, b) that lacks supports in $R_{i,k}$, $R_{j,k}$ and D_k is removed from $R_{i,j}$. Note that line 10 of Algorithm 8 appears as follows in the original paper:

$$\textbf{foreach } b \in D_j \text{ such that } (a, b) \in R_{i,j} \textbf{ do}$$

However, executing such an instruction adds unnecessary constraint checks for each value in D_j to the cost of the algorithm in practice. For this reason, we reformulated the instruction as appears our proposed pseudocode.

We illustrate the operation of PC-8 on the example shown in Figure 2.13. The tuples removed by PC-8's initial pass (lines 1–6) are shown in gray. As PC-8 deletes tu-



$R_{1,2}$	
1	2
1	4
2	4
3	2
3	4
4	2

$R_{1,3}$	
1	3
1	4
2	4
4	6

$R_{1,4}$	
1	1
1	3
2	1
2	3
3	1
3	3
4	1
4	3

$R_{2,3}$	
2	3
2	4
2	6
4	3
4	6

$R_{2,4}$	
2	1
4	1
4	3

$R_{3,4}$	
3	1
4	1
4	3
6	1
6	3

Figure 2.13: *Left*: A CSP where the dashed line denotes a universal constraint *Right*: Relations of the CSP with tuples removed by PC-8's lines 1–6 shown in gray

ples in the course of its initial pass (lines 1–6) elements are added to the queue(line 6), and after the initial pass the queue has the following tuples:

$$\begin{aligned}\mathcal{Q} \leftarrow \ & \{((V_1,2),V_2),((V_2,4),V_1),((V_1,3),V_2),((V_2,2),V_1),((V_1,2),V_3),((V_3,4),V_1),\\ & ((V_2,4),V_4),((V_4,3),V_2),((V_3,4),V_4),((V_3,6),V_4),((V_4,3),V_3),((V_1,2),V_4),\\ & ((V_1,3),V_4),((V_1,4),V_4),((V_4,1),V_1),((V_4,3),V_1),((V_2,4),V_3),((V_3,6),V_2)\}\end{aligned}$$

PC-8 continues by popping elements off of the queue (line 8) and checking for the existence of supporting tuples. For the first element, $((V_1,2),V_2)$, tuples in the relations between V_1 and every third variable in the problem, that is, V_3 then V_4 are considered, where $V_1 \leftarrow 2$ as illustrated in Figure 2.14. There is no such tuple $(2,*) \in R_{1,3}$ and no further checks are made to determine if any tuples need to be removed from $R_{1,3}$. The same holds for $R_{1,4}$. All other queue elements yield similar results until $((V_4,3),V_2)$ is considered. The tuples where $V_4 \leftarrow 3$, $(*,3) \in R_{2,4}$, are checked against V_1 as illustrated in Figure 2.15. When no support is found in D_1 the tuple $(1,3)$ is removed from $R_{1,4}$.

The processing continues without further updates until the queue is empty, at which point PC-8 terminates. Figure 2.16 shows the updated relations after PC-8

Figure 2.14: *Left*: Seeking supports for $(2, *) \in R_{1,3}$ in D_2. *Right*: Seeking supports for $(2, *) \in R_{1,4}$ in D_2

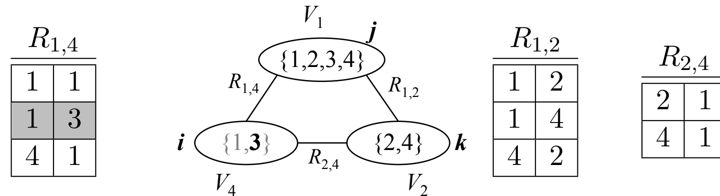
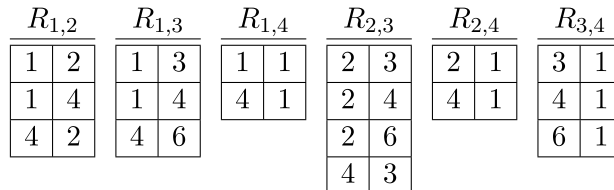


Figure 2.15: PC-8 seeks supports for $(*, 3) \in R_{1,4}$ in D_2. The tuple $(1, 3)$ is removed from $R_{1,4}$ because no support exists in D_2

completes.



$R_{1,2}$	
1	2
1	4
4	2

$R_{1,3}$	
1	3
1	4
4	6

$R_{1,4}$	
1	1
4	1

$R_{2,3}$	
2	3
2	4
2	6
4	3

$R_{2,4}$	
2	1
4	1

$R_{3,4}$	
3	1
4	1
6	1

Figure 2.16: Path-consistent relations of the CSP in Figure 2.8 after execution of PC-8

2.8.4 Introducing Support Bookkeeping (PC-2001)

In this section, we discuss PC-2001 (Algorithm 9) [Bessière *et al.*, 2005]. This algorithm extends, to path consistency, the idea of bookkeeping supports introduced in AC-2001 [Bessière and Régin, 2001; Zhang and Yap, 2001] (see Section 2.7).

PC-2001 operates on the complete constraint graph like the previously studied PC algorithms, PC-2 and PC-8. Like AC-2001, PC-2001 requires that the variables'

Algorithm 9: PC-2001($\mathcal{P}$) *Adapted from* [Bessière *et al.*, 2005]

```
   Input: P = (V, D, C) with a complete constraint graph
   Output: Path consistent P
 1 Q ← ∅
 2 foreach V_i, V_j, V_k ∈ V such that (i < j) ∧ (i ≠ k) ∧ (j ≠ k) do
 3     foreach (a, b) ∈ R_{i,j} do
 4         foreach c ∈ D_k do
 5             if (a, c) ∈ R_{i,k} ∧ (b, c) ∈ R_{j,k} then
 6                 Last((V_i, a), (V_j, b), V_k) ← {c}
 7                 break
 8         if not Last((V_i, a), (V_j, b), V_k) then
 9             R_{i,j} ← R_{i,j} \ {(a, b)}
10             Q ← Q ∪ {((V_i, a), V_j), ((V_j, b), V_i)}
11 while Q do                    /* REVISE_PATH [Bessière et al., 2005] */
12     ((V_i, a), V_k) ← Pop(Q)
13     foreach V_j ∈ V such that (i ≠ j) ∧ (k ≠ j) do
14         foreach (a, b) ∈ R_{i,j} do
15             if i < j then c ← Last((V_i, a), (V_j, b), V_k) else
16                 c ← Last((V_j, a), (V_i, b), V_k)
17             found ← false
18             while c ≤ Final(D_k) do
19                 if (a, c) ∈ R_{i,k} ∧ (b, c) ∈ R_{j,k} then
20                     if i < j then Last((V_i, a), (V_j, b), V_k) ← c else
21                         Last((V_j, a), (V_i, b), V_k) ← c
22                     found ← true
23                     break
24                 c ← Next(D_k, c)
25             if not found then
26                 R_{i,j} ← R_{i,j} \ {(a, b)}
27                 Q ← Q ∪ {((V_i, a), V_j), ((V_j, b), V_i)}
28 return P
```

domains be totally ordered, and it also uses $\textsc{Next}(D_i, a)$ to obtain the value after a in D_i. The function $\textsc{Final}(D_i)$ yields the last element in D_i.

PC-2001 uses a different *Last* structure than AC-2001. $\mathit{Last}((V_i, a), (V_j, b), V_k)$

points to the value in D_k that supports the assignments $\{(V_i, a), (V_j, b)\}$ (i.e., the tuple $(a, b) \in R_{i,j}$). This data structure incurs a large space overhead. Indeed, the space complexity of PC-2001 is $\mathcal{O}(n^3 d^2)$, where n is the number of variables in the problem and d is the maximal domain size [Bessière *et al.*, 2005].

PC-2001 begins by initializing the *Last* structure, as well as making an initial pass through all of the relations and updating the queue (lines 1–10). PC-2001's first pass does the same relation updates and queue additions as PC-8's first pass (lines 1–6). PC-2001 also makes the same revisions and queue additions in the rest of the algorithm (lines 11–27) as are done in the rest of PC-8 (lines 7–13). PC-2001's advantage is that the *Last* structure saves some constraint checks after the initial pass.

We illustrate the operation of PC-2001 on the example shown in Figure 2.13 of Section 2.8.3. When $((V_2, 2), V_1)$ is removed from the queue and a support is sought in V_1 for the tuple $(2, 6) \in R_{2,3}$, $Last((V_2, 2), (V_3, 6), V_1)$ stores, or points to, the value $4 \in D_1$. After that, when seeking a support for the tuple $(2, 6) \in R_{2,3}$ in D_1, the values that precede $4 \in D_1$ are ignored, thus saving some consistency-checking effort.

2.8.5 Partial Path Consistency (BSH-PPC)

Bliek and Sam-Haroud [1999] proposed the BSH-PPC algorithm (Algorithm 10) to enforce PPC on a CSP with a triangulated graph. BSH-PPC uses a queue consisting of constraints, which it initializes to hold hall of the constraints. BSH-PPC removes each constraint, $C_{i,j}$, from the queue and identifies each triangle in the constraint graph that $C_{i,j}$ participates in (line 4). For each triangle (V_i, V_j, V_k), first, Revise-3 (Algorithm 5) is used to revise $C_{i,j}$ with respect to V_k, second, $C_{i,k}$ with respect to V_j, and finally, $C_{j,k}$ with respect to V_i. BSH-PPC adds any revised constraint to the

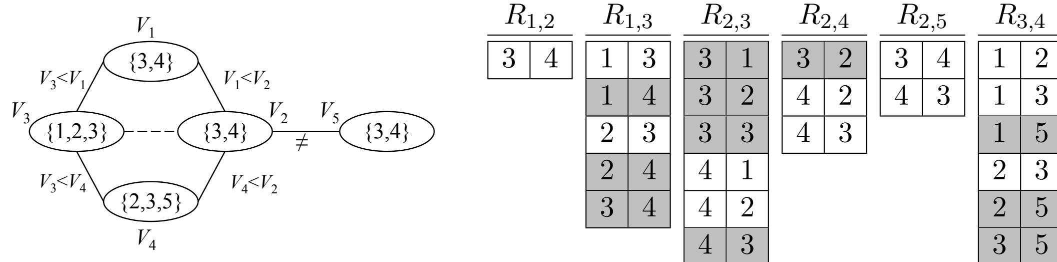
Algorithm 10: BSH-PPC($\mathcal{P}$) *Adapted from* [Bliek and Sam-Haroud, 1999]

Input: $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ with a triangulated constraint graph
Output: Partially path consistent $\mathcal{P}$

1 $\mathcal{Q} \leftarrow \mathcal{C}$
2 **while** $\mathcal{Q}$ **do**
3 $\quad C_{i,j} \leftarrow$ Pop($\mathcal{Q}$)
4 $\quad$ **foreach** $(V_i, V_k, V_j) \in \mathcal{V}$ such that $(C_{i,k} \in \mathcal{C}) \wedge (C_{j,k} \in \mathcal{C})$ **do**
5 $\quad\quad$ **if** Revise-3(V_i, V_j, V_k) **then** $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{C_{i,j}\}$
6 $\quad\quad$ **if** Revise-3(V_i, V_k, V_j) **then** $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{C_{i,k}\}$
7 $\quad\quad$ **if** Revise-3(V_j, V_k, V_i) **then** $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{C_{j,k}\}$
8 **return** $\mathcal{P}$

queue.

We illustrate the operation of BSH-PPC on the example of the triangulated CSP shown in Figure 2.17 (Figures 2.9 and 2.10 previously contained this same example). First, the queue is initialized with all the constraints in the problem.



$R_{1,2}$

3	4

$R_{1,3}$

1	3
1	4
2	3
2	4
3	4

$R_{2,3}$

3	1
3	2
3	3
4	1
4	2
4	3

$R_{2,4}$

3	2
4	2
4	3

$R_{2,5}$

3	4
4	3

$R_{3,4}$

1	2
1	3
1	5
2	3
2	5
3	5

Figure 2.17: *Left*: A triangulated graph, the dashed line is a universal constraint. *Right*: The relations of the CSP. BSH-PPC removes the tuples shown in gray

$$\mathcal{Q} \leftarrow \{C_{1,2}, C_{1,3}, C_{2,3}, C_{2,4}, C_{2,5}, C_{3,4}\}.$$

1. The first item $C_{1,2}$ is popped from the queue. The only triplet considered is (V_1, V_3, V_2). We revise $R_{1,2}$ given V_3, but no tuple is removed from $R_{1,2}$. Then, we examine $R_{1,3}$ against V_2 followed by $R_{2,3}$ against V_1. The tuples of $R_{1,3}$ and $R_{2,3}$ shown in gray in Figure 2.17 are removed at this time. We attempt to add

$C_{1,3}$ and $C_{2,3}$ to $\mathcal{Q}$, but they are already in the queue, which remains unchanged:

$$\mathcal{Q} \leftarrow \{C_{1,3}, C_{2,3}, C_{2,4}, C_{2,5}, C_{3,4}\}.$$

2. The second item, $C_{1,3}$, is popped from the queue. The only triplet considered is (V_1, V_2, V_3). No revision takes place, and no new constraint is added to the queue:

$$\mathcal{Q} \leftarrow \{C_{2,3}, C_{2,4}, C_{2,5}, C_{3,4}\}.$$

3. $C_{2,3}$ is popped from the queue. Two triplets are considered: (V_2, V_3, V_1) and (V_2, V_4, V_3). The examination of (V_2, V_3, V_1) yields no change. The examination of (V_2, V_4, V_3) updates the $R_{2,4}$ and $R_{3,4}$ by removing the tuples shown in gray in Figure 2.17. We attempt to add $C_{2,4}$ and $C_{3,4}$ to $\mathcal{Q}$, but they are already in the queue:

$$\mathcal{Q} \leftarrow \{C_{2,4}, C_{2,5}, C_{3,4}\}.$$

4. $C_{2,4}$ is popped from the queue. The tuple (V_2, V_3, V_4), is considered. No revision takes place. The queue is now:

$$\mathcal{Q} \leftarrow \{C_{2,5}, C_{3,4}\}.$$

5. $C_{2,5}$ is popped from the queue. Because the constraint is not part of any triangle no constraint is examined. The queue is now:

$$\mathcal{Q} \leftarrow \{C_{3,4}\}.$$

6. $C_{3,4}$ is popped from the queue. The tuple (V_3, V_2, V_4), is considered. No revision

takes place. The algorithm terminates.

BSH-PPC considers both of the triangles (V_1, V_3, V_2) and (V_2, V_4, V_3) three times each, which is wasteful. We introduce ΔPPC in Section 4.2 to improve on this behavior.

2.8.6 Strong Conservative Dual Consistency (sCDC1)

Lecoutre *et al.* [2007a] proposed the sCDC1 algorithm (Algorithm 11), which enforces strong conservative dual consistency by filtering both the domains and the relations of the CSP.

Algorithm 11: sCDC1($\mathcal{P}$) — *Adapted from* [Lecoutre *et al.*, 2007a]

```
   Input: P = (V, D, C)                    /* Unaltered constraint graph */
   Output: Strong conservative dual consistent P
 1 AC-2001(P)
 2 ind ← 1
 3 marker ← 1
 4 repeat
 5 |  if sDC1-Check(P, Var(ind)) then
 6 |  |  AC-2001-core(P, {Var(ind)})
 7 |  |  marker ← ind
 8 |  ind ← ind + 1
 9 |  if ind > |V| then ind ← 1
10 until ind = marker
11 return P
```

sCDC1 is conservative, meaning it does not require the input problem to be triangulated, completed, or otherwise changed. It *does not add any edges to the graph.* It does assume that the variables are ordered and given an index starting with 1 and going to n. It uses the following data structures and functions:

- Index(V_i) yields the index of the variable V_i.
- *Var*(*ind*) is a vector that facilitates accessing each variable by its index *ind*.

Algorithm 12: sCDC1-Check($\mathcal{P}, V_x$) *Adapted from* [Lecoutre *et al.*, 2007a]

```
   Input: P = (V, D, C), V_x ∈ V
   Output: true if D_x was updated, false otherwise
1  modified ← false
2  foreach a ∈ D_x do
3      P′ = (V′, D′, C′) ← Copy(P)
4      V′_x ← a
5      AC-2001-core(P′, {V′_x})
6      if P′ is consistent then
7          foreach V_y ∈ V such that C_{x,y} ∈ C do
8              foreach b ∈ D_y such that b ∉ D′_y do
9                  if (a, b) ∈ R_{x,y} then
10                     R_{x,y} ← R_{x,y} \ {(a, b)}
11                     modified ← true
12     else
13         D_x ← D_x \ {a}
14         modified ← true
15 return modified
```

- Copy($\mathcal{P}$) generates a copy of all the variables, domains, and constraints of $\mathcal{P}$.[9]

sCDC1 uses two position pointers, *ind* and *marker*, which are initialized to 1. *marker* records the index of the last variable that caused an update, and *ind* keeps the index of the current variable.

The main loop of sCDC1 (lines 4–10) continues until the next variable to process is the last one that made an update. sCDC1 begins by enforcing AC on the entire problem. However, to illustrate the operation of sCDC1, we run it on the example in Figure 2.18 without first enforcing AC in order to maintain the simplicity of the example. (Otherwise, sCDC1 does not yield any filtering beyond that done by AC on this example.) First, sCDC1 passes sCDC1-Check (Algorithm 12) the variable V_1.

[9]In practice, we found it is more efficient to keep only one copy of the problem in memory and to apply the AC algorithm while keeping undo records. We use AC2001 in our implementation and restore the domains in memory using the undo records after each singleton test, as well as using undo records for the *Last* structure.

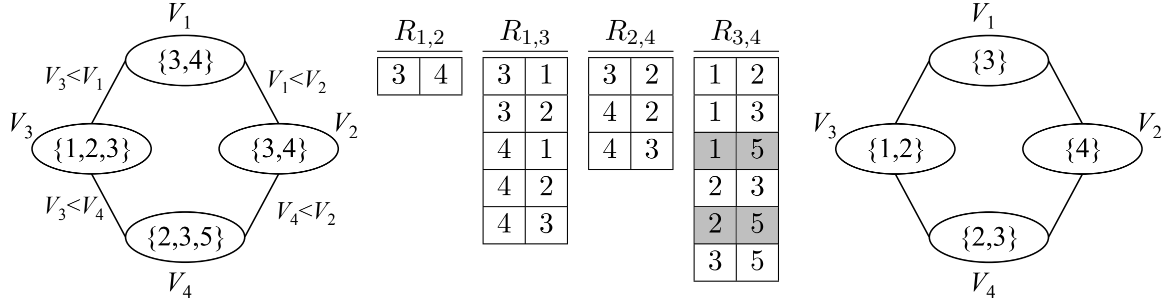


Figure 2.18: *Left*: The CSP before processing. *Middle*: Relations as revised by sCDC1, removed tuples are shown in gray. *Right*: The CSP after processing

sCDC1-Check makes instantiation $V_1' \leftarrow 3$ in $\mathcal{P}'$'. AC determines that the resulting problem, $\mathcal{P}'$ is consistent and updates the domains $D_2' = \{4\}, D_3' = \{1,2\}, D_4' = \{2,3\}$. Line 7 examines V_2 then V_3, which are the neighbors of V_1. Because the tuple $(3,3)$ does not appear in $R_{1,2}$, the relation is not altered; the same applies for $R_{1,3}$ (lines 8–11). When considering $V_1' \leftarrow 4$, AC determines inconsistency and value 4 is removed from D_1 (the original domain).

Next, we call sCDC1-Check on V_2. There, we first consider $V_2' \leftarrow 3$. AC determines inconsistency. Thus, the original domain gets updated, $D_2 \leftarrow \{4\}$. Then, we consider $V_2' \leftarrow 4$. sCDC1-Check proceeds without updating any domain or relation.

Next, V_3 is passed to sDC1-Check. First, we consider $V_3' \leftarrow 1$. AC determines $\mathcal{P}'$ is consistent and updates the domains $D_1' = \{3\}, D_2' = \{4\}, D_4' = \{2,3\}$. Note that value 5 is removed from $D_4' = \{2,3\}$. Thus, the tuple $(1,5)$ is removed from $R_{3,4}$. Moving to $V_3' \leftarrow 2$, we determine that the tuple $(2,5)$ must be removed from $R_{3,4}$. Finally, considering $V_3' \leftarrow 3$, AC yields failure, and $D_3 \leftarrow \{1,2\}$.

Next, sDC1-Check is passed V_4. There, no filtering occurs when considering $V_4' \leftarrow 2$ and $V_4' \leftarrow 3$. When considering $V_4' \leftarrow 5$, AC detects inconsistency and $D_4 \leftarrow \{2,3\}$.

Then, sDC1-Check is passed V_1, V_2, and V_3 in sequence without causing any updates. Thus, sDC1-Check does not need to be passed V_4 again. And the algorithm terminates.

2.8.7 Strong Dual Consistency (sDC2)

McGregor [1979] proposed an algorithm for enforcing strong path consistency by using arc consistency as a 'basic tool.' Lecoutre *et al.* [2007b] incorporated this mechanism into their algorithm sDC2 (Algorithm 13), which enforces strong dual consistency on the *completed* graph. sDC2 operates by running two singleton tests on each of the two variable-value pairs of a tuple that is consistent with the constraint defined over the two variables (i.e., a support tuple).

It is difficult to provide a simple walk through the pseudocode of sDC2. It suffices to say that, in addition to singleton checks, sDC2 calls a forward-checking procedure (FC-2001, Algorithm 15) and an arc-consistency procedure (AC-2001-core, Algorithm 2).

Further, in addition to the functions and data structures used by sCDC1 (see Section 2.8.6), sDC2 uses *dom*(*ind*), which gives the domain of the variable whose index is *ind*. Like sCDC1, sDC2 uses *marker* that allows it to continue iterating over the variables in the problem until it completes a revolution over the variables without causing an update. It introduces a *LastModified* structure to keep track of when each variable was last affected by a change, where a change is either an update of its domain or that of the relation of any constraint defined over the variable. At lines 9 and 10 of sDC2, *totalvals* stores the total number of values in the variables' domains. At lines 14–17, if the total the number of values has changed, every variable is marked as modified. It would be possible to mark a smaller set of variables, but

Algorithm 13: sDC2($\mathcal{P}$) *Adapted from* [Lecoutre *et al.*, 2007b]

```
   Input: P = (V, D, C) with a complete constraint graph
   Output: Strong dual consistent P
 1 AC-2001(P)
 2 ind ← 0
 3 marker ← 0
 4 count ← 0
 5 foreach V_i ∈ V do LastModified(V_i) ← 0
 6 repeat
 7     count ← count + 1
 8     if |dom(ind)| > 1 then
 9         totalvals ← 0
10         foreach D_i ∈ D do totalvals ← totalvals + |D_i|
11         if sDC2-Check(P,Var(ind), count) then
12             LastModified(Var(ind)) ← count
13             AC-2001-core(P, {Var(ind)})
14             newtotalvals ← 0
15             foreach D_i ∈ D do newtotalvals ← newtotalvals + |D_i|
16             if totalvals ≠ newtotalvals then
17                 foreach V_i ∈ V do LastModified(V_i) ← count
18             marker ← ind
19     ind ← ind + 1
20     if ind ≥ |V| then ind ← 0
21 until ind ≠ marker
22 return P
```

Lecoutre *et al.* [2007b] found that this operation would increase complexity of the implementation without any noticeable effect.

Whenever sDC2 calls sDC2-Check (Algorithm 14) on a given variable V_x, AC-2001 is run on a copy of the problem the first time that sDC2-Check is passed V_x. After this step, forward checking is done on all the variables connected to V_x. Then, AC-2001-core only needs to be run on variables that have changed since they were last passed to sDC2-Check.

Algorithm 14: sDC2-Check($\mathcal{P}, V_x, count$) *Adapted from* [Lecoutre *et al.*, 2007b]

Input: $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C}), V_x, count$
Output: *true* if D_x or any constraint of V_x is modified, *false* otherwise

1 $modified \leftarrow false$
2 **foreach** $a \in D_x$ **do**
3 $\quad \mathcal{P}' = (\mathcal{V}', \mathcal{D}', \mathcal{C}') \leftarrow$Copy($\mathcal{P}$)
4 $\quad V'_x \leftarrow a$ `/* Variable assignment */`
5 $\quad$ **if** $count \leq |\mathcal{V}|$ **then** AC-2001-core($\mathcal{P}', \{V'_x\}$)
6 $\quad$ **else**
7 $\quad\quad$ FC-2001($\mathcal{P}', \{V'_x\}$)
8 $\quad\quad$ **if** $\mathcal{P}'$ *is consistent* **then**
9 $\quad\quad\quad \mathcal{Y} \leftarrow \emptyset$
10 $\quad\quad\quad$ **foreach** $V_y \in \mathcal{V}$ **do**
11 $\quad\quad\quad\quad$ **if** $count - LastModified(V_y) < |\mathcal{V}|$ **then** $\mathcal{Y} \leftarrow \mathcal{Y} \cup \{V_y\}$
12 $\quad\quad\quad$ AC-2001-core($\mathcal{P}', \mathcal{Y}$)
13 $\quad$ **if** $\mathcal{P}'$ *is consistent* **then**
14 $\quad\quad$ **foreach** $V_y \in \mathcal{V}$ such that $C_{x,y} \in \mathcal{C}$ **do**
15 $\quad\quad\quad$ **foreach** $b \in V_y$ such that $(b \notin V'_y)$ **do**
16 $\quad\quad\quad\quad$ **if** $\{a, b\} \in R_{x,y}$ **then**
17 $\quad\quad\quad\quad\quad R_{x,y} \leftarrow R_{x,y} \setminus \{(a, b)\}$
18 $\quad\quad\quad\quad\quad LastModified(V_y) \leftarrow count$
19 $\quad\quad\quad\quad\quad modified \leftarrow true$
20 $\quad$ **else**
21 $\quad\quad D_x \leftarrow D_x \setminus \{a\}$
22 $\quad\quad modified \leftarrow true$
23 **return** *modified*

Algorithm 15: FC-2001($\mathcal{P}, V_x$) *Adapted from* [Lecoutre *et al.*, 2007b]

Input: $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C}), V_x$
Output: Forward-checked $\mathcal{P}$

1 $\mathcal{Q} \leftarrow \emptyset$
2 **foreach** $V_i \in \mathcal{V}$ such that $C_{i,x} \in \mathcal{C}$ **do** $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{(V_i, V_x)\}$
3 **while** $\mathcal{Q}$ **do**
4 $\quad (V_i, V_j) \leftarrow$Pop($\mathcal{Q}$)
5 $\quad$ Revise-2001(V_i, V_j)
6 **return** $\mathcal{P}$

2.9 Path Consistency Algorithms in Practice

Look-ahead techniques, which enforce consistency during search, typically use arc-consistency but not path-consistency algorithms because the latter are typically computationally expensive and modify the structure of the constraint graph.

Temporal networks with convex constraints are the only constraint networks that path consistency is enforced on in practice. For convex constraints it is known that partial path consistency guarantees path consistency [Bliek and Sam-Haroud, 1999] and that path consistency guarantees that the problem can be solved in a backtrack-free manner Dechter:91:

1. Indeed, Xu and Choueiry [2003] proposed the algorithm ΔSTP to solve the the Simple Temporal Problem by improving the operation of the BSH-PPC algorithm of Bliek and Sam-Haroud [1999] and specializing it to the STP. Further, they noted the dependency of the performance of ΔSTP on the ordering of the triangles in the propagation queue. They attribute to a private communication with Nic Wilson in 2005 the realization that a perfect elimination ordering of the vertices of the graph and a tree decomposition control the performance ΔSTP and can be used to predict a tight bound for time complexity [Bui *et al.*, 2007].

2. Planken *et al.* proposed P^3C as an improvement of ΔSTP [2008]. Further, they provided the theoretical characterization of the complexity of the algorithm exploiting a perfect elimination ordering.

3. Recently, Long *et al.* [2016] generalized P^3C into DPC+, which they applied to qualitative constraint networks for spatio-temporal reasoning.[10]

[10] DPC+ simply replaces the scalar *minimum* and *addition* operators of ΔSTP with the relational operators *intersection* and *composition* respectively.

In Boolean satisfiability (SAT), the generation of binary clauses for variable elimination by resolution is equivalent to a path-consistency operation and the basis of the powerful pre-processing in modern SAT solvers [Eén and Biere, 2005].

In this thesis, we consider algorithms that operate on arbitrary constraints and do not restrict ourselves to any type of constraints.

Summary

In this chapter, we reviewed background information about CSPs. In particular, we discussed several consistency properties and reviewed in great detail the consistency algorithms that we use in our empirical evaluation.

Chapter 3

Gaps in the Literature

In this chapter, we discuss various imprecisions and errors that have appeared in the literature and that are relevant to anyone interested in path consistency and algorithms for enforcing it.

In particular, we discuss the definition of the partial path consistency property. We show that conservative path consistency and partial path consistency are not equivalent on triangulated graphs, contrary to what is stated by Lecoutre *et al.* [2011]. We report an error in the pseudocode of the PPC algorithm reported by Bliek and Sam-Haroud [1999].[1] Finally, we give an example where PC detects inconsistency but PPC fails to (a case that was never encountered in practice before).

3.1 On the Definition of PPC

In order to introduce PPC, Bliek and Sam-Haroud [1999] distinguish between enforcing PC on a CSP and on the constraint graph. They state the following propositions:

[1]Note that the error was fixed in the implementation of Bliek and Sam-Haroud [1999] and, thus, does not affect their experiments (private communication).

1. A constraint graph is PC iff all paths in the graph are PC [Montanari, 1974; Mackworth, 1977].

2. A CSP is PC if the completion of its constraint graph is PC [Montanari, 1974; Mackworth, 1977].

3. A CSP is PPC if its (original) constraint graph is PC.

4. A triangulated constraint graph is PC iff every path of length two is PC.

Then, the authors give an algorithm, which we call BSH-PPC,[2] that takes a constraint graph as input, and triangulates the graph while ensuring that every path of length two in the triangulated graph is PC. Thus, the algorithm ensures that the triangulated constraint graph is PC and, consequently, the CSP is PPC.

If we run BSH-PPC on the example shown in Figure 3.1, the algorithm determines that the CSP is PPC. Thus, we should expect that *all paths* in the graph are

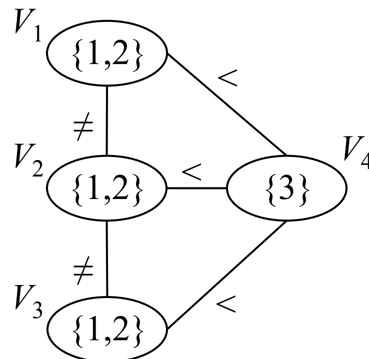


Figure 3.1: A CSP that is PPC because every *closed* graph-path is PC

PC. However, if we consider the instantiation $\{(V_1, 1), (V_3, 2)\}$ we can easily see that it is *not* consistent along the path (V_1, V_2, V_3). Consequently, we conclude that the definitions of Bliek and Sam-Haroud [1999] do not correctly characterize the consistency property PPC, and we choose to use instead the definition of Lecoutre *et al.*

[2] See Section 2.8.5.

[2011],[3] which restricts the paths on which the PC property must hold to only closed graph-paths.

3.2 CPC and PPC on Triangulated Graphs

Below, we report two errors by Lecoutre *et al.* [2011].

3.2.1 Incorrect Example

Lecoutre *et al.* [2011] provided the example shown in Figure 3.2 to illustrate a constraint network where "every 2-length graph-path is [...] consistent."[4] However, the path (V_2, V_3, V_5) is not consistent for the locally consistent instantiation $\{(V_2, b), (V_5, b)\}$.

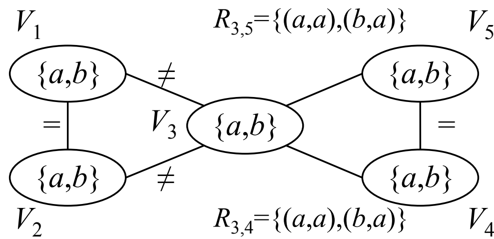


Figure 3.2: An incorrect example with the inconsistent two-length path (V_2, V_3, V_5) [Lecoutre *et al.*, 2011]

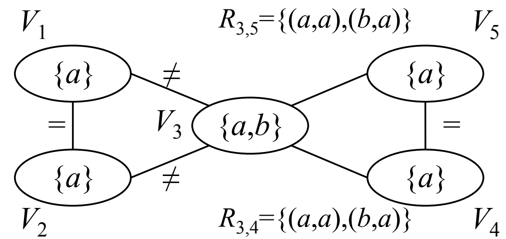


Figure 3.3: A triangulated CSP that is CPC, but not PPC

We propose to correct this example by setting $D_1 = D_2 = D_4 = D_5 = \{a\}$, as shown in Figure 3.3.

[3] See Definition 2.5.5 in Section 2.5.2.
[4] See Figure 4 on page 185 [Lecoutre *et al.*, 2011].

3.2.2 Incorrect Proposition

Further, Lecoutre *et al.* [2011] state an incorrect proposition:[5]

> **Proposition 4.** *(Bliek & Sam-Haroud 1999) Let P be a binary CN P with a triangulated constraint graph. P is PPC-consistent iff P is CPC-consistent.*

This proposition has the following errors:

1. First, no such statement appears in the paper of Bliek and Sam-Haroud [1999].
2. Second, the CSP shown in Figure 3.3 is a counterexample to Proposition 4 of Lecoutre *et al.* [2011]. Indeed, the CSP is triangulated and CPC but it is not PPC because the closed graph-path (V_3, V_1, V_3, V_4) is not path consistent for the assignment $((V_3, a), (V_4, a))$.

Thus, it is incorrect to say that PPC and CPC are equivalent on triangulated graphs.

3.3 Imprecision in the Pseudocode of BSH-PPC

The pseudocode of BSH-PPC provided by Bliek and Sam-Haroud [1999] iterates over every combination of three connected variables (V_i, V_k, V_j), called *Related-Triplets,*"that correspond to actual triangles in" the constraint graph. It is not clear whether or not two vertices in this triangle can be equal.

- If we consider that the three vertices must be distinct, then the algorithm fails to detect the inconsistency of the example shown in Figure 3.4.

[5] See Proposition 4 on page 186 [Lecoutre *et al.*, 2011].

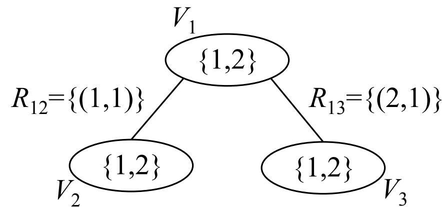


Figure 3.4: BSH-PPC cannot detect the inconsistency of the instantiation $\{(V_2, 1), (V_1, 1)\}$ along the path (V_2, V_1, V_3, V_1) if it considers only triangles with distinct vertices

- If we allow two vertices to be the same, then processing the triangle (V_1, V_2, V_1) updates $D_1 \leftarrow \{1\}$. Further, processing the triangle (V_1, V_3, V_1) updates $D_1 \leftarrow \emptyset$, thus detecting inconsistency.

However, considering all triangles with two variables that are the same is not necessary. Instead, we should simply pay special attention to edges incident to articulation points in the triangulated graph. Thus, assuming that all triplets considered are defined over distinct variables, then the algorithm can be fixed by enforcing arc consistency on edges incident to articulation points[6] as we do in PPC+AP (Algorithm 16) in Section 4.1, Chapter 4.

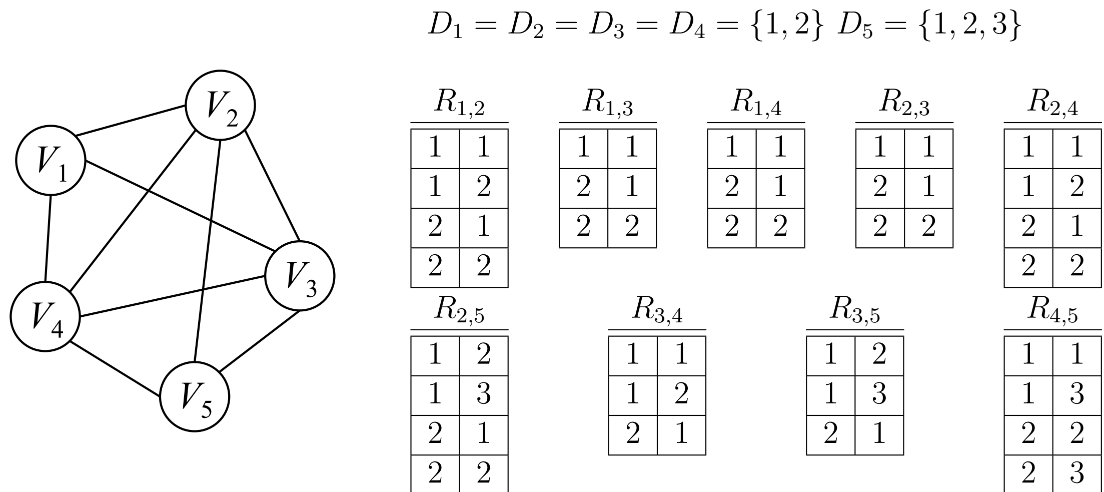
A private communication with the authors confirmed that, indeed, *their code singles out articulation points and that their reported experimental results are correct*. They acknowledged the shortcoming of the pseudocode in the paper.

Note that this imprecision was not detected by previous research adapting BSH-PPC because they are restricted to temporal constraint networks, which ignore the variables' domains [Xu and Choueiry, 2003; Planken *et al.*, 2008].

[6]This problem was first identified by Christopher Thiel in Spring 2008.

3.4 Comparing PC and PPC

Lecoutre *et al.* [2011] showed that PC is strictly stronger than CDC, and that CDC is strictly stronger than PPC. They provide the example in Figure 3.5 that is PPC but not CDC.[7] Applying a PC or CDC algorithm to this instance removes the tuple $(1, 2)$ from the relation $R_{1,2}$, whereas enforcing PPC yields no filtering.



$R_{1,2}$	
1	1
1	2
2	1
2	2

$R_{1,3}$	
1	1
2	1
2	2

$R_{1,4}$	
1	1
2	1
2	2

$R_{2,3}$	
1	1
2	1
2	2

$R_{2,4}$	
1	1
1	2
2	1
2	2

$R_{2,5}$	
1	2
1	3
2	1
2	2

$R_{3,4}$	
1	1
1	2
2	1

$R_{3,5}$	
1	2
1	3
2	1

$R_{4,5}$	
1	1
1	3
2	2
2	3

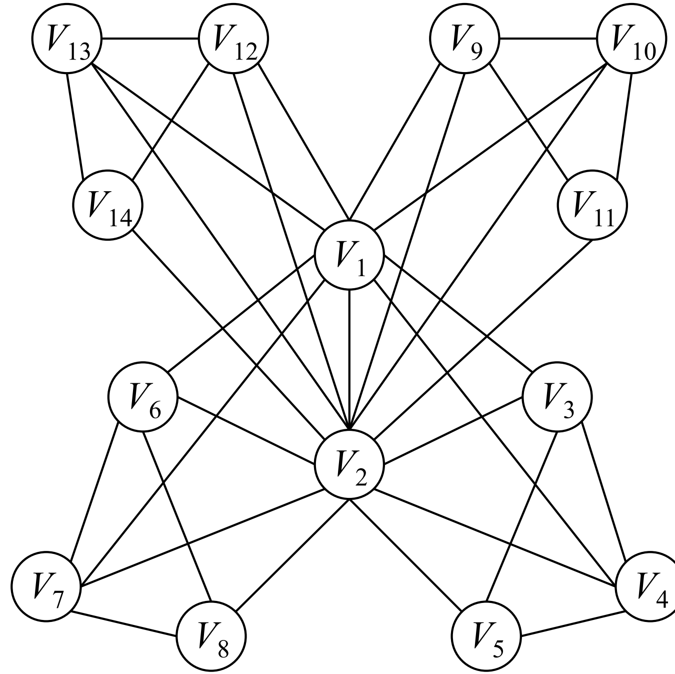
Figure 3.5: A binary CSP that is PPC but not CDC

In practice, enforcing either PPC or PC on a CSP was found to detect *inconsistency* on exactly the same instances as noted by Bliek and Sam-Haroud [1999] and also in our experiments. Indeed, Bliek and Sam-Haroud wrote:

> *For the tests conducted on random problems, insolubility detected by PC was also detected by PPC.*

Since 1999, it was an open question whether or not a PC algorithm can detect inconsistency of an instance that is PPC. The example in Figure 3.6, with the relation definitions reported in Figure 3.7, provides, for the first time, a counterexample that settles this question.

[7] Figure 9 of their paper.



$\forall i \in \{1, 2, 3, 4, 6, 7, 9, 10, 12, 13\}\ D_i = \{1, 2\}$
$D_5 = D_8 = D_{11} = D_{14} = \{1, 2, 3\}$

Figure 3.6: Enforcing PC uncovers the inconsistency of the CSP, but enforcing PPC yields no filtering

$R_{1,2}$	
1	1
1	2
2	1
2	2

$R_{1,3}$	
1	1
2	1
2	2

$R_{2,5}$	
1	3
2	1
2	2

$R_{3,4}$	
1	1
1	2
2	1

$R_{3,5}$	
1	2
1	3
2	1

$R_{4,5}$	
1	1
1	3
2	2
2	3

$R_{2,8}$	
1	1
1	2
2	2
2	3

$$R_{1,2} = R_{2,4} = R_{2,7} = R_{2,10} = R_{2,13} = \{1, 2\} \times \{1, 2\}$$
$$R_{1,3} = R_{1,4} = R_{1,6} = R_{1,7} = R_{2,3} = R_{2,12} = \{(1, 1), (2, 1), (2, 2)\}$$
$$R_{2,5} = R_{2,14} = \{(1, 3), (2, 1), (2, 2)\}$$
$$R_{3,4} = R_{6,7} = R_{9,10} = R_{12,13} = R_{1,9} = R_{1,10} = R_{1,12} = R_{1,13} = R_{2,6} = R_{2,9} = \{(1, 1), (1, 2), (2, 1)\}$$
$$R_{3,5} = R_{6,8} = R_{9,11} = R_{12,14} = \{(1, 2), (1, 3), (2, 1)\}$$
$$R_{4,5} = R_{7,8} = R_{10,11} = R_{13,14} = \{(1, 1), (1, 3), (2, 2), (2, 3)\}$$
$$R_{2,8} = R_{2,11} = \{(1, 1), (1, 2), (2, 2), (2, 3)\}$$

Figure 3.7: Relations of the CSP in Figure 3.6

Summary

In this chapter, we cleared various imprecisions, errors, and omissions that appeared in the literature concerning path consistency and its closely related properties.

Chapter 4

New Algorithms for (Partial) Path Consistency

In this chapter, we introduce new algorithms for enforcing PPC and PC. The properties and characteristics of those algorithms are summarized in Tables 4.1 and 4.2.

Table 4.1: Summary of the introduced algorithms

	Section	**Graph**	**Property**	**Pruning w.r.t. PC-2**
PPC+AP	Section 4.1	Triangulated	PPC	Less than PC-2, more than BSH-PPC
ΔPPC	Section 4.2	Triangulated	PPC	Less than PC-2, more than BSH-PPC
σ-ΔPPC	Section 4.3	Triangulated	PPC	Less than PC-2, more than BSH-PPC
PC-8^+	Section 4.4	Complete	PC	same as PC-2

PPC+AP corrects the error of the BSH-PPC algorithm (Section 2.8.5); ΔPPC improves the performance of PPC+AP by processing all the edges of a triangle at once; σ-ΔPPC follows a perfect elimination ordering for processing the triangles in the triangulated graph and uses support structures to reduce the number of con-

Table 4.2: Data structures and propagation queues of the introduced algorithms

	Supports	Element of queue	Pseudocode
PPC+AP	None	$C_{i,j}$	PPC+AP (Algorithm 16) REVISE-3+AP (Algorithm 19) PROPAGATE-APS (Algorithm 17) FILTER-DOM (Algorithm 7) FILTER-RELS (Algorithm 18)
ΔPPC	None	(V_i, V_j, V_k) $i < j < k$	ΔPPC (Algorithm 20) REVISE-TRIANGLE (Algorithm 21) REVISE-3+AP (Algorithm 19) PROPAGATE-APS (Algorithm 17) FILTER-DOM (Algorithm 7) FILTER-RELS (Algorithm 18)
σ-ΔPPC	A set of three tuples	(V_k, V_j, V_i) $i < j < k$	σ-ΔPPC (Algorithm 23) INIT-TRIANGLE-VECTOR (Algorithm 22) REVISE-TRIANGLE (Algorithm 21) REVISE-3+AP (Algorithm 19) PROPAGATE-APS (Algorithm 17) FILTER-DOM (Algorithm 7) FILTER-RELS (Algorithm 18)
PC-8^+	(V_i, a, V_j, b)	$((V_i, a), V_k)$	REVISE-PC-8-FLAG (Algorithm 25) PC-8-FLAG (Algorithm 24) PC-8-ORDERING (Algorithm 26) PC-8^+ (Algorithm 27)

straint checks; finally, PC-8^+ introduces two improvements for the PC-8 algorithm (Section 2.8.3).

We first discuss each algorithm, then we discuss the time and space complexities of all introduced algorithms.

4.1 Correcting BSH-PPC (PPC+AP)

PPC+AP (Algorithm 16) is our proposed correction of the BSH-PPC algorithm.

Like BSH-PPC, PPC+AP requires that the constraint graph of the input problem be triangulated and iterates over a propagation queue of edges. Correcting BSH-PPC, PPC+AP ensures the correct propagation across the articulation points and the cut edges in the graph.[1]

Algorithm 16: PPC+AP($\mathcal{P}$)

Input: $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ with a triangulated constraint graph
Output: Partial-path consistent $\mathcal{P}$

```
1  Q ← C
2  A_points ← ARTICULATION-POINTS(P)
3  Q_AP ← A_points
4  PROPAGATE-APS(A_points, Q_AP)
5  while Q do
6      C_{i,j} ← POP(Q)
7      foreach (V_i, V_k, V_j) ∈ V such that (C_{i,k} ∈ C) ∧ (C_{j,k} ∈ C) do
8          Q ← Q ∪ REVISE-3+AP(V_i, V_j, V_k, A_points)
9          Q ← Q ∪ REVISE-3+AP(V_i, V_k, V_j, A_points)
10         Q ← Q ∪ REVISE-3+AP(V_j, V_k, V_i, A_points)
11 return P
```

More specifically, PPC+AP first identifies the articulation points (line 2). Then, Propagate-APs (Algorithm 17) 'synchronizes' the domains of the articulation points with the relations of the constraints incident to them using the functions Filter-Dom (Algorithm 7) and Filter-Rels (Algorithm 18), which implement the projection and selection operations. Propagate-APs handles also propagation along any cut edges.

For every constraint in the propagation queue, PPC+AP calls Revise-3+AP (Algorithm 19) on every 'triplet relations' in which the constraint appears. However, unlike BSH-PPC, Revise-3+AP includes a special processing for articulation points and cut edges.

---

[1] See discussion in Section 3.3.

Algorithm 17: PROPAGATE-APS($\mathcal{A}_{points}, \mathcal{Q}_{AP}$)

Input: $\mathcal{A}_{points}$: Articulation vertices
$\mathcal{Q}_{AP}$: Queue of articulation points
Output: $\mathcal{U}_{edges}$: Set of non-cut edges that have been updated

```
1  U_edges ← ∅
2  while Q_AP do
3      foreach V_i ∈ Q_AP do FILTER-DOM(D_i, {C_i ∈ C | V_i ∈ scope(C_i)})
4      Q_orig ← Q_AP
5      Q_AP ← ∅
6      foreach V_i ∈ Q_orig do
7          Q_e ←FILTER-RELS({C_i ∈ C | V_i ∈ scope(C_i)}, D_i)
8          foreach C_i,j ∈ Q_e do
9              if V_j ∈ A_points then Q_AP ← Q_AP ∪ {V_j}
10             else U_edges ← U_edges ∪ {C_i,j}
11 return U_edges
```

Algorithm 18: FILTER-RELS($\mathcal{C}_i, D_i$)

Input: $\mathcal{C}_i$: Set of constraints incident to variable V_i
D_i: Domain of variable V_i
Output: $\mathcal{C}_u$: Set of updated constraints

```
1  C_u ← ∅
2  foreach C ∈ C_i do
3      R^old ← rel(C)
4      R ← σ_{V_i∈D_i}(R^old)
5      if R ≠ R^old then
6          rel(C) ← R
7          C_u ← C_u ∪ {C}
8  return C_u
```

An implementation of ARTICULATION-POINTS is not given and can be found elsewhere [Hopcroft and Tarjan, 1973].

Algorithm 19: REVISE-3+AP($V_i, V_j, V_k, \mathcal{A}_{points}$)

Input: $V_i, V_j, V_k \in \mathcal{V}, \mathcal{A}_{points}$: Articulation vertices
Output: $\mathcal{Q}_u$: Queue of updated constraints

```
1 Q_u ← ∅
2 if REVISE-3(V_i, V_j, V_k) then
3     Q_u ← Q_u ∪ {C_{i,j}}
4     if V_i ∈ A_points then
5         Q_u ← Q_u ∪ PROPAGATE-APS(A_points, {V_i})
6     else if V_j ∈ A_points then
7         Q_u ← Q_u ∪ PROPAGATE-APS(A_points, {V_j})
8 return Q_u
```

4.2 Exploiting Triangles ($\triangle$PPC)

In Section 2.8.5, we illustrated the operation of the BSH-PPC algorithm (Algorithm 10) on the example shown in Figure 4.1. BSH-PPC loops through a *queue of constraints*. Consequently, it examines three times each of the two triangles in Figure 4.1 and ends up calling the algorithm REVISE-3 18 times even though no change occurred between the two calls to REVISE-3. It is obvious that the performance of BSH-PPC can be significantly improved. In a manner similar to the $\triangle$STP algorithm by Xu and Choueiry [2003], we propose $\triangle$PPC (Algorithm 20), which improves on the performance of PPC+AP as follows:

1. $\triangle$PPC uses a queue of *triangles* instead of the queues of edges used in PPC+AP and BSH-PPC.

2. Every time we pop an element from the queue, REVISE-TRIANGLE (Algorithm 21) revises all three edges of the triangle at the same time.

3. Whenever an edge is updated, we add to the queue, unless they are already there, all the triangles in which the edge appears except the triangle under consideration.

To this end, we need three data structures:

1. A table *TrianglesEdge* storing for each edge, the list of triangles in which the edge appears. Such table has at most $\mathcal{O}(n^2)$ entries with $\mathcal{O}(n)$ items each, where n is the number of variables in the CSP.

2. A table storing all the articulation points, whose size is $\mathcal{O}(n)$.

3. A propagation queue $\mathcal{Q}_t$ of triangles, whose size is $\mathcal{O}(n^3)$.

Like BSH-PPC and PPC+AP, ΔPPC operates on a triangulated graph and integrates the processing of the articulation points introduced in PPC+AP.

Algorithm 20: ΔPPC($\mathcal{P}$)

Input: $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ with a triangulated constraint graph
Output: Partially path consistent $\mathcal{P}$

```
1  A_points ←ARTICULATION-POINTS(P)
2  Q_AP ← A_points
3  PROPAGATE-APS(A_points, Q_AP)
4  Q_t ← {(V_i, V_j, V_k) ∈ V^3 | (i < j < k) ∧ (C_{i,j}, C_{i,k}, C_{j,k} ∈ C)}
5  while Q_t do
6  |   (V_i, V_j, V_k) ←PEEK(Q_t)
7  |   Q_e ←REVISE-TRIANGLE(V_i, V_j, V_k, A_points)
8  |   while Q_e do
9  |   |   edge ←POP(Q_e)
10 |   |_  Q_t ← Q_t ∪ TrianglesEdge(edge)
11 |_  POP(Q_t)
12 return P
```

We illustrate the operation of ΔPPC on the example of the triangulated CSP shown in Figure 4.1.[2]

First, line 1 of ΔPPC identifies V_2 as the unique articulation point. Thus, Filter-Dom (line 2) filters removes values 3 from D_2; and Filter-Rels removes

[2]This example was shown in Figures 2.9, 2.10, and 2.17.

Algorithm 21: Revise-Triangle($V_i, V_j, V_k, \mathcal{A}_{points}$)

Input: $V_x, V_y, V_z \in \mathcal{V}$, $\mathcal{A}_{points}$: articulation vertices
Output: $\mathcal{Q}_e$: Queue of updated constraints

1. $\mathcal{Q}_e \leftarrow \emptyset$
2. $\mathcal{Q}_e \leftarrow \mathcal{Q}_e \cup$ Revise-3+AP($V_i, V_j, V_k, \mathcal{A}_{points}$)
3. $\mathcal{Q}_e \leftarrow \mathcal{Q}_e \cup$ Revise-3+AP($V_i, V_k, V_j, \mathcal{A}_{points}$)
4. $\mathcal{Q}_e \leftarrow \mathcal{Q}_e \cup$ Revise-3+AP($V_j, V_k, V_i, \mathcal{A}_{points}$)
5. **return** $\mathcal{Q}_e$

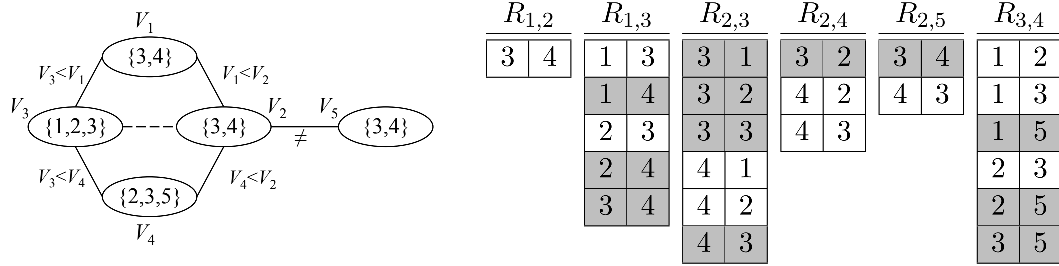


Figure 4.1: A triangulated graph, the dashed line denotes a universal constraint

$R_{1,2}$

3	4

$R_{1,3}$

1	3
1	4
2	3
2	4
3	4

$R_{2,3}$

3	1
3	2
3	3
4	1
4	2
4	3

$R_{2,4}$

3	2
4	2
4	3

$R_{2,5}$

3	4
4	3

$R_{3,4}$

1	2
1	3
1	5
2	3
2	5
3	5

Figure 4.2: The relations. Gray tuples are removed by ΔPPC

$\{(3,1),(3,2),(3,3)\}$ from $R_{2,3}$, the tuple $(3,2)$ from $R_{2,4}$, and the tuple $(3,4)$ from $R_{2,5}$. (Note that BSH-PPC does not remove $(3,4)$ from $R_{2,5}$.)

The propagation queue $\mathcal{Q}_t$ is initialized with two the triangles in the problem:

$$\mathcal{Q}_t \leftarrow \{(V_1, V_2, V_3), (V_2, V_3, V_4)\}.$$

The first triangle, (V_1, V_2, V_3), is copied from the queue. Revising $C_{1,2}$ with respect to V_3 removes no tuples. Revising $R_{1,3}$ given V_2 yields the removal to the tuples {(1,4),(2,4),(3,4)}, shown in grey in Figure 4.2. Revising $R_{2,3}$ given V_1 removes the tuple $(4,3)$ from $R_{2,3}$. Because V_2 is an articulation point, Filter-Dom and Filter-Rels are called again, but yield no further revisions. Now, the processed triangle is removed from $\mathcal{Q}_t$, and we peek at the second triangle in $\mathcal{Q}_t$. Considering the triangle (V_2, V_3, V_4), we update $R_{3,4}$ removing the tuples $\{(1,5),(2,5),(3,5)\}$ as shown in grey in Figure 4.2. The examination of the two other relations yields no updates. We pop

(V_2, V_3, V_4) from $\mathcal{Q}_t$, which becomes empty and the algorithm terminates.

4.3 σ-ΔPPC

Below, we introduce the main contribution of this thesis, σ-ΔPPC, an algorithm that enforces partial path consistency by correcting then improving the operations of the BSH-PPC algorithm.

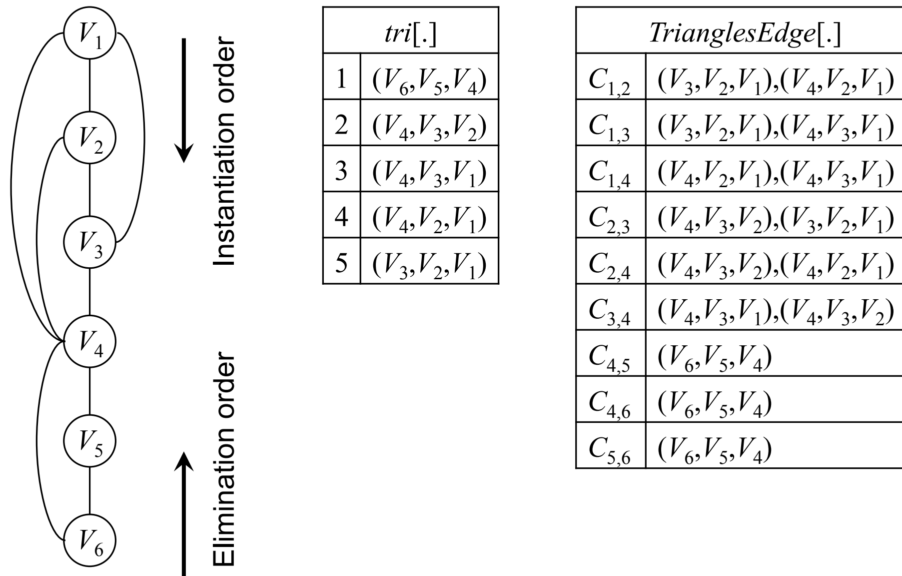
4.3.1 Idea and Data Structures

In order to reduce the number of revisions during constraint propagation and quickly reach a fixpoint (i.e., quiescence), one may want to follow the structure of a tree decomposition of the constraint graph of the CSP [Dechter, 2003]. This operation can be achieved using a perfect elimination ordering of the constraint graph. The idea is to triangulate the CSP's graph, process the triangles in the elimination ordering, then in the instantiation ordering, which is the reverse of the elimination ordering (see Section 2.3), and repeat the operation until quiescence. When the constraints are convex or when composition distributes over intersection, a single iteration suffices:

- The consistency of the CSP is guaranteed after traversing the triangles in the elimination order, and
- The constraints are guaranteed minimal after traversing the triangles in the instantiation ordering [Planken *et al.*, 2008; Long *et al.*, 2016].

For arbitrary binary constraints, the operation must be repeated until quiescence. Our new algorithm σ-ΔPPC (Algorithm 23) implements this idea for arbitrary constraints. In the case of convex or distributive constraints, the loop on line 18 can be eliminated, yielding P^3C [Planken *et al.*, 2008] and DPC+ [Long *et al.*, 2016].

As input to σ-ΔPPC (Algorithm 23), we provide $\mathcal{P}$, after triangulating the corresponding constraint graph and *peo*, a perfect elimination ordering of the graph. See illustration in Figure 4.3. We store the triangles of the graph in an array called



	tri[.]
1	(V_6,V_5,V_4)
2	(V_4,V_3,V_2)
3	(V_4,V_3,V_1)
4	(V_4,V_2,V_1)
5	(V_3,V_2,V_1)

	TrianglesEdge[.]
$C_{1,2}$	$(V_3,V_2,V_1),(V_4,V_2,V_1)$
$C_{1,3}$	$(V_3,V_2,V_1),(V_4,V_3,V_1)$
$C_{1,4}$	$(V_4,V_2,V_1),(V_4,V_3,V_1)$
$C_{2,3}$	$(V_4,V_3,V_2),(V_3,V_2,V_1)$
$C_{2,4}$	$(V_4,V_3,V_2),(V_4,V_2,V_1)$
$C_{3,4}$	$(V_4,V_3,V_1),(V_4,V_3,V_2)$
$C_{4,5}$	(V_6,V_5,V_4)
$C_{4,6}$	(V_6,V_5,V_4)
$C_{5,6}$	(V_6,V_5,V_4)

Figure 4.3: A triangulated graph, a *peo*, *tri*[$\cdot$], and *TrianglesEdge*[$\cdot$]

tri[$\cdot$]. The triangles are ordered along the *peo* such that a triangle on $k > j > i$ is stored as (V_k, V_j, V_i). The size of *tri*[$\cdot$] is $\mathcal{O}(n^3)$ where n is the number of variables in the CSP. Init-Triangle-Vector (Algorithm 22) initializes this array. A table *TrianglesEdge* stores, for each edge, the list of triangles in which the edge appears. The initialization of this table is too obvious to be reported. Such table has at most $\mathcal{O}(n^2)$ entries, each with $\mathcal{O}(n)$ items.

4.3.2 The σ-ΔPPC Algorithm

The σ-ΔPPC algorithm first identifies the articulation points in the triangulated graph. Then, it filters the domains of the corresponding variables given their incident constraints using Filter-Dom. It then filters the same constraints given the updated

Algorithm 22: INIT-TRIANGLE-VECTOR($\mathcal{P}, peo, tri[\cdot]$)

```
Input: P = (V, D, C),
       peo = (|V|, ..., 1): a perfect elimination ordering on V,
       tri[·]: an empty vector to store all the triangles in the graph
Output: counter: the number of triangles in the graph.
       As a side effect, it stores all triangles in tri
1 counter ← 1                                        /* An index for tri[·] */
2 for k ← |V| downto 3 by −1 do
3   for j ← k − 1 downto 2 do
4     for i ← j − 1 downto 1 do
5       if C_{i,j}, C_{j,k}, C_{i,k} ∈ C then
6         tri[counter] ← (V_k, V_j, V_j)
7         counter ← counter + 1
8 return counter − 1
```

domains using FILTER-RELS. The above is done in lines 1–3.

Starting from $k \leftarrow n$, σ-ΔPPC traverses the triangles along *peo*, then continues the same process down along the instantiation ordering then *peo* until reaching a fixpoint. Every time a triangle is considered, REVISE-TRIANGLE updates all three relations in the triangle while processing the domains of the variables at that articulation nodes. Every time an edge is updated, all the triangles in which at appears are flagged for revision using the bit vector *flag*$[\cdot]$.

In summary, σ-ΔPPC introduces the following improvements:

1. It exploits a perfect elimination ordering.

2. It processes the triangles in linear fashion, iterating up and down along the ordering.

3. It flags the triangles for revision only when one of the edges of the triangle was revised.

Algorithm 23: σ-ΔPPC($\mathcal{P}$, *peo*)

```
   Input: P = (V, D, C) with a triangulated constraint graph,
          peo = (|V|, ..., 1) a perfect elimination ordering on V
   Output: P if it is path consistent
 1 A_points ←ARTICULATION-POINTS(P)
 2 Q_AP ← A_points
 3 PROPAGATE-APs(A_points, Q_AP)
 4 tri[·]                          /* An empty vector to store O(n^3) triangles */
 5 size ← INIT-TRIANGLE-VECTOR(P, peo, tri[·])
 6 flag[·]       /* An empty vector to flag triangles that lost at least a tuple */
 7 change ← false
 8 Q_e ← ∅
 9 for i ← 1 to size do                  /* Sweeping along the elimination ordering */
10     (V_k, V_j, V_i) ← tri[i]
11     flag[i] ← false
12     Q_e ← Q_e ∪ REVISE-TRIANGLE(V_i, V_j, V_k, A_points)
13 if Q_e ≠ ∅ then change ← true
14 T ← ∅
15 foreach edge ∈ Q_e do T ← T ∪ TrianglesEdge(edge)
16 for i ← 1 to size do
17     if tri[i] ∈ T then flag[i] ← true
18 while change = true do
19     change ← false
20     for i ← size − 1 downto 1 by −1 do   /* Along the instantiation ordering */
21         (V_k, V_j, V_i) ← tri[i]
22         flag[i] ← false
23         Q_e ←REVISE-TRIANGLE(V_i, V_j, V_k, A_points)
24         if Q_e ≠ ∅ then change ← true
25         T ← ∅
26         foreach edge ∈ Q_e do T ← T ∪ TrianglesEdge(edge)
27         for i ← 1 to size do if tri[i] ∈ T then flag[i] ← true
28     if not change then break
29     change ← false
30     for i ← 2 to size by 1 do
31         (V_k, V_j, V_i) ← tri[i]
32         flag[i] ← false
33         Q_e ←REVISE-TRIANGLE(V_i, V_j, V_k, A_points)
34         if Q_e ≠ ∅ then change ← true
35         T ← ∅
36         foreach edge ∈ Q_e do T ← T ∪ TrianglesEdge(edge)
37         for i ← 1 to size do if tri[i] ∈ T then flag[i] ← true
38 return P
```

4. Finally, we introduce support structures for the tuples in a relation so that, when a triangle is revisited, only those tuples that have lost support are effectively

revised.

4.3.3 Support Structures in σ-ΔPPC

In addition to the above listed features, σ-ΔPPC uses support structures for the tuples in the relations.

For every a triangle in the graph, the idea is to record, for every tuple in a relation, its two supporting tuples in the relations in the triangle. Figure 4.4 illustrates this situation. Consider the three relations $R_{i,j}$, $R_{i,k}$, and $R_{j,k}$ in the triangle (V_i, V_j, V_k).

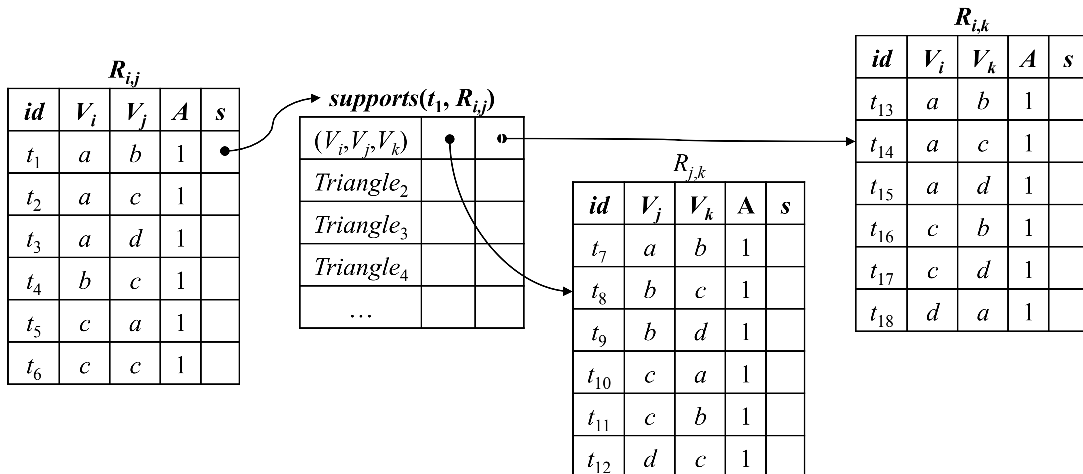


Figure 4.4: The support structure for tuples in σ-ΔPPC

We store each relation as an indexed array of tuples. Further, we add the Boolean flag 'A' (which stands for 'Active') for each tuple in the relation. This flag is initialized to 1 at the beginning and reset to 0 when the tuple is deleted. In addition to this flag, each tuple $t \in R_{i,j}$ has a pointer to its support structure *supports*$(t, R_{i,j})$. This support structure is an array indexed by all the triangles in which the relation $R_{i,j}$ appears such as the triangle (V_i, V_j, V_k). For the given triangle (V_i, V_j, V_k), *supports*$(t, R_{i,j})$ stores two pointers, each to the tuples that are consistent with t in (V_i, V_j, V_k).

Given a relation $R_{i,j}$ in the triangle (V_i, V_j, V_k), as soon as we find that tuple $(a, b) \in R_{i,j}$ is consistent with $(a, c) \in R_{i,k}$ and $(b, c) \in R_{j,k}$, we set the pointers of *supports*$((a, b), R_{i,j})$ to point to the tuples $(a, c) \in R_{i,k}$ and $(b, c) \in R_{j,k}$. At the same time, we update *supports*$((a, c), R_{i,k})$ and *supports*$((b, c), R_{j,k})$ even if they were already pointing to another existing support. In this manner, any future check to the consistency of any of these three tuples in this triangle can follow the pointers and verify whether or not the supporting tuples are active.

There are $\mathcal{O}(n)$ rows in each *supports* because each constraint appears in at most $(n - 2)$ triangles. Given that each relation has at most $\mathcal{O}(d^2)$ tuples and the triangulated graph has at most $\mathcal{O}(e + e')$ constraints, the number of *supports* structures is $\mathcal{O}((e + e')d^2)$ and the space needed to store them is $\mathcal{O}(n(e + e')d^2)$.

One can selectively choose whether or not to use the support structures. Generally speaking, while using them reduces the number of constraint checks, their size may become prohibitively large when the number of tuples in the problem's relations is exceedingly large. We denote σ-ΔPPC the version of our algorithm that does not use the support structures and σ-ΔPPC^{sup} the one that does.

The idea of support was first introduced in AC-2001 and PC-2001 [Bessière *et al.*, 2005]. Our support structures are inspired from those used by Karakashian *et al.* [2010] and differ from those used in PC-2001. For each tuple (a, b) and each third variable k, PC-2001 stores $Last((V_i, a), (V_j, b), V_k) \leftarrow c$ and cannot store the three tuples in the triangle as supporting each other. As a result, verifying whether or not the supports of (a, b) are still active requires checking the constraints and adds to the number of constraint checks, which we avoid.

PC-2001 does have an advantage over our support structures in that PC-2001 orders the domains. As a result, when the support $Last((V_i, a), (V_j, b), V_k) \leftarrow c$ is no longer valid, PC-2001 continues to the next value, after c, in D_k because it has

already checked all values before c. Our support structures do not order the domains: when c is no longer a valid support, our algorithm must check all values x where $x \in D_k \setminus \{c\}$.

4.3.4 Reflections on PPC

While the PPC property (Definition 2.5.5) was proposed as a new approximation of PC (Definition 2.5.2), it becomes apparent that any algorithm for enforcing PPC *must* operate along a perfect elimination ordering, exactly as advocated by the DPC algorithm [Dechter and Pearl, 1988], so that the worst-case complexity can be bound by the induced width of the ordering. This realization was first expressed by Nic Wilson in a private communication and integrated by Yorke-Smith [2005] in an algorithm for reasoning about time and later in P^3C [Planken *et al.*, 2008].

σ-ΔPPC is the first algorithm that generalizes this mechanism to arbitrary constraints. Unlike P^3C and DPC+ [Long *et al.*, 2016], which operate on restricted classes of constraints (i.e., STP constraints and distributive subalgebra), and unlike the original PPC, σ-ΔPPC updates all the edges of the triangles at the same time, handles the articulation points in the constraint graph, and uses support structures to reduce constraint checks.

4.4 PC-8$^+$ and PC-2001$^+$

We propose two separate improvements that we apply to both PC-8 (Algorithm 8, Section 2.8.3) and PC-2001 (Algorithm 9, Section 2.8.4). Below, we discuss them only in the context of PC-8. Their adaptation to PC-2001 is straightforward and not reported in this section.

Our two improvements yield PC-8-FLAG (Algorithm 24) and PC-8-ORDERING (Algorithm 26), which we then combine to form the new algorithm PC-8$^+$ (Algorithm 27).

4.4.1 PC-8-FLAG

Consider the following lines of PC-8 (Algorithm 8):

1 $\mathcal{Q} \leftarrow \emptyset$
2 **foreach** $V_i, V_j, V_k \in \mathcal{V}$ such that $(i < j) \wedge (i \neq k) \wedge (j \neq k)$ **do**
3   **foreach** $(a, b) \in R_{i,j}$ **do**
4   **if** $\nexists c \in D_k$ such that $(a, c) \in R_{i,k} \wedge (b, c) \in R_{j,k}$ **then**
5             $R_{i,j} \leftarrow R_{i,j} \setminus \{(a, b)\}$
6             $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{((V_i, a), V_j), ((V_j, b), V_i)\}$

Let's apply them to the example shown in Figure 4.5. If, for $(a, b) \in R_{i,j}$, we find

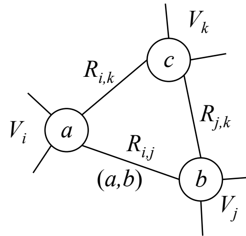


Figure 4.5: Three variables of a CSP

a value $c \in D_k$ such that $(a, c) \in R_{i,k}$ and $(b, c) \in R_{i,k}$, we say that (a, c) and (b, c) have 'served' in a consistency check. If such a value, c, is not found in D_k, then, PC-8 removes (a, b) from $R_{i,j}$, and, in line 6, it adds $((V_i, a), V_j)$ and $((V_j, b), V_i)$ to the propagation queue, $\mathcal{Q}$. Those items are added to $\mathcal{Q}$ because the tuple (a, b) could have served in the consistency check of other tuples in other triangles, whose consistency is now 'threatened' and should, thus, be re-examined. Adding elements

to the queue necessarily induces the execution of more operations by PC-8. When (a, b) never 'served' in a consistency checks, those operations are wasted.

To address this issue, we introduce the bit-array *flag* that stores, for every tuple, such as (a, b), whether or not it 'served' in a successful consistency check. All the entries in *flag* are initially set to *false*. When, in the example above, $c \in D_k$ is found consistent with (a, b), the *flag* values of $(a, c) \in R_{i,k}$ and $(b, c) \in R_{i,k}$ are set to *true*. Further, $((V_i, a), V_j)$ and $((V_j, b), V_i)$ are added to the queue only when the *flag* value of (a, b) is *true*.

Modifying PC-8 to integrate the above mechanism yields the two algorithms PC-8-FLAG (Algorithm 24) and REVISE-PC-8-FLAG (Algorithm 25). The modification is bound to reduce the number of constraint checks.

Algorithm 24: PC-8-FLAG($\mathcal{P}$)

Input: $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ with a complete constraint graph
Output: Path consistent $\mathcal{P}$

```
1  Q ← ∅
2  foreach V_i, V_j ∈ V such that (i < j) do
3      foreach (a, b) ∈ R_{i,j} do
4          flag((a, b), R_{i,j}) ← false
5  foreach V_i, V_j, V_k ∈ V such that (i < j) ∧ (i ≠ k) ∧ (j ≠ k) do
6      Q ← Q ∪ REVISE-PC-8-FLAG(V_i, V_j, V_k, flag)
7  while Q do
8      ((V_i, a), V_k) ← POP(Q)
9      foreach V_j ∈ V such that (i ≠ j) ∧ (k ≠ j) do
10         Q ← Q ∪ REVISE-PC-8-FLAG(V_i, V_j, V_k, flag)
11 return P
```

Algorithm 25: Revise-PC-8-Flag($V_i, V_j, V_k, flag$)

```
   Input: V_i, V_j, V_k ∈ 𝒱, flag: flag structure
   Output: 𝒬: A set of elements to add to the queue
1  𝒬 ← ∅
2  foreach (a, b) ∈ R_{i,j} do
3      if ∄ c ∈ D_k such that (a, c) ∈ R_{i,k} ∧ (b, c) ∈ R_{j,k} then
4          R_{i,j} ← R_{i,j} \ {(a, b)}
5          if flag((a, b), R_{i,j}) then 𝒬 ← 𝒬 ∪ {((V_i, a), V_j), ((V_j, b), V_i)}
6      else
7          flag((a, c), R_{i,k})) ← true
8          flag((b, c), R_{j,k})) ← true
9  return 𝒬
```

4.4.2 PC-8-Ordering

Let's consider again the first few lines of PC-8 (Algorithm 8) shown below:

```
1  𝒬 ← ∅
2  foreach V_i, V_j, V_k ∈ 𝒱 such that (i < j) ∧ (i ≠ k) ∧ (j ≠ k) do
3      foreach (a, b) ∈ R_{i,j} do
4          if ∄ c ∈ D_k such that (a, c) ∈ R_{i,k} ∧ (b, c) ∈ R_{j,k} then
5              R_{i,j} ← R_{i,j} \ {(a, b)}
6              𝒬 ← 𝒬 ∪ {((V_i, a), V_j), ((V_j, b), V_i)}
```

We modify line 2 of the above pseudocode to clarify the exact sequence of operations (while renaming the queue for the sake of clarity):

```
1  𝒬_1 ← ∅
2  for i ← 1 to n by 1 do
3      for j ← i + 1 to n by 1 do
4          foreach V_k ∈ 𝒱 such that (k ≠ i) ∧ (k ≠ j) do
5              foreach (a, b) ∈ R_{i,j} do
6                  if ∄ c ∈ D_k such that (a, c) ∈ R_{i,k} ∧ (b, c) ∈ R_{j,k} then
7                      R_{i,j} ← R_{i,j} \ {(a, b)}
8                      𝒬_1 ← 𝒬_1 ∪ {((V_i, a), V_j), ((V_j, b), V_i)}
```

In this reformulated pseudocode, it becomes clear that the tuples of $R_{i,j}$ are always examined before a relation $R_{x,y}$ with $(x > i)$ or $(x = i) \wedge (y > j)$ is considered. Consequently, if any $(a, b) \in R_{i,j}$ is removed in line 7 of the reformulated pseudocode, this tuple is necessarily removed before $R_{x,y}$ is even considered, and thus, could not have possibly 'served' to establish the consistency of any tuple in $R_{x,y}$.

Now, let us re-examine how PC-8 processes the elements in its queue, for the first time, right after the loop reported above:

```
7  while Q do
8      ((V_i, a), V_k) ← Pop(Q)
9      foreach V_j ∈ 𝒱 such that (j ≠ i) ∧ (j ≠ k) do
10         foreach (a, b) ∈ R_{i,j} do
11             if ∄ c ∈ D_k such that (a, c) ∈ R_{i,k} ∧ (b, c) ∈ R_{j,k} then
12                 R_{i,j} ← R_{i,j} \ {(a, b)}
13                 Q ← Q ∪ {((V_i, a), V_j), ((V_j, b), V_i)}
```

When $((V_i, a), V_k)$ is popped from the queue, it becomes clear that it is useless to consider, in line 9, any variable V_j where $j < k$. Such an effort is wasted. Such a consideration is safe the *first* time the queue is processed at line 7. In any subsequent iteration through the queue, $\forall j \; V_j$ should be considered in line 9.

This observation is the basis of our algorithm PC-8-Ordering (Algorithm 26), which saves some consistency checks when processing the queue right after the first loop (lines 9–16). As the queue is emptied, then PC-8-Ordering operates like PC-8 (lines 17–23).

Algorithm 26: PC-8-Ordering($\mathcal{P}$)

```
   Input: P = (V, D, C) with a complete constraint graph
   Output: Path consistent P
1  Q1 ← ∅
2  for i ← 1 to n by 1 do
3      for j ← i + 1 to n by 1 do
4          foreach V_k ∈ V such that (k ≠ i) ∧ (k ≠ j) do
5              foreach (a, b) ∈ R_{i,j} do
6                  if ∄ c ∈ D_k such that (a, c) ∈ R_{i,k} ∧ (b, c) ∈ R_{j,k} then
7                      R_{i,j} ← R_{i,j} \ {(a, b)}
8                      Q1 ← Q1 ∪ {((V_i, a), V_j), ((V_j, b), V_i)}
9  Q2 ← ∅
10 while Q1 do
11     ((V_i, a), V_k) ← Pop(Q1)
12     foreach V_j ∈ V such that (i ≠ j) ∧ (j < k) do
13         foreach (a, b) ∈ R_{i,j} do
14             if ∄ c ∈ D_k such that (a, c) ∈ R_{i,k} ∧ (b, c) ∈ R_{j,k} then
15                 R_{i,j} ← R_{i,j} \ {(a, b)}
16                 Q2 ← Q2 ∪ {((V_i, a), V_j), ((V_j, b), V_i)}
17 while Q2 do
18     ((V_i, a), V_k) ← Pop(Q2)
19     foreach V_j ∈ V such that (i ≠ j) ∧ (k ≠ j) do
20         foreach (a, b) ∈ R_{i,j} do
21             if ∄ c ∈ D_k such that (a, c) ∈ R_{i,k} ∧ (b, c) ∈ R_{j,k} then
22                 R_{i,j} ← R_{i,j} \ {(a, b)}
23                 Q2 ← Q2 ∪ {((V_i, a), V_j), ((V_j, b), V_i)}
24 return P
```

4.4.3 PC-8$^+$

The PC-8$^+$ algorithm (Algorithm 27) combines the approaches of PC-8-Flag and PC-8-Ordering. Although the two improvements are not independent, the number of constraint checks in PC-8$^+$ is guaranteed to never exceed either of their number.

Algorithm 27: PC-8$^+$($\mathcal{P}$)

```
    Input: P = (V, D, C) with a complete constraint graph
    Output: Path consistent P
 1  foreach V_i, V_j ∈ V such that (i < j) do
 2  └ foreach (a, b) ∈ R_{i,j} do flag((a, b), R_{i,j}) ← false
 3  Q_1 ← ∅
 4  for i ← 1 to n by 1 do
 5  │ for j ← i + 1 to n by 1 do
 6  │ │ foreach V_k ∈ V such that (k ≠ i) ∧ (k ≠ j) do
 7  └ └ └ Q_1 ← Q_1 ∪ Revise-PC-8-Flag(V_i, V_j, V_k, flag)
 8  Q_2 ← ∅
 9  while Q_1 do
10  │ ((V_i, a), V_k) ← Pop(Q_1)
11  │ foreach V_j ∈ V such that (i ≠ j) ∧ (j < k) do
12  └ └ Q_2 ← Q_2 ∪ Revise-PC-8-Flag(V_i, V_j, V_k, flag)
13  while Q_2 do
14  │ ((V_i, a), V_k) ← Pop(Q_2)
15  │ foreach V_j ∈ V such that (i ≠ j) ∧ (k ≠ j) do
16  └ └ Q_2 ← Q_2 ∪ Revise-PC-8-Flag(V_i, V_j, V_k, flag)
17  return P
```

4.5 Time and Space Complexities

The asymptotic complexity of PPC+AP, ΔPPC, and PC-8$^+$ are that of the original algorithms upon which they improve. Thus,

- PPC+AP and ΔPPC are $\mathcal{O}(\delta(e+e')d^5)$ and $\mathcal{O}(e'd^2)$ in space. The data structures used are $\mathcal{O}(n^3)$, necessary to store the triangles of the constraint network.
- PC-8$^+$ is $\mathcal{O}(n^3d^4)$ in time and $\mathcal{O}(n^2d^2)$ in space. The space for the new data structures is $\mathcal{O}(n^2d^2)$.

Because it follows a perfect elimination ordering *peo* of the triangulated graph, the complexity of σ-ΔPPC can be bounded by the width of the ordering w_{peo} after trian-

gulation. Each revision along the ordering is $\mathcal{O}(w_{peo}^2 \cdot n \cdot d^3)$ and there could at most $\mathcal{O}((e+e')d^2)$ revisions. Thus, σ-ΔPPC is $\mathcal{O}(w_{peo}^2 \cdot n \cdot (e+e')d^5)$ in time and $\mathcal{O}(e'd^2)$ in space. Its data structures are dominated by the space of storing the triangles of the triangulated graph, which is $\mathcal{O}(n^3)$.

The above information is summarized in Table 4.3.

Table 4.3: Properties of the introduced algorithms

	Constraint graph	**Enforces consistency property**	**Complexity**		
			Time	**Space**	
				Constraints	**Data struc.**
PPC+AP	Triangulated	PPC	$\mathcal{O}(\delta(e+e')d^5)$	$\mathcal{O}(e'd^2)$	$\mathcal{O}(n^3)^*$
ΔPPC	Triangulated	PPC	$\mathcal{O}(\delta(e+e')d^5)$	$\mathcal{O}(e'd^2)$	$\mathcal{O}(n^3)^*$
PC-8$^+$	Completed	PC	$\mathcal{O}(n^3d^4)$	$\mathcal{O}(n^2d^2)$	$\mathcal{O}(n^2d)$
σ-ΔPPC	Triangulated	PPC	$\mathcal{O}(w_{peo}^2 \cdot n \cdot d^3)$	$\mathcal{O}(e'd^2)$	$\mathcal{O}(n^3)^*$

d: Maximum domain size, MAX($|D_i|$)
n: Number of variables, $|\mathcal{V}|$
e: Number of constraints before graph is altered, $|\mathcal{C}|$
e': Number of edges added by a triangulation of the graph
*: $\mathcal{O}(n^3+e+e')$
w_{peo}: Width of the triangulated graph in the *peo* ordering
δ: The maximum degree of the graph

Summary

In this chapter, we introduced PPC+AP, an algorithm that corrects the original BSH-PPC, whose published pseudocode ignores the articulation points of the constraint graph although the original code and the original experiments are correct. Then, we modified PPC+AP to operate on the triangles of the graph instead of operating on its edges, yielding the ΔPPC algorithm. We then refined ΔPPC into σ-ΔPPC, which processes the triangles along a perfect elimination ordering, flags triangles to skip processing unaffected triangles, and uses support structures to reduce unnecessary

revisions of the tuples in a relation. Finally, we introduced two improvements to PC-8, namely, PC-8-Flag and PC-8-Ordering, which we combined into PC-8^+.

Table 4.5 reports the algorithms we have discussed in Chapters 2 and 4. These algorithms are empirically evaluated in the next chapter, Chapter 5.

Algorithm	**Property**	**Original publication**
DPC	sDPC	[Dechter and Pearl, 1988]
sCDC1	sCDC	[Lecoutre *et al.*, 2007a]
PC-2	PC	[Mackworth and Freuder, 1984]
PC-8		[Chmeiss and Jégou, 1998]
PC-8-Ordering		*proposed*
PC-8-Flag		*proposed*
PC-8^+		*proposed*
PC-2001		[Bessière *et al.*, 2005]
PC-2001-Ordering		*proposed*
PC-2001-Flag		*proposed*
PC-2001^+		*proposed*
PPC+AP *	PPC	[Bliek and Sam-Haroud, 1999]
ΔPPC		*proposed*
σ-ΔPPC		*proposed*
σ-$\Delta\text{PPC}^{sup2001}$		*proposed*
σ-ΔPPC^{sup}		*proposed*
sDC2	sDC	[Lecoutre *et al.*, 2007b]

** Corrected version of original.*

Chapter 5

Empirical Evaluations

In this chapter, we discuss our empirical evaluations of the discussed and introduced algorithms, first on well-known benchmark problems found at the XCSP website[1], then on four randomly generated data-sets covering the phase-transition phenomenon [Cheeseman *et al.*, 1991]. Finally, we discuss our results.

5.1 Benchmark Problems

In this section, we first introduce the benchmark problems tested, then compare the performance of the known and new algorithms for path consistency and its approximations on benchmark binary problems.

5.1.1 Problem Characteristics

The binary CSP benchmarks consist of 2,288 instances in 79 benchmarks. We eliminate the following five graph-coloring instances, because their table constraints are too large to generate: 3-insertions-3-3.xml (k-insertion), 4-fullins-5-7_ext.xml (full-

[1] `http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html`

insertion), inithx-i-1-10_ext.xml (register-inithx), and mulsol-i-3-20.xml and mulsol-i-4-31.xml (register-mulsol), Tables 5.1, 5.2, and 5.3 summarize the characteristics of the problems tested and the number of edges added to complete the graph and to triangulate it.

Table 5.1: Characteristics of the benchmark problems (table 1 of 3)

		# instances							**# edges added for**	
	Benchmark	**Total**	**Consistent**	**Inconsistent**	**# variables**	**# edges**	**density**	**# art. points**	**Completion**	**Triangulation**
	BH-4-13	7	7	0	208.0	4,217.0	19.6%	0.0	17,104.0	296.0
	BH-4-4	10	10	0	64.0	431.0	21.4%	0.0	1,522.0	62.0
	BH-4-7	20	20	0	112.0	1,261.0	20.3%	0.0	4,844.0	133.0
	QCP-10	15	14	1	100.0	822.0	16.6%	0.0	4,128.0	1,729.7
	QCP-15	15	15	0	225.0	2,519.3	10.0%	0.0	22,680.7	6,192.9
	QCP-20	15	15	0	400.0	5,540.0	6.9%	0.0	74,260.0	15,687.2
	QCP-25	15	15	0	625.0	10,142.0	5.2%	0.0	184,858.0	32,023.3
	QWH-10	10	10	0	100.0	756.0	15.3%	0.0	4,194.0	1,282.1
	QWH-15	10	10	0	225.0	2,324.0	9.2%	0.0	22,876.0	4,863.7
	QWH-20	10	10	0	400.0	5,092.0	6.4%	0.0	74,708.0	12,398.9
	QWH-25	10	10	0	625.0	9,300.0	4.8%	0.0	185,700.0	25,411.6
	bqwh-15-106	100	100	0	106.0	593.4	10.7%	0.0	4,971.6	1,912.0
	bqwh-18-141	100	100	0	141.0	878.3	8.9%	0.1	8,991.8	3,554.7
	coloring	22	20	2	119.7	1,203.0	18.8%	1.0	18,309.7	10,154.1
composed	25-1-2	10	0	10	33.0	224.0	42.4%	0.2	304.0	63.2
composed	25-1-25	10	0	10	33.0	247.0	46.8%	0.0	281.0	111.1
composed	25-1-40	10	0	10	33.0	262.0	49.6%	0.0	266.0	126.2
composed	25-1-80	10	0	10	33.0	302.0	57.2%	0.0	226.0	134.4
composed	25-10-20	10	10	0	105.0	620.0	11.4%	0.0	4,840.0	441.0
composed	75-1-2	10	0	10	83.0	624.0	18.3%	0.2	2,779.0	1,102.8
composed	75-1-25	10	0	10	83.0	647.0	19.0%	0.0	2,756.0	1,213.5
composed	75-1-40	10	0	10	83.0	662.0	19.5%	0.0	2,741.0	1,293.8
composed	75-1-80	10	0	10	83.0	702.0	20.6%	0.0	2,701.0	1,384.8
	driver	7	7	0	351.6	7,201.4	9.9%	5.3	67,498.1	7,648.7
	frb30-15	10	10	0	30.0	212.0	48.7%	0.0	223.0	122.4
	frb35-17	10	10	0	35.0	264.6	44.5%	0.0	330.4	174.0
	frb40-19	10	10	0	40.0	320.4	41.1%	0.0	459.6	253.0
	frb45-21	10	10	0	45.0	379.2	38.3%	0.0	610.8	349.2
	frb50-23	10	10	0	50.0	438.8	35.8%	0.0	786.2	453.2
	frb53-24	10	10	0	53.0	473.6	34.4%	0.0	904.4	523.2
	frb56-25	10	10	0	56.0	515.2	33.5%	0.0	1,024.8	588.0
	frb59-26	10	10	0	59.0	550.0	32.1%	0.0	1,161.0	680.8
	geom	100	100	0	50.0	421.4	34.4%	0.0	803.6	108.9

Summary continues in next table.

Table 5.2: Characteristics of the benchmark problems (table 2 of 3)

		# instances							# edges added for	
	Benchmark	**Total**	**Consistent**	**Inconsistent**	**# variables**	**# edges**	**density**	**# art. points**	**Completion**	**Triangulation**
graphColoring	hos	14	14	0	1,248.3	24,011.3	2.7%	0.4	836,180.8	39,741.2
graphColoring	full-insertion	40	40	0	856.8	12,809.2	6.5%	0.0	350,562.2	122,695.6
graphColoring	k-insertion	32	32	0	445.0	2,963.7	4.3%	0.0	183,376.6	22,600.0
graphColoring	leighton-15	26	26	0	450.0	12,113.0	12.0%	0.5	88,912.0	54,070.1
graphColoring	leighton-25	31	31	0	450.0	12,676.9	12.5%	0.0	88,348.1	47,266.9
graphColoring	leighton-5	8	8	0	450.0	7,752.0	7.7%	0.0	93,273.0	55,552.8
graphColoring	mug	8	8	0	94.0	156.0	3.6%	0.0	4,233.0	60.0
graphColoring	myciel	16	16	0	96.5	981.9	20.4%	0.0	6,045.9	1,181.9
graphColoring	register-fpsol	37	37	0	464.6	10,051.6	9.3%	0.0	42,025.2	228.3
graphColoring	register-inithx	31	31	0	720.6	15,958.8	6.2%	0.0	135,209.5	284.9
graphColoring	register-mulsol	47	47	0	188.9	3,928.6	22.2%	0.0	9,637.4	231.9
graphColoring	register-zeroin	31	31	0	209.5	3,775.1	17.3%	0.0	6,637.9	286.5
graphColoring	school	8	8	0	368.5	16,853.5	24.7%	1.0	48,086.5	21,705.5
graphColoring	sgb-book	26	26	0	202.3	655.3	7.0%	21.7	36,732.2	206.1
graphColoring	sgb-games	4	4	0	120.0	638.0	8.9%	0.0	6,502.0	1,472.0
graphColoring	sgb-miles	42	42	0	128.0	3,051.6	37.5%	0.5	4,803.4	421.4
graphColoring	sgb-queen	50	50	0	140.6	2,745.0	28.1%	0.0	9,344.3	7,533.1
	hanoi	5	5	0	47.6	46.6	11.1%	45.6	2,014.8	0.0
	langford	4	4	0	25.3	358.8	100.0%	0.0	0.0	0.0
	lard	10	10	0	87.5	3,788.5	100.0%	0.0	0.0	0.0
	marc	10	5	5	88.0	3,844.0	100.0%	0.0	0.0	0.0
	os-taillard-4	30	21	9	16.0	48.0	40.0%	0.0	72.0	38.0
	os-taillard-5	30	30	0	25.0	100.0	33.3%	0.0	200.0	110.0
rand	23-Feb	10	10	0	23.0	253.0	100.0%	0.0	0.0	0.0
rand	24-Feb	10	10	0	24.0	276.0	100.0%	0.0	0.0	0.0
rand	25-Feb	10	10	0	25.0	300.0	100.0%	0.0	0.0	0.0
rand	26-Feb	10	10	0	26.0	325.0	100.0%	0.0	0.0	0.0
rand	27-Feb	10	10	0	27.0	351.0	100.0%	0.0	0.0	0.0
rand	2-30-15	50	50	0	30.0	222.2	51.1%	0.0	212.8	116.2
rand	2-30-15-fcd	50	50	0	30.0	222.2	51.1%	0.0	212.8	116.2
rand	2-40-19	50	50	0	40.0	338.3	43.4%	0.0	441.7	253.2
rand	2-40-19-fcd	50	50	0	40.0	338.3	43.4%	0.0	441.7	253.2
rand	2-50-23	50	50	0	50.0	467.2	38.1%	0.0	757.8	451.8
rand	2-50-23-fcd	50	50	0	50.0	467.2	38.1%	0.0	757.8	451.8
rlfap	Graphs	14	8	6	547.7	3,059.0	2.7%	0.9	12,787.4	19,348.1
rlfap	GraphsMod	12	6	6	751.3	2,933.1	1.2%	18.9	179,180.1	20,013.5
rlfap	Scens11	12	12	0	680.0	4,103.0	1.8%	4.0	226,757.0	1,757.0
rlfap	Scens	11	6	5	559.3	3,486.2	3.0%	4.5	44,442.4	1,458.9
rlfap	ScensMod	13	9	4	328.0	1,173.4	2.4%	11.2	58,752.8	810.4

Summary continues in next table.

Table 5.3: Characteristics of the benchmark problems (table 3 of 3)

	# instances							# edges added for	
Benchmark	**Total**	**Consistent**	**Inconsistent**	**# variables**	**# edges**	**density**	**# art. points**	**Completion**	**Triangulation**
tightness0.1	100	100	0	40.0	752.2	96.4%	0.0	27.8	21.1
tightness0.2	100	100	0	40.0	414.0	53.1%	0.0	366.0	233.7
tightness0.35	100	100	0	40.0	250.0	32.1%	0.0	530.0	247.6
tightness0.5	100	100	0	40.0	180.0	23.1%	0.0	600.0	214.0
tightness0.65	100	100	0	40.0	135.0	17.3%	0.2	645.0	169.1
tightness0.8	100	100	0	40.0	103.0	13.2%	1.0	676.2	120.4
tightness0.9	100	86	14	40.0	84.0	10.8%	2.4	691.4	87.8

5.1.2 Experimental Set-Up

We run our experiments on the Crane supercomputer with a timeout limit of five hours per instance and 60 GiB of memory. We record the instruction count using the 'perf' tool and convert the number of instructions to a time value by assuming a 3.0 GHz CPU.

Comparing the original BSH-PPC algorithm and our correction of it, PPC+AP, the 'bug' is revealed on the five instances of the hanoi benchmark, whose constraint graph is a chain. PPC+AP correctly enforces the PPC property and removes on average 191,558.6 tuples while BSH-PPC removes none.

Below, we report the following experiments:

1. In Section 5.1.3, we compare the effect of enforcing arc consistency (AC) before running any other algorithm and show that, in general, running AC first reduces the CPU time and increases the number of problems completed within the allocated time threshold.

2. In Section 5.1.4, we assess the benefits of using a propagation queue of triangles instead of a queue of edges.

3. In Section 5.1.5, we demonstrate the benefits of following a PEO while enforcing consistency on the triangles.

4. In Section 5.1.6, we compare the effectiveness of the type of support structures used in PC-2001 to that of our new support structures in the context of our σ-ΔPPC algorithm.

5. In Section 5.1.7, we assess the effectiveness of our support structures in our σ-ΔPPC algorithm.

6. In Section 5.1.8, we assess the effectiveness of the two improvements we propose for PC-8. And, in Section 5.1.9, we do the same evaluation for PC-2001.

7. In Section 5.1.10, we compare, among each other, the best algorithms we have in each category.

In each experiment, we report, as appropriate:

1. The number of instances completed by the algorithms being compared.

2. The CPU time computed as an average over the instances completed by the algorithms.

3. The number of deleted tuples in the consistent instances completed by the compared algorithms. When no instance is completed by all algorithms being compared, then the CPU time is not shown but the mark '-' appears in the table.

4. The summation of the above values over the tested benchmarks.

5.1.3 Pre-processing with Arc Consistency

In this section, we evaluate the effect of running arc consistency before any of our algorithms, except sCDC1, because this algorithm enforces AC as a first step. In order to enforce arc consistency as a pre-processing step to a given algorithm:

1. We first run AC-2001 (Section 2.7) on the problem instance at hand.
2. We generate universal constraints for the added edges necessary to complete the constraint graph or to triangulate it, as appropriate.
3. Then, we synchronize the constraint relations with the domains by applying Filter-Rels (Algorithm 18).

Finally, we execute the considered path-consistency algorithm. For all PPC-based algorithms, we skip the initial processing of articulation points (i.e., in PPC+AP (Algorithm 16), line 4 is omitted) because the operation is already secured by the pre-processing with AC.

In Section B.1 of Appendix B, we report the detailed results of pre-processing by AC for the following algorithms: DPC , PC-2, PC-8, PC-8-Ordering, PC-2001, PC-2001-Ordering, PPC+AP, ΔPPC, σ-ΔPPC, σ-$\Delta\text{PPC}^{sup2001}$, σ-ΔPPC^{sup}, and sDC2. In Table 5.4, we summarize the results reported in the appendix and include a reference to the corresponding table of the appendix. Generally speaking, pre-processing with AC:

1. consistently saves CPU time,
2. consistently increases the number of instances completed, and
3. does not affect the number of tuples removed (except for DPC, where this number is increased).

Table 5.4: Summary of effect of pre-processing with AC

	# instances			CPU time (s)			
	w/o AC	**with AC**	**Gain**	**w/o AC**	**with AC**	**Saving**	**Reference**
DPC	2,288	2,288	0	1,989.3	1,829.1	160.2	Table B.2
PC-2	2,228	2,250	22	24,174.5	18,662.9	5,511.6	Table B.3
PC-8	2,243	2,264	21	23,570.8	19,009.7	4,561.1	Table B.4
PC-8-Ordering	2,242	2,266	24	21,229.3	18,385.9	2,913.4	Table B.5
PC-2001	2,091	2,128	37	6,930.8	5,786.2	1,144.6	Table B.6
PC-2001-Ordering	2,091	2,128	37	6,493.0	5,695.3	797.7	Table B.7
PPC+AP	2,288	2,288	0	7,396.8	6,747.1	649.6	Table B.8
ΔPPC	2,285	2,285	0	4,229.7	3,576.1	653.7	Table B.9
σ-ΔPPC	2,288	2,288	0	4,929.6	3,651.4	1,278.2	Table B.10
σ-ΔPPC^{sup2001}	2,184	2,193	9	2,782.8	2,202.0	580.9	Table B.11
σ-ΔPPC^{sup}	2,184	2,193	9	2,787.2	2,228.9	558.3	Table B.12
sDC2	2,247	2,255	8	11,567.6	11,791.2	223.6	Table B.13

Indeed, in the case of DPC, pre-processing with AC increases the number of tuples deleted and decreases total processing time. Both algorithms, DPC and AC+DPC, complete all 2,288 instances. Table 5.5 shows selected results. The complete results are shown in Table B.2 in Appendix B.

Table 5.5: Enforcing AC before DPC

		# tuples deleted			CPU time (s)		
		DPC	**AC+DPC**	**Gain**	**DPC**	**AC+DPC**	**Saving**
QCP-25	15	1,449,297.2	1,620,407.6	171,110.4	34.9	7.6	27.3
QWH-25	10	1,145,354.8	1,306,672.0	161,317.2	26.2	5.2	21.0
composed-75-1-2	10	319.0	*Inconsistent*	-	0.2	0.2	0.0
composed-75-1-25	10	1,605.0	*Inconsistent*	-	0.2	0.2	0.0
hanoi	5	0.0	191,558.6	191,558.6	0.2	2.0	-1.9

Most notably, AC+DPC detects the inconsistency of all the instances of the composed-75-1-2 and composed-75-1-25 benchmarks, while DPC does not. Further, on the hanoi benchmark, AC+DPC uses slightly more CPU time, although it filters

many tuples while DPC filters none.

Given that pre-processing with AC provides such a robust improvement, in the rest of this chapter, *we always include this pre-processing step.*

5.1.4 Propagation Queue: Edges versus Triangles

In this section, we assess the impact of replacing the queue of edges of PPC+AP (Algorithm 16), our corrected version of the original PPC algorithm, with a queue of triangles in ΔPPC (Section 4.2).

In Table 5.6 we report selected results from Tables B.14 and B.15. In general, using triangles improves performance. However, ΔPPC does not complete three instances of the graphColoring full-insertion benchmark while PPC+AP completes all instances. Each of these instances has 4,146 variables, ΔPPC exceeds the memory limit because the queue of triangles is $\mathcal{O}(n^3)$, where n is the number of variables, whereas the queue of edges is $\mathcal{O}(n^2)$.

Table 5.6: A propagation queue made of edges (AC+PPC+AP) vs triangles (AC+ΔPPC)

			# instances			**CPU time (s)**		
			AC+PPC+AP	**AC+ΔPPC**	**Gain**	**AC+PPC+AP**	**AC+ΔPPC**	**Saving**
graphColoring	full-insertion	40	40	37	-3	170.2	74.4	95.8
	leighton-15	26	26	26	0	476.2	176.9	299.4
	leighton-25	31	31	31	0	1,084.6	385.1	699.4
	register-fpsol	37	37	37	0	175.3	80.4	94.9
	register-inithx	31	31	31	0	175.3	84.4	90.9
	school	8	8	8	0	280.9	105.2	175.7

5.1.5 Propagating Along a PEO

In this section, we evaluate the impact of propagating consistency along a PEO by comparing the performance of ΔPPC to that of σ-ΔPPC. The complete results, appearing in Tables B.16 and B.17, show a consistent but not sizable improvement in CPU time. Table 5.7 shows selected results.

While ΔPPC fails to run on three instances of graphColoring full-insertion because of memory limitations, σ-ΔPPC does not suffer from this limitation. Indeed, the latter does not use any propagation queue.

Table 5.7: Propagating along a PEO: AC+ΔPPC vs AC+σ-ΔPPC

		# Instances			CPU time (s)		
		AC+ΔPPC	**AC+σ-ΔPPC**	**Gain**	**AC+ΔPPC**	**AC+σ-ΔPPC**	**Saving**
coloring	22	22	22	0	5.0	2.6	2.4
graphColoring-full-insertion	40	37	40	3	74.4	55.8	18.6
graphColoring-k-insertion	32	32	32	0	13.4	8.5	4.9
graphColoring-leighton-5	8	8	8	0	37.4	23.5	13.9
rlfap-Graphs	14	14	14	0	583.2	589.3	-6.1
rlfap-GraphsMod	12	12	12	0	106.0	103.3	2.7

5.1.6 Comparing Types of Supports

We enhance our best PPC enforcing algorithm, σ-ΔPPC, with two types of support structures: supports à la PC-2001 in σ-$\Delta\text{PPC}^{sup2001}$ and our new support structures in σ-ΔPPC^{sup}, see discussion in Section 4.3.3. The detailed results of the evaluation on benchmark problems appear in Tables B.18 and B.19 of Appendix B.[2]

[2] Table B.18 reports the results of benchmarks where no filtering occurs and Table B.19 those where filtering occurs.

Generally speaking, σ-ΔPPC^{sup} consistently uses slightly more CPU time (about 8%) than σ-ΔPPC^{sup2001} on instances where no filtering occurs. On problems where tuples are removed, σ-ΔPPC^{sup} uses about 1% less CPU time than σ-ΔPPC^{sup2001}.

In Table 5.8, we show representative results from Tables B.18 and B.19 with the number of tuples deleted shown for reference. On the two os-taillard benchmarks,

Table 5.8: Comparing the two types of support structures

			CPU time (s)		
		# deleted tuples	**AC+σ-ΔPPC^{sup2001}**	**AC+σ-ΔPPC^{sup}**	**Saving**
graphColoring-school	8	0.0	115.9	128.4	-12.5
os-taillard-4	30	550,541.0	127.7	129.8	-2.2
os-taillard-5	30	559,583.8	1,056.1	1,059.1	-3.1
rlfap-Graphs	14	743,935.3	90.2	73.3	16.9
rlfap-Scens11	12	1,584,627.0	29.2	25.6	3.6
rlfap-Scens	11	1,210,624.8	26.0	23.4	2.5
tightness0.9	100	27,134.2	28.6	25.5	3.0

σ-ΔPPC^{sup2001} slightly outperforms σ-ΔPPC^{sup}. The better performance of the PC-2001-style supports on these benchmarks can be traced to the fact that domains of these benchmarks are totally ordered. In contrast, on the rlfap-Graphs benchmark, σ-ΔPPC^{sup} outperforms σ-ΔPPC^{sup2001} because the former is able to save many constraint checks.

In conclusion, we expect the new support structures to be more useful, in practice, on difficult problems, which are those where path consistency is most needed.

5.1.7 Benefits of Using Supports

In this section, we evaluate the impact of using support structures by comparing σ-ΔPPC to σ-ΔPPC^{sup}.

The complete results are reported in Tables B.20, B.21, and B.22 of Appendix B.[3] Generally speaking, we lose about 54.8% when no filtering occurs and gain about 13.9% when filtering occurs. The usefulness of support structures becomes more consequential in the context of search, which is beyond the scope of this thesis.

In Table 5.9, we select some representative results from Tables B.20, B.21, and B.22 with the tuples removed shown for reference. For the tightness benchmarks, as the tightness increases, support structures become increasingly more useful.

Table 5.9: Impact of using support structures

			CPU time (s)		
		# deleted tuples	**AC+σ-ΔPPC**	**AC+σ-ΔPPC^{sup}**	**Saving**
graphColoring-school	8	0.0	81.2	128.4	-47.2
os-taillard-4	30	550,541.0	154.1	129.8	24.3
os-taillard-5	30	559,583.8	1,265.8	1,059.1	203.7
rlfap-Graphs	14	743,935.3	141.1	73.3	67.8
tightness0.65	100	474.3	1.6	1.2	0.4
tightness0.8	100	4,030.1	6.8	3.5	3.3
tightness0.9	100	27,134.2	69.9	25.5	44.4

5.1.8 Improving PC-8

In this section, we evaluate the three improvements we proposed to the PC-8 algorithm in Section 4.4, namely, PC-8-Ordering, PC-8-Flag, and their combination,

[3] Table B.20 reports the benchmarks where no filtering occurs and Tables B.21 and B.22 report the benchmarks where filtering occurs.

PC-8$^+$. Tables B.23 and B.24 show the complete results for the benchmark problems. Table 5.10 reports some representative results.

Table 5.10: Improvements to PC-8

		# instances				CPU time (s)			
		AC+PC-8	**AC+PC-8$^+$**	**AC+PC-8-Flag**	**AC+PC-8-Ordering**	**AC+PC-8**	**AC+PC-8$^+$**	**AC+PC-8-Flag**	**AC+PC-8-Ordering**
driver	7	*All instances complete*				94.6	81.2	87.3	80.7
graphColoring-hos	14	14	13	13	14	2,786.7	3,121.7	3,107.0	2,801.8
rlfap-ScensMod	13	11	11	11	13	963.3	1,009.6	1,045.6	920.5
os-taillard-5	30	*All instances complete*				1,563.0	1,498.5	1,575.8	1,475.5

PC-8-Flag saves constraint checks with the use of the flag bit vector. Consequently, we would expect it to also save time. However, on the results reported in Tables B.23 and B.24 , the CPU time saving is not systematic and PC-8-Flag is sometimes more costly in CPU time than PC-8 (e.g., all benchmarks Table 5.10 except driver). Further, PC-8-Flag (and also PC-8$^+$) cannot complete one instance of the graphColoring-hos benchmark due to exceeding the memory limit. The PC-8-Ordering algorithm is able to complete two instances of the rlfap-ScensMod benchmark more than any other algorithm.

Generally speaking, PC-8-Ordering shows the best performance both in terms of CPU time and the number of completed instances, and even consistently outperforms PC-8$^+$.

5.1.9 Improving PC-2001

In this section, we briefly comment on the improvements to the PC-2001 algorithm, PC-2001-Ordering, PC-2001-Flag, and PC-2001$^+$. The complete results are re-

ported in Tables B.25 and B.25 of Appendix B.

Table 5.11 reports some representative results. Once again, the flag modification saves constraint checks and therefore time in theory, but, in practice, on most benchmarks, it increases the CPU time, though there are exceptions like the driver benchmark. Once again, PC-2001-Ordering consistently outperforms all other algorithms in terms of CPU time and number of instances completed.

Table 5.11: Improvements to PC-2001

			CPU time (s)			
		# instances	**AC+PC-2001**	**AC+PC-2001$^+$**	**AC+PC-2001-Flag**	**AC+PC-2001-Ordering**
driver	7	7	91.0	79.7	86.8	78.9
graphColoring-hos	14	4	249.2	277.3	277.4	249.3
os-taillard-5	30	30	1,236.7	1,248.0	1,253.5	1,229.6

5.1.10 Comparing Main Algorithms

In this section, we compare the best of our introduced algorithms, PC-8-Ordering, PC-2001-Ordering, and σ-ΔPPC^{sup} against the best published algorithms, DPC, sDC2, and sCDC1. The complete results are reported in Tables B.26 and B.27.

In general, the DPC algorithm is the quickest of all considered algorithms because:

1. It operates on a triangulated graph.

2. It sweeps only once through the triangles of the triangulated graph.

3. It updates only one edge of each triangle, while enforcing directional arc consistency.

Consequently, it filters significantly fewer tuples than any other algorithm. Whether or not this amount of filtering is sufficient needs to be determined by evaluating the resulting effectiveness of search, which is beyond the scope of this thesis.

In terms of the number of completed instances, both DPC and sCDC1 complete all 2,288 instances. Then, comes PC-8-Ordering completing 2,266 instances, followed by sDC2 completing 2,255 instances. The main obstacle to completing an instance is the need for memory space. The two algorithms with the largest support structures, σ-ΔPPC^{sup} and PC-2001-Ordering complete the fewest instances (2,193 and 2,128, respectively). Note that σ-ΔPPC^{sup} operates on a triangulated graph and has no propagation queue while PC-2001-Ordering operates on the complete graph and has a propagation queue of edges.

In terms of CPU time, and excluding DPC, sCDC1 ranks first, followed by σ-ΔPPC^{sup}, then sDC2. Further, PC-2001-Ordering outperforms PC-8-Ordering.

Table 5.12 shows selected representative results. Note that σ-ΔPPC^{sup} outperforms sCDC1 on three of the four reported benchmark problems.

Table 5.12: Comparing the best performing algorithms

		# instances						**CPU time (s)**					
		AC+DPC	**AC+PC-2001-Ordering**	**AC+PC-8-Ordering**	**AC+σ-ΔPPC^{sup}**	**AC+sDC2**	**sCDC1**	**AC+DPC**	**AC+PC-2001-Ordering**	**AC+PC-8-Ordering**	**AC+σ-ΔPPC^{sup}**	**AC+sDC2**	**sCDC1**
BH-4-13	7	*All instances complete*						37.1	556.4	524.8	100.2	653.7	182.3
rlfap-Scens	11	11	7	9	11	9	11	13.9	160.5	141.4	14.6	78.5	16.0
rlfap-ScensMod	13	13	7	13	13	12	13	1.7	132.5	142.1	2.2	100.8	4.8
os-taillard-5	30	*All instances complete*						924.7	1,229.6	1,475.5	1,059.1	1,187.3	1,012.1

5.2 Randomly Generated Problems

In this section, we compare the performance of the studied algorithms on randomly generated problems.

5.2.1 Problem Characteristics

Using a Model B generator, we consider four sets of binary CSP instances with 20 instances per data point. For each problem set, we vary the constraint tightness to study the area around the phase transition. The four data sets have the following $\langle n, a, t, d \rangle$ characteristics, where n is the number variables in the CSP, a is the domain size (the same for all variables), t is the constraint tightness ($t = \frac{\text{number of forbidden tuples}}{\text{total number of tuples}}$, the same for all constraints), d is the constraint ratio or density ($d = \frac{2e}{n(n-1)}$), and e is the number of constraints in the CSP:

$$\langle 50, 25, t, 20\% \rangle,\ \langle 50, 50, t, 20\% \rangle,\ \langle 50, 25, t, 40\% \rangle, \text{ and } \langle 50, 50, t, 40\% \rangle.$$

The goal of the above data sets is to compare the performance of the studied algorithm under relatively low (i.e., 20%) and high (i.e., 40%) constraint density and for problems with large (i.e., 25) and very large (i.e., 50) domains.

We run our experiments on the Crane supercomputer with a timeout limit of three hours per instance and 16 GiB of memory. These limits are large enough to allow every algorithm to complete every instance. We record the instruction count using the 'perf' tool and convert the number of instructions to a time value by assuming a 3.0 GHz CPU.

All our results assume pre-processing by AC before every algorithm except sCDC1.

5.2.2 Results

Below, we report the averages of the CPU time in a table and charts for the $\langle 50, 25, t, 20\% \rangle$ random data-set. While the tables and charts for other three data-sets are reported in Appendix C, we comment on all four data-sets throughout this section. Although the results are, generally speaking, similar, we do note some differences.

5.2.2.1 $\langle 50, 25, t, 20\% \rangle$

Table 5.13 reports the averages for all algorithms, listed from left to right in increasing cost. DPC is obviously the cheapest algorithm, but it also the one that does the least filtering. The performance of σ-ΔPPC^{sup} validates our approach as this algorithm outperforms all others.

Table 5.13: Comparing CPU times on random problems $\langle 50, 25, t, 20\% \rangle$

Tightness	**AC+DPC**	**AC+σ-ΔPPC^{sup}**	**sCDC1**	**AC+σ-ΔPPC^{sup2001}**	**AC+ΔPPC**	**AC+σ-ΔPPC**	**AC+PC-2001-Ordering**	**AC+PC-2001^{+}**	**AC+PPC+AP**	**AC+PC-2001**	**AC+PC-2001-Flag**	**AC+PC-8^{+}**	**AC+PC-8-Ordering**	**AC+PC-8-Flag**	**AC+PC-8**	**AC+sDC2**	**AC+PC-2**
52.0%	2.0	6.7	6.2	8.5	8.0	8.1	14.3	15.3	12.5	15.0	15.7	14.5	13.7	15.0	14.6	26.4	17.0
54.0%	2.1	7.0	7.5	9.6	9.3	9.4	15.2	16.0	13.5	16.3	16.8	15.4	14.8	16.5	16.4	31.2	19.9
56.0%	2.1	7.3	9.9	11.1	11.3	11.6	16.7	17.4	15.5	18.6	18.8	17.3	17.0	19.3	19.8	42.6	25.3
58.0%	2.1	8.2	15.2	14.6	16.4	16.8	19.8	20.2	21.6	22.6	22.4	21.5	21.7	24.6	25.9	64.5	37.3
59.0%	2.1	9.6	23.2	20.9	25.8	26.6	24.3	24.7	32.6	27.5	27.2	28.2	28.7	32.0	33.6	112.2	62.0
59.5%	2.1	14.3	27.0	34.0	45.7	47.1	50.2	50.4	53.0	53.4	54.1	69.8	70.6	74.1	75.7	132.7	142.4
60.0%	2.1	15.3	19.5	34.4	46.8	47.7	55.5	56.2	52.7	59.1	60.3	79.3	79.5	84.0	85.0	98.5	132.5
61.0%	2.1	9.8	5.0	12.6	15.2	14.5	38.2	38.2	13.8	41.0	42.1	49.8	49.2	54.3	54.4	22.7	49.5
62.0%	2.2	8.5	3.1	9.4	10.2	9.3	29.6	30.3	5.3	31.9	33.4	36.6	35.6	39.8	39.8	14.3	34.8
64.0%	2.3	6.6	1.4	6.7	5.8	5.7	16.7	17.9	3.4	16.1	17.3	17.5	16.4	16.7	15.7	6.2	14.7
66.0%	2.4	3.9	0.7	4.0	4.0	4.6	9.3	10.0	2.5	9.4	10.0	8.9	8.2	8.9	8.2	3.0	7.6

Figure 5.1 shows the best published algorithms. DPC is obviously the best on all random data-sets. sCDC1 is generally the next best algorithm, except in the case of

the $\langle 50, 50, t, 40\% \rangle$ data-set where it is outperformed by PC-8 and PC-2001, both of which enforce PC, a stronger consistency than CDC. On all data sets, PC-2001 is the fastest PC algorithm, followed by PC-8, sDC2, and finally PC-2.

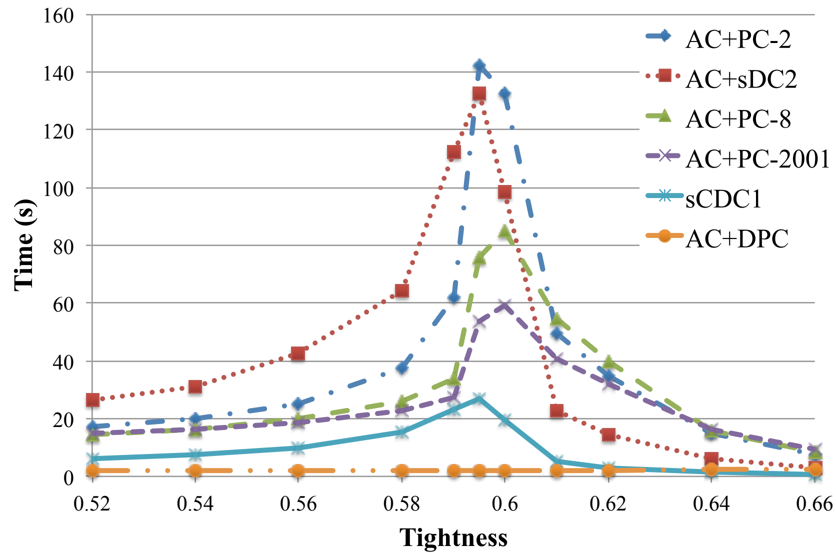


Figure 5.1: $\langle 50, 25, t, 20\% \rangle$: Comparing the best previously known algorithms

Figure 5.2 shows the performance of the most competitive algorithms, including our proposed algorithms. The performance of σ-ΔPPC^{sup} is clearly outstanding.

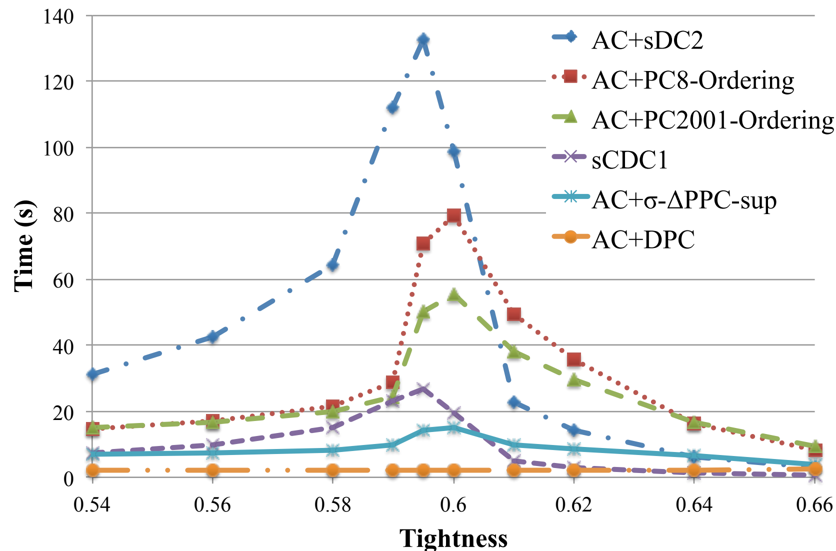


Figure 5.2: $\langle 50, 25, t, 20\% \rangle$: Comparing the most competitive algorithms

Figure 5.3 compares PC-8, PC-8-Ordering, PC-8-Flag, and PC-8$^+$. Interestingly, for all data sets, PC-8$^+$ outperforms all other algorithms. Both PC-8-Ordering and PC-8-Flag outperform PC-8 on all data sets.

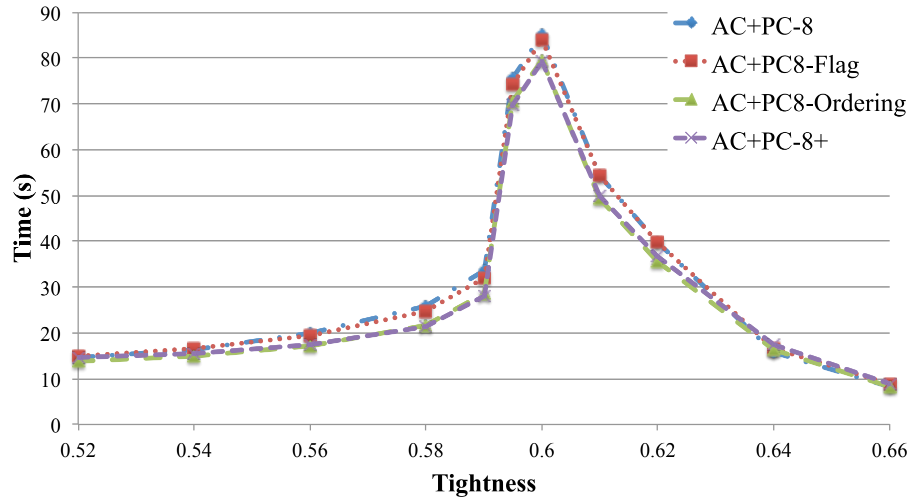


Figure 5.3: $\langle 50, 25, t, 20\% \rangle$: Improving PC-8

Figure 5.4 compares our improvements to PC-2001. Interestingly, on all data sets, PC-2001-Ordering and PC-2001$^+$ have comparable performance. In all data sets, they outperform PC-2001. PC-2001-Flag slightly outperforms PC-2001 for $a = 50$ and the opposite holds for $a = 25$.

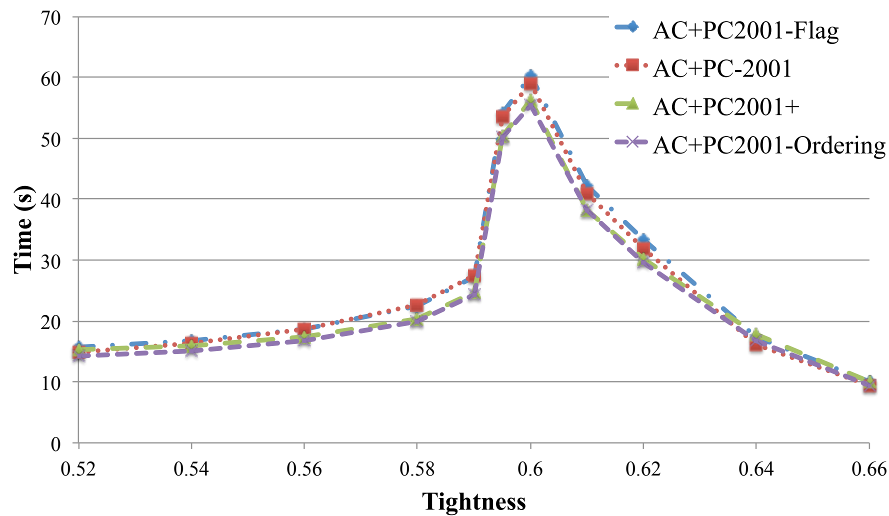


Figure 5.4: $\langle 50, 25, t, 20\% \rangle$: Improving PC-2001

Figure 5.5 compares the impact of the propagation queue and PEO. For 20% density, ΔPPC outperforms all other algorithms, but for 40% density, σ-ΔPPC outperforms all other algorithms.

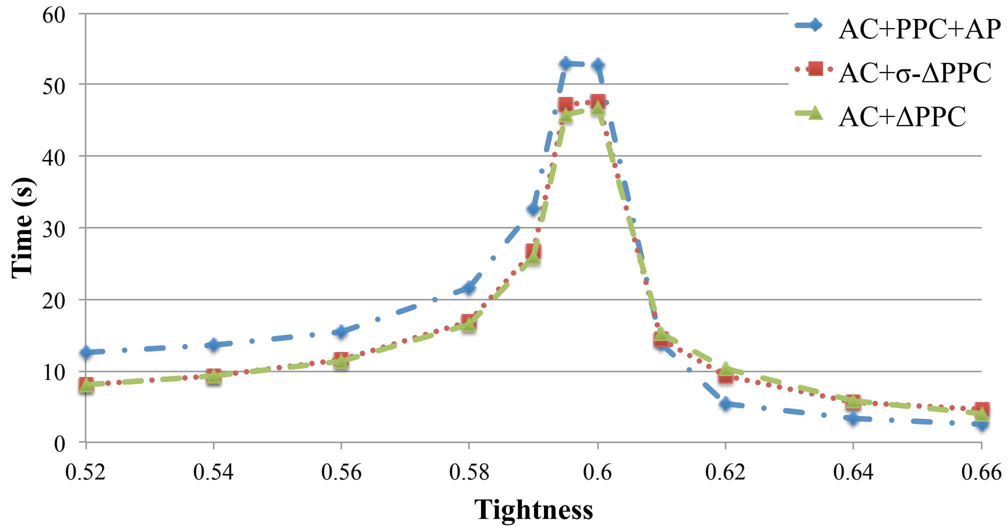


Figure 5.5: $\langle 50, 25, t, 20\% \rangle$: Impact of the propagation queue and PEO

In Figure 5.6, we assess the impact of support structures. For all data sets, σ-ΔPPC^{sup} outperforms σ-ΔPPC^{sup2001}, which, in turn, outperforms σ-ΔPPC.

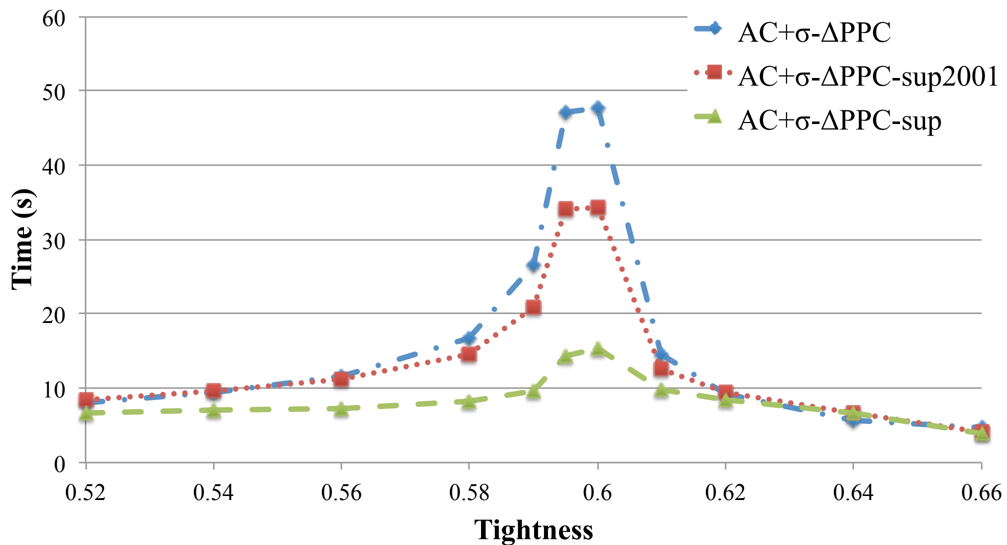


Figure 5.6: $\langle 50, 25, t, 20\% \rangle$: Impact of support structures

5.3 Discussion

Drawing from the results of our tests on benchmark and random CSPs, we conclude:

1. Pre-processing with arc consistency is an important step. It decisively reduces CPU time and, in the case of DPC, increases the filtering.

2. DPC is a very 'cheap' algorithm in terms of CPU time. While it yields less filtering than PC algorithms, one may want to consider enforcing DPC before search, at least on existing triangles if one does not want to add edges to the constraint graph. Enforcing DPC during search is not done in practice because it was thought to require fixing the instantiation ordering of the variables while search typically performs best with a dynamic variable ordering. However, recently the ConSystLab has argued that the fixed ordering of DPC should not be an impediment to running it as a look-ahead during search.

3. We note that among our PC-8 improvements, PC-8^+ outperforms all other versions of PC-8 on randomly generated problems. However, on benchmark problems, PC-8-ORDERING outperforms all others. The same comment holds for PC-2001^+ and PC-2001-ORDERING.

4. Performance on benchmark problems qualitatively differs from performance on randomly generated problems. Notably, there are many instances in the benchmarks where no filtering can be done. On problems with no or little filtering, the CPU time needed for creating and maintaining support structures is wasted, which is detrimental to the corresponding algorithms.

5. Supports can yield significant CPU-time savings, but can also be costly in terms of memory space. Generally speaking, it is possible to predict the size of the

supports before processing a given instance. It may be advantageous either to entirely opt out of using supports, or to delay generating and storing supports until the problem is small enough that doing so becomes advantageous.

6. Before ruling out any of the proposed algorithms, it is important to evaluate them in the context of search, either as a pre-processing step or as a look-ahead strategy. In this thesis, we focused our efforts to studying the existing algorithms and proposing new ones. Evaluating our algorithms in the context of search has to be left out to future research.

Summary

In this chapter, we discussed our empirical evaluations of the discussed and introduced algorithms, first on well-known benchmark problems, then on four randomly generated data-sets. Finally, we discussed our results.

Chapter 6

Conclusions and Future Work

In this thesis, we studied path consistency and its approximations, namely,

- *Strong approximation*: strong dual consistency.
- *Weak approximation*: directional path consistency and partial path consistency.
- *Incomparable*: strong conservative dual consistency.

We studied the main algorithms reported in the literature and proposed new ones that improve on the state of the art. A full list of our contributions is reported in Section 1.2. Below we identify new directions for future research:

1. Study modern bit-based implementations of consistency algorithms to improve implementation and practical performance.
2. Investigate hash-based techniques to reduce the cost of looking up tuples in table constraints.
3. Investigate strategies for generating and using supports selectively.
4. Investigate applying DPC before search on the existing triangles of the constraint graph.

5. Empirically determine the percentage of tested instances are path consistent but unsolvable.[1]

6. Implement the studied and proposed algorithms as a real-full look-ahead strategy during search [Haralick and Elliott, 1980].

[1]This evaluation was sugggested by Peter Revesz, a member of the Examining Committee.

Appendix A

Data Structures

The code is written in C++. Below, we describe the data structures of the variables, the variables' domains, and constraints in our code. Those structures are illustrated in Figure A.1.

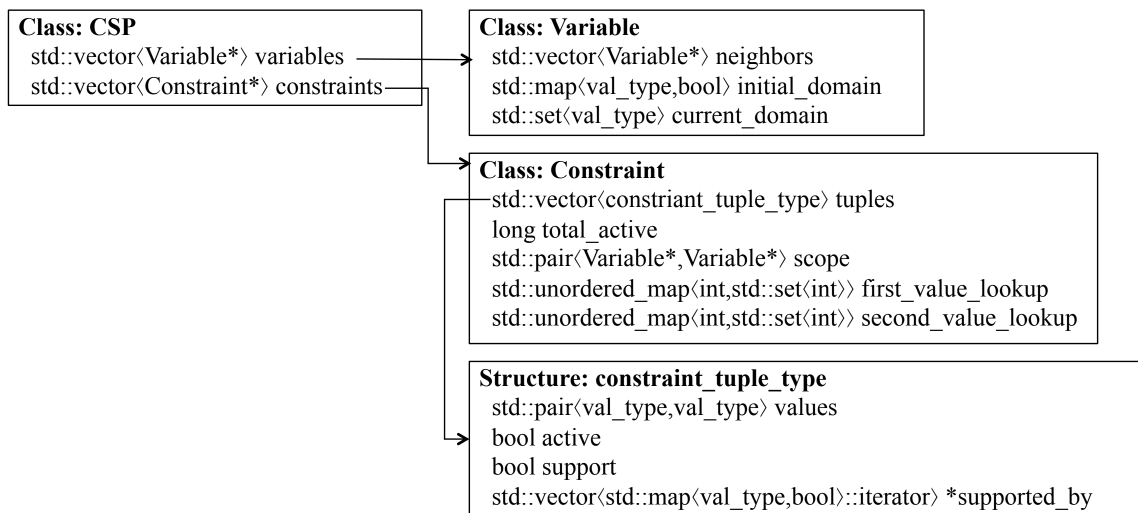


Figure A.1: The data structures.

The class `CSP` has two attributes:

1. `variables` stores `std::vector<Variable *>`, which is a vector of pointers to variables.

2. constraints stores std::vector<Constraint *>, which is a vector of pointers to constraints.

The Variable class has three attributes:

1. neighbors stores std::vector<Variable *>, which is a vector of pointers to the variables adjacent to the variable in the constraint graph.

2. fixed_domain stores std::map<val_type,bool>, which is a map of the sorted values in the initial domain of a CSP variable where bool indicates whether or not the value is 'alive.'

3. current_domain stores std::set<val_type>, which is the set of the domain values. The domain values are of the type val_type, which is an integer.

Note that the size of the attribute fixed_domain never changes after the problem is created while the size of current_domain can fluctuate as values are added and removed.

The Constraint class has five attributes:

1. tuples stores a one-dimensional vector, of type std::vector<constraint_tuple_type>, storing structures of type constraint_tuple_type, which has four attributes:

   a) values stores a tuple of values of type std::pair<val_type,val_type>.

   b) active is a Boolean indicating whether the tuple is active or deleted.

   c) support is a Boolean indicating whether the tuple is supported by a value. This attribute is only used in PC-8$^+$, PC-8-Flag, PC-2001$^+$, and PC-2001-Flag (see Section 4.4).

   d) *supported_by is a pointer to a vector of pointers, each of which points to the domain value (in fixed_domain) of a CSP variable that supports the

tuple. The vector of pointers is called std::vector<std::map<val_type,bool>::iterator>. This attribute is used only in PC-2001 (see Section 2.8.4).

2. total_active, which stores the number of active tuples in the relation.

3. scope stores a pair of pointers, std::pair<Variable*,Variable*>, to the variables in the scope of the constraint.

4. first_value_lookup is hashtable std::unordered_map<int,std::set<int>> providing for each value in the domain of the first variable in the scope of the constraint the initial set of values in the domain of the second variable with which it is consistent according to the constraint.

5. second_value_lookup is hashtable std::unordered_map<int,std::set<int>> providing for each value in the domain of the second variable in the scope of the constraint the initial set of values in the domain of the first variable with which it is consistent according to the constraint.

Note that the last two attributes, first_value_lookup and second_value_lookup, are redundant and used only for convenience. They are not updated during processing.

Appendix B

Results of Benchmark Problems

In this appendix, we include almost all the results of our experiments on benchmark problems, clearly stating and explaining omissions. In Chapter 5, we report only summaries of these results but discuss them in detail.

B.1 Pre-processing with Arc Consistency

In this section, we report the full results of comparing running arc consistency before each of the algorithms listed in Table B.1:

- Notably, we do not include sCDC1 because this algorithm enforces arc consistency as its first step.
- We omit all the graphColoring benchmarks from all tables because no tuples can be deleted by any of the discussed algorithms (i.e., both arc and path consistency), the time gained or lost by pre-processing is under one second, and the instances completed are the same with and without pre-processing with arc consistency.

- Further, we omit some instances of some benchmarks (i.e., frb and rand instances in Table B.2) and entire benchmarks (i.e., frb and rand in Tables B.3–B.7; composed, frd, and rand in Tables B.8–B.13) because pre-processing with AC does not impact performance.

- Unless we show the number of tuples deleted by each algorithm, the two algorithms compared in a given table remove the same number of tuples.

- Finally, the CPU time reported is averaged over the instances of a benchmark completed by *both* algorithms.

Table B.1: Enforcing arc consistency as a pre-processing step

Results	**Comparing**	**To**	**Not shown**
Table B.2	DPC	AC+DPC	Some instances of frb, rand
Table B.3	PC-2	AC+PC-2	frb, rand
Table B.4	PC-8	AC+PC-8	frb, rand
Table B.5	PC-8-Ordering	AC+PC-8-Ordering	frb, rand
Table B.6	PC-2001	AC+PC-2001	frb, rand
Table B.7	PC-2001-Ordering	AC+PC-2001-Ordering	frb, rand
Table B.8	PPC+AP	AC+PCC+AP	composed, frb, rand
Table B.9	ΔPPC	AC+ΔPPC	composed, frb, rand
Table B.10	σ-ΔPPC	AC+σ-ΔPPC	composed, frb, rand
Table B.11	σ-ΔPPC^{sup2001}	AC+σ-ΔPPC^{sup2001}	composed, frb, rand
Table B.12	σ-ΔPPC^{sup}	AC+σ-ΔPPC^{sup}	composed, frb, rand
Table B.13	sDC2	AC+sDC2	composed, frb, rand

Table B.2: Enforcing AC before DPC. AC+DPC and DPC complete the same number of instances

			# tuples deleted			CPU time (s)		
			DPC	**AC+DPC**	**Gain**	**DPC**	**AC+DPC**	**Saving**
	Total	**1,667**	**26,810,010.3**	**34,572,988.6**	**7,764,902.3**	**1,495.9**	**1,335.0**	**160.9**
	BH-4-13	7	723,666.0	2,340,635.0	1,616,969.0	42.2	37.1	5.1
	BH-4-4	10	10,903.5	53,312.0	42,408.5	0.2	0.2	0.0
	BH-4-7	20	73,480.0	204,924.0	131,444.0	2.5	2.3	0.2
	QCP-10	15	25,394.1	29,366.4	3,972.3	0.3	0.2	0.1
	QCP-15	15	157,944.9	179,737.8	21,792.9	2.2	0.8	1.4
	QCP-20	15	555,556.9	626,260.5	70,703.7	10.1	2.7	7.4
	QCP-25	15	1,449,297.2	1,620,407.6	171,110.4	34.9	7.6	27.3
	QWH-10	10	20,457.5	24,290.8	3,833.3	0.2	0.1	0.1
	QWH-15	10	124,499.1	144,940.5	20,441.4	1.6	0.5	1.1
	QWH-20	10	438,685.6	504,906.6	66,221.0	7.5	1.9	5.7
	QWH-25	10	1,145,354.8	1,306,672.0	161,317.2	26.2	5.2	21.0
	bqwh-15-106	100	263.0	384.1	121.1	0.1	0.1	0.0
	bqwh-18-141	100	342.7	474.9	132.2	0.2	0.2	0.0
	coloring	22	0.0	0.0	0.0	1.0	1.0	0.0
composed	25-1-2	10	*All instances are found inconsistent*			0.0	0.0	0.0
	25-1-25	10	1,805.0	2,014.0	209.0	0.0	0.0	0.0
	25-1-40	10	2,555.0	2,897.0	342.0	0.1	0.1	0.0
	25-1-80	10	*All instances are found inconsistent*			0.1	0.1	0.0
	25-10-20	10	1,008.1	1,094.7	86.6	0.1	0.2	0.0
	75-1-2	10	319.0	*Inconsistent*	-	0.2	0.2	0.0
	75-1-25	10	1,605.0	*Inconsistent*	-	0.2	0.2	0.0
	75-1-40	10	*All instances are found inconsistent*			0.2	0.2	0.0
	75-1-80	10	2,426.0	3,092.5	666.5	0.3	0.3	0.0
	driver	7	12,050.1	17,819.0	5,768.9	1.1	1.1	0.0
	frb30-15	10	103.4	124.0	20.6	0.2	0.2	0.0
	geom	100	0.0	0.0	0.0	0.5	0.5	0.0
	hanoi	5	0.0	191,558.6	191,558.6	0.2	2.0	-1.9
	langford	4	88,065.3	227,498.0	139,432.8	0.9	0.7	0.2
	lard	10	7,170,733.2	7,335,584.5	164,851.3	74.7	35.9	38.8
	marc	10	12,510,907.8	15,188,207.2	2,677,299.4	42.9	24.2	18.7
	os-taillard-4	30	162,989.8	187,485.2	24,495.4	109.9	109.2	0.7
	os-taillard-5	30	137,760.1	138,267.3	507.1	924.8	924.7	0.2
rand	2-30-15	50	71.2	81.3	10.2	0.1	0.1	0.0
	2-30-15-fcd	50	72.6	87.9	15.3	0.1	0.1	0.0
	2-40-19	50	34.7	43.8	9.1	0.4	0.4	0.0
	2-40-19-fcd	50	34.6	42.3	7.7	0.4	0.4	0.0
	2-50-23-fcd	50	16.8	26.4	9.6	1.0	1.1	0.0
rlfap	Graphs	14	279,465.0	311,367.0	31,902.0	99.7	90.9	8.8
	GraphsMod	12	270,571.4	842,489.1	571,917.7	54.4	32.9	21.5
	Scens11	12	529,989.3	1,333,950.8	803,961.5	16.2	14.6	1.7
	Scens	11	646,131.7	1,002,191.3	356,059.7	19.2	17.1	2.1
	ScensMod	13	257,339.6	742,643.4	485,303.8	6.1	5.2	0.9
	tightness0.1	100	0.9	0.9	0.0	0.1	0.1	0.0
	tightness0.2	100	2.0	2.0	0.0	0.1	0.1	0.0
	tightness0.35	100	9.3	9.3	0.0	0.2	0.2	0.0
	tightness0.5	100	27.1	27.1	0.0	0.4	0.4	0.0
	tightness0.65	100	149.1	149.1	0.0	0.5	0.5	0.0
	tightness0.8	100	1,127.1	1,127.7	0.6	1.6	1.6	0.0
	tightness0.9	100	6,794.8	6,794.8	0.0	9.6	9.6	0.0

Table B.3: Enforcing AC before PC-2. AC+PC-2 completes more instances than PC-2 and consistently saves time

			# instances			CPU time (s)		
			PC-2	AC+PC-2	Gain	PC-2	AC+PC-2	Saving
	Total	**1,407**	**1,372**	**1,394**	**22**	**17,723.3**	**12,211.1**	**5,512.2**
	BH-4-13	7	7	7	0	906.4	437.4	469.0
	BH-4-4	10	10	10	0	1.6	0.5	1.1
	BH-4-7	20	20	20	0	36.2	17.1	19.1
	QCP-10	15	15	15	0	1.2	0.6	0.6
	QCP-15	15	15	15	0	11.9	4.8	7.2
	QCP-20	15	15	15	0	70.8	26.1	44.7
	QCP-25	15	15	15	0	313.9	110.8	203.1
	QWH-10	10	10	10	0	1.1	0.6	0.5
	QWH-15	10	10	10	0	10.1	4.3	5.8
	QWH-20	10	10	10	0	61.6	23.8	37.8
	QWH-25	10	10	10	0	266.0	100.9	165.1
	bqwh-15-106	100	100	100	0	1.3	1.3	0.1
	bqwh-18-141	100	100	100	0	2.4	2.3	0.1
	coloring	22	22	22	0	8.6	8.6	0.0
composed	25-1-2	10	10	10	0	0.1	0.1	0.0
composed	25-1-25	10	10	10	0	0.1	0.1	0.0
composed	25-1-40	10	10	10	0	0.2	0.1	0.0
composed	25-1-80	10	10	10	0	0.2	0.2	0.0
composed	25-10-20	10	10	10	0	11.2	11.1	0.1
composed	75-1-2	10	10	10	0	1.8	1.7	0.0
composed	75-1-25	10	10	10	0	1.8	1.7	0.0
composed	75-1-40	10	10	10	0	1.8	1.7	0.0
composed	75-1-80	10	10	10	0	1.8	1.8	0.0
	driver	7	7	7	0	154.6	151.4	3.3
	geom	100	100	100	0	1.9	1.9	0.0
	hanoi	5	3	5	2	19.9	0.0	19.8
	langford	4	4	4	0	2.4	1.3	1.2
	lard	10	10	10	0	173.5	58.7	114.8
	marc	10	10	10	0	44.4	25.1	19.4
	os-taillard-4	30	30	30	0	215.5	212.4	3.1
	os-taillard-5	30	30	30	0	1,621.4	1,620.7	0.7
rlfap	Graphs	14	6	9	3	1,107.7	1,022.5	85.2
rlfap	GraphsMod	12	5	8	3	1,648.3	1,227.3	421.0
rlfap	Scens11	12	2	10	8	6,728.3	4,167.7	2,560.7
rlfap	Scens	11	5	9	4	1,424.9	787.9	637.0
rlfap	ScensMod	13	11	13	2	2,069.7	1,377.7	692.0
	tightness0.1	100	100	100	0	0.2	0.2	0.0
	tightness0.2	100	100	100	0	0.3	0.3	0.0
	tightness0.35	100	100	100	0	0.7	0.7	0.0
	tightness0.5	100	100	100	0	2.0	2.0	0.0
	tightness0.65	100	100	100	0	6.6	6.6	0.0
	tightness0.8	100	100	100	0	43.2	43.3	0.0
	tightness0.9	100	100	100	0	745.6	746.0	-0.4

Table B.4: Enforcing AC before PC-8. AC+PC-8 completes more instances than PC-8 and generally saves time

			# instances			CPU time (s)		
			PC-8	AC+PC-8	Gain	PC-8	AC+PC-8	Saving
	Total	**1,407**	**1,372**	**1,393**	**21**	**11,949.3**	**7,388.6**	**4,560.7**
	BH-4-13	7	7	7	0	1,488.2	536.2	952.0
	BH-4-4	10	10	10	0	2.5	0.5	2.0
	BH-4-7	20	20	20	0	59.3	20.9	38.4
	QCP-10	15	15	15	0	1.9	0.5	1.4
	QCP-15	15	15	15	0	20.8	3.5	17.4
	QCP-20	15	15	15	0	128.6	17.0	111.6
	QCP-25	15	15	15	0	548.7	57.8	490.8
	QWH-10	10	10	10	0	1.4	0.4	1.0
	QWH-15	10	10	10	0	16.1	2.7	13.4
	QWH-20	10	10	10	0	104.0	13.3	90.7
	QWH-25	10	10	10	0	438.0	44.4	393.5
	bqwh-15-106	100	100	100	0	1.1	1.0	0.1
	bqwh-18-141	100	100	100	0	2.2	2.0	0.2
	coloring	22	22	22	0	7.1	7.1	0.0
composed	25-1-2	10	10	10	0	0.2	0.2	0.0
composed	25-1-25	10	10	10	0	0.2	0.2	0.0
composed	25-1-40	10	10	10	0	0.2	0.2	0.0
composed	25-1-80	10	10	10	0	0.2	0.2	0.0
composed	25-10-20	10	10	10	0	13.2	12.9	0.2
composed	75-1-2	10	10	10	0	2.1	2.0	0.0
composed	75-1-25	10	10	10	0	2.1	2.0	0.0
composed	75-1-40	10	10	10	0	2.1	2.0	0.0
composed	75-1-80	10	10	10	0	2.1	2.1	0.0
	driver	7	7	7	0	99.2	94.6	4.5
	geom	100	100	100	0	2.2	2.2	0.0
	hanoi	5	3	5	2	31.3	0.0	31.3
	langford	4	4	4	0	3.6	1.8	1.8
	lard	10	10	10	0	279.3	67.7	211.6
	marc	10	10	10	0	54.6	25.0	29.6
	os-taillard-4	30	30	30	0	209.1	205.9	3.2
	os-taillard-5	30	30	30	0	1,563.6	1,563.0	0.6
rlfap	Graphs	14	8	9	1	922.2	810.2	112.0
rlfap	GraphsMod	12	7	10	3	2,319.3	1,905.6	413.7
rlfap	Scens11	12	0	9	9	-	-	-
rlfap	Scens	11	5	9	4	2,044.6	1,012.6	1,032.0
rlfap	ScensMod	13	9	11	2	996.9	389.6	607.3
	tightness0.1	100	100	100	0	0.2	0.2	0.0
	tightness0.2	100	100	100	0	0.4	0.4	0.0
	tightness0.35	100	100	100	0	0.7	0.7	0.0
	tightness0.5	100	100	100	0	1.9	1.9	0.0
	tightness0.65	100	100	100	0	5.9	5.9	0.0
	tightness0.8	100	100	100	0	42.2	42.1	0.1
	tightness0.9	100	100	100	0	530.0	529.9	0.1

Table B.5: Enforcing AC before PC-8-Ordering. AC+PC-8-Ordering completes more instances than PC-8-Ordering and consistently saves time

			# instances			CPU time (s)		
			PC-8-Ordering	**AC+PC-8-Ordering**	**Gain**	**PC-8-Ordering**	**AC+PC-8-Ordering**	**Saving**
	Total	**1,407**	**1,371**	**1,395**	**24**	**9,616.4**	**6,705.4**	**2,911.0**
	BH-4-13	7	7	7	0	1,071.0	524.8	546.2
	BH-4-4	10	10	10	0	2.0	0.5	1.4
	BH-4-7	20	20	20	0	42.6	20.4	22.2
	QCP-10	15	15	15	0	1.3	0.5	0.8
	QCP-15	15	15	15	0	13.7	3.5	10.2
	QCP-20	15	15	15	0	81.8	17.0	64.7
	QCP-25	15	15	15	0	340.5	58.1	282.5
	QWH-10	10	10	10	0	1.1	0.4	0.7
	QWH-15	10	10	10	0	10.8	2.7	8.0
	QWH-20	10	10	10	0	66.3	13.3	53.1
	QWH-25	10	10	10	0	271.0	44.6	226.4
	bqwh-15-106	100	100	100	0	1.1	1.0	0.1
	bqwh-18-141	100	100	100	0	2.0	1.9	0.1
	coloring	22	22	22	0	7.2	7.2	0.0
composed	25-1-2	10	10	10	0	0.2	0.2	0.0
	25-1-25	10	10	10	0	0.2	0.2	0.0
	25-1-40	10	10	10	0	0.2	0.2	0.0
	25-1-80	10	10	10	0	0.2	0.2	0.0
	25-10-20	10	10	10	0	12.8	12.7	0.2
	75-1-2	10	10	10	0	2.1	2.1	0.0
	75-1-25	10	10	10	0	2.1	2.1	0.0
	75-1-40	10	10	10	0	2.1	2.1	0.0
	75-1-80	10	10	10	0	2.1	2.1	0.0
	driver	7	7	7	0	82.8	80.7	2.1
	geom	100	100	100	0	2.2	2.2	0.0
	hanoi	5	3	5	2	23.6	0.0	23.5
	langford	4	4	4	0	3.2	1.6	1.6
	lard	10	10	10	0	212.7	67.9	144.8
	marc	10	10	10	0	49.3	25.0	24.3
	os-taillard-4	30	30	30	0	199.4	196.3	3.1
	os-taillard-5	30	30	30	0	1,475.4	1,475.5	-0.1
rlfap	Graphs	14	8	9	1	877.0	773.3	103.7
	GraphsMod	12	6	9	3	1,765.8	1,513.7	252.0
	Scens11	12	0	10	10	-	-	-
	Scens	11	5	9	4	1,758.4	961.7	796.7
	ScensMod	13	9	13	4	701.8	359.3	342.6
	tightness0.1	100	100	100	0	0.2	0.2	0.0
	tightness0.2	100	100	100	0	0.4	0.4	0.0
	tightness0.35	100	100	100	0	0.7	0.7	0.0
	tightness0.5	100	100	100	0	1.9	1.9	0.0
	tightness0.65	100	100	100	0	5.3	5.3	0.0
	tightness0.8	100	100	100	0	35.1	35.1	0.0
	tightness0.9	100	100	100	0	487.0	487.0	-0.1

Table B.6: Enforcing AC before PC-2001. AC+PC-2001 completes more instances than AC+PC-2001 and consistently saves time

		# instances			CPU time (s)		
		PC-2001	**AC+PC-2001**	**Gain**	**PC-2001**	**AC+PC-2001**	**Saving**
Total	**1,407**	**1,332**	**1,369**	**37**	**4,935.9**	**3,792.3**	**1,143.5**
BH-4-13	7	7	7	0	1,059.3	561.4	497.9
BH-4-4	10	10	10	0	2.1	0.6	1.5
BH-4-7	20	20	20	0	43.3	22.4	20.9
QCP-10	15	15	15	0	1.9	0.6	1.3
QCP-15	15	15	15	0	21.7	4.4	17.3
QCP-20	15	15	15	0	139.4	22.1	117.3
QCP-25	15	0	15	15	-	-	-
QWH-10	10	10	10	0	1.4	0.5	1.0
QWH-15	10	10	10	0	17.1	3.4	13.7
QWH-20	10	10	10	0	112.3	17.4	94.9
QWH-25	10	0	10	10	-	-	-
bqwh-15-106	100	100	100	0	1.1	1.0	0.1
bqwh-18-141	100	100	100	0	2.1	2.0	0.1
coloring	22	22	22	0	7.6	7.6	0.0
composed 25-1-2	10	10	10	0	0.2	0.2	0.0
composed 25-1-25	10	10	10	0	0.2	0.2	0.0
composed 25-1-40	10	10	10	0	0.2	0.2	0.0
composed 25-1-80	10	10	10	0	0.2	0.2	0.0
composed 25-10-20	10	10	10	0	11.3	11.1	0.2
composed 75-1-2	10	10	10	0	2.2	2.2	0.0
composed 75-1-25	10	10	10	0	2.2	2.2	0.0
composed 75-1-40	10	10	10	0	2.3	2.2	0.0
composed 75-1-80	10	10	10	0	2.2	2.2	0.0
driver	7	7	7	0	95.5	91.0	4.5
geom	100	100	100	0	2.4	2.4	0.0
hanoi	5	3	5	2	30.7	0.0	30.7
langford	4	4	4	0	2.6	1.7	0.9
lard	10	10	10	0	233.9	83.0	150.9
marc	10	10	10	0	67.7	37.8	29.9
os-taillard-4	30	30	30	0	155.6	153.5	2.1
os-taillard-5	30	30	30	0	1,236.9	1,236.7	0.3
rlfap Graphs	14	3	8	5	762.5	702.1	60.5
rlfap GraphsMod	12	2	2	0	0.1	0.1	0.0
rlfap Scens11	12	0	0	0	-	-	-
rlfap Scens	11	2	7	5	324.9	298.8	26.1
rlfap ScensMod	13	7	7	0	212.1	141.1	71.1
tightness0.1	100	100	100	0	0.2	0.2	0.0
tightness0.2	100	100	100	0	0.4	0.4	0.0
tightness0.35	100	100	100	0	0.8	0.8	0.0
tightness0.5	100	100	100	0	2.1	2.1	0.0
tightness0.65	100	100	100	0	5.9	5.9	0.0
tightness0.8	100	100	100	0	35.5	35.2	0.3
tightness0.9	100	100	100	0	335.7	335.4	0.2

Table B.7: Enforcing AC before PC-2001-Ordering. AC+PC-2001-Ordering completes more instances than PC-2001-Ordering and consistently saves time

			# instances			CPU time (s)		
			PC-2001-Ordering	**AC+PC-2001-Ordering**	**Gain**	**PC-2001-Ordering**	**AC+PC-2001-Ordering**	**Saving**
	Total	**1,407**	**1,332**	**1,369**	**37**	**4,498.9**	**3,701.5**	**797.4**
	BH-4-13	7	7	7	0	874.8	556.4	318.4
	BH-4-4	10	10	10	0	1.8	0.6	1.1
	BH-4-7	20	20	20	0	35.7	22.1	13.5
	QCP-10	15	15	15	0	1.5	0.6	0.9
	QCP-15	15	15	15	0	16.3	4.4	11.9
	QCP-20	15	15	15	0	102.6	22.0	80.6
	QCP-25	15	0	15	15	-	-	-
	QWH-10	10	10	10	0	1.1	0.4	0.7
	QWH-15	10	10	10	0	12.9	3.4	9.5
	QWH-20	10	10	10	0	82.8	17.3	65.5
	QWH-25	10	0	10	10	-	-	-
	bqwh-15-106	100	100	100	0	1.0	1.0	0.0
	bqwh-18-141	100	100	100	0	2.0	1.9	0.1
	coloring	22	22	22	0	7.6	7.6	0.0
composed	25-1-2	10	10	10	0	0.2	0.2	0.0
composed	25-1-25	10	10	10	0	0.2	0.2	0.0
composed	25-1-40	10	10	10	0	0.2	0.2	0.0
composed	25-1-80	10	10	10	0	0.2	0.2	0.0
composed	25-10-20	10	10	10	0	11.1	10.9	0.1
composed	75-1-2	10	10	10	0	2.2	2.2	0.0
composed	75-1-25	10	10	10	0	2.2	2.2	0.0
composed	75-1-40	10	10	10	0	2.2	2.2	0.0
composed	75-1-80	10	10	10	0	2.2	2.2	0.0
	driver	7	7	7	0	81.4	78.9	2.5
	geom	100	100	100	0	2.4	2.4	0.0
	hanoi	5	3	5	2	27.0	0.0	26.9
	langford	4	4	4	0	2.4	1.6	0.8
	lard	10	10	10	0	189.8	83.0	106.8
	marc	10	10	10	0	62.6	37.8	24.7
	os-taillard-4	30	30	30	0	155.0	153.0	1.9
	os-taillard-5	30	30	30	0	1,229.5	1,229.6	-0.1
rlfap	Graphs	14	3	8	5	729.3	676.6	52.7
rlfap	GraphsMod	12	2	2	0	0.1	0.1	0.0
rlfap	Scens11	12	0	0	0	-	-	-
rlfap	Scens	11	2	7	5	313.1	286.9	26.2
rlfap	ScensMod	13	7	7	0	184.7	132.5	52.2
	tightness0.1	100	100	100	0	0.2	0.2	0.0
	tightness0.2	100	100	100	0	0.4	0.4	0.0
	tightness0.35	100	100	100	0	0.8	0.8	0.0
	tightness0.5	100	100	100	0	2.0	2.0	0.0
	tightness0.65	100	100	100	0	5.5	5.5	0.0
	tightness0.8	100	100	100	0	31.4	31.4	0.0
	tightness0.9	100	100	100	0	320.4	320.1	0.2

Table B.8: Enforcing AC before PPC+AP. Both algorithms complete 1,837 instances but pre-processing with AC generally saves time

			CPU time (s)		
			PPC+AP	**AC+PPC+AP**	**Saving**
	Total	**1,317**	**4,123.5**	**3,473.1**	**650.4**
	BH-4-13	7	347.4	147.6	199.8
	BH-4-4	10	0.8	0.3	0.6
	BH-4-7	20	14.5	6.8	7.7
	QCP-10	15	1.1	0.6	0.5
	QCP-15	15	8.9	3.4	5.5
	QCP-20	15	45.4	13.8	31.6
	QCP-25	15	162.9	40.4	122.5
	QWH-10	10	0.8	0.4	0.4
	QWH-15	10	5.9	2.2	3.8
	QWH-20	10	32.1	8.6	23.5
	QWH-25	10	111.6	25.0	86.6
	bqwh-15-106	100	0.5	0.5	0.0
	bqwh-18-141	100	1.1	1.1	0.0
	coloring	22	8.3	8.3	0.0
	driver	7	7.5	7.5	-0.1
	geom	100	1.0	1.0	0.0
	hanoi	5	0.2	2.0	-1.8
	langford	4	4.3	1.9	2.3
	lard	10	215.8	127.2	88.6
	marc	10	46.1	27.4	18.8
	os-taillard-4	30	176.1	174.0	2.1
	os-taillard-5	30	1,502.6	1,502.9	-0.2
rlfap	Graphs	14	1,054.6	1,018.7	35.9
rlfap	GraphsMod	12	163.0	148.6	14.4
rlfap	Scens11	12	54.2	50.6	3.6
rlfap	Scens	11	41.8	39.4	2.4
rlfap	ScensMod	13	15.2	13.3	1.9
	tightness0.1	100	0.4	0.4	0.0
	tightness0.2	100	0.5	0.5	0.0
	tightness0.35	100	0.7	0.7	0.0
	tightness0.5	100	1.2	1.2	0.0
	tightness0.65	100	2.2	2.2	0.0
	tightness0.8	100	9.2	9.2	0.0
	tightness0.9	100	85.3	85.3	0.1

Table B.9: Enforcing AC before ΔPPC. Both algorithms complete 1,837 instances but pre-processing with AC generally saves time

			CPU time (s)		
			ΔPPC	**AC+ΔPPC**	**Saving**
	Total	**1,317**	**3,138.4**	**2,483.9**	**654.5**
	BH-4-13	7	170.2	79.4	90.8
	BH-4-4	10	0.5	0.2	0.3
	BH-4-7	20	7.7	3.9	3.8
	QCP-10	15	1.0	0.4	0.6
	QCP-15	15	9.0	2.0	7.0
	QCP-20	15	43.9	8.0	35.9
	QCP-25	15	165.7	22.7	143.1
	QWH-10	10	0.8	0.3	0.5
	QWH-15	10	6.3	1.5	4.8
	QWH-20	10	33.6	5.6	28.0
	QWH-25	10	118.4	15.3	103.1
	bqwh-15-106	100	0.4	0.3	0.0
	bqwh-18-141	100	0.8	0.7	0.0
	coloring	22	5.0	5.0	0.0
	driver	7	6.0	5.9	0.0
	geom	100	0.6	0.6	0.0
	hanoi	5	0.2	2.0	-1.8
	langford	4	2.9	1.3	1.6
	lard	10	181.3	58.9	122.5
	marc	10	52.2	25.2	26.9
	os-taillard-4	30	156.7	154.6	2.1
	os-taillard-5	30	1,257.4	1,256.9	0.5
rlfap	Graphs	14	618.1	583.2	35.0
rlfap	GraphsMod	12	145.4	106.0	39.4
rlfap	Scens11	12	35.8	30.2	5.6
rlfap	Scens	11	29.4	26.7	2.7
rlfap	ScensMod	13	11.3	9.1	2.2
	tightness0.1	100	0.2	0.2	0.0
	tightness0.2	100	0.2	0.2	0.0
	tightness0.35	100	0.4	0.4	0.0
	tightness0.5	100	0.8	0.8	0.0
	tightness0.65	100	1.5	1.5	0.0
	tightness0.8	100	6.7	6.8	0.0
	tightness0.9	100	68.0	68.0	0.0

Table B.10: Enforcing AC before σ-ΔPPC. Both algorithms complete 1,837 instances but pre-processing with AC generally saves time

			CPU time (s)		
			σ-ΔPPC	**AC+σ-ΔPPC**	**Saving**
	Total	**1,317**	**3,752.0**	**2,472.9**	**1,279.1**
	BH-4-13	7	208.2	79.4	128.8
	BH-4-4	10	0.5	0.2	0.3
	BH-4-7	20	9.0	3.9	5.1
	QCP-10	15	1.1	0.3	0.9
	QCP-15	15	13.4	1.4	12.0
	QCP-20	15	84.6	5.5	79.0
	QCP-25	15	345.1	15.7	329.4
	QWH-10	10	0.9	0.2	0.6
	QWH-15	10	9.4	1.0	8.3
	QWH-20	10	60.3	3.9	56.4
	QWH-25	10	243.6	10.3	233.3
	bqwh-15-106	100	0.3	0.3	0.0
	bqwh-18-141	100	0.6	0.5	0.0
	coloring	22	2.6	2.6	0.0
	driver	7	4.5	4.5	0.0
	geom	100	0.6	0.6	0.0
	hanoi	5	0.2	2.0	-1.8
	langford	4	2.8	1.2	1.5
	lard	10	199.9	58.7	141.2
	marc	10	67.5	25.0	42.5
	os-taillard-4	30	156.0	154.1	1.9
	os-taillard-5	30	1,262.5	1,262.8	-0.3
rlfap	Graphs	14	742.6	589.3	153.4
rlfap	GraphsMod	12	177.9	103.3	74.7
rlfap	Scens11	12	35.9	30.3	5.6
rlfap	Scens	11	31.0	27.0	4.0
rlfap	ScensMod	13	11.3	8.9	2.3
	tightness0.1	100	0.2	0.2	0.0
	tightness0.2	100	0.2	0.2	0.0
	tightness0.35	100	0.4	0.4	0.0
	tightness0.5	100	0.8	0.8	0.0
	tightness0.65	100	1.6	1.6	0.0
	tightness0.8	100	6.8	6.8	0.0
	tightness0.9	100	69.9	69.9	0.0

Table B.11: Enforcing AC before σ-ΔPPC^{sup2001}. Pre-processing with AC allows us to complete more instances and saves time

			# instances			**CPU time (s)**		
			σ-ΔPPC^{sup2001}	**AC+σ-ΔPPC^{sup2001}**	**Gain**	**σ-ΔPPC^{sup2001}**	**AC+σ-ΔPPC^{sup2001}**	**Saving**
	Total	**1,3717**	**1,300**	**1,309**	**9**	**2,255.1**	**1,673.4**	**581.6**
	BH-4-13	7	7	7	0	170.0	95.7	74.3
	BH-4-4	10	10	10	0	0.4	0.2	0.2
	BH-4-7	20	20	20	0	7.8	4.7	3.1
	QCP-10	15	15	15	0	0.9	0.4	0.5
	QCP-15	15	15	15	0	7.8	2.1	5.7
	QCP-20	15	15	15	0	43.8	8.8	35.1
	QCP-25	15	7	15	8	167.4	27.1	140.3
	QWH-10	10	10	10	0	0.7	0.3	0.4
	QWH-15	10	10	10	0	5.5	1.4	4.0
	QWH-20	10	10	10	0	31.5	6.1	25.4
	QWH-25	10	10	10	0	118.5	17.5	101.0
	bqwh-15-106	100	100	100	0	0.3	0.3	0.0
	bqwh-18-141	100	100	100	0	0.6	0.6	0.0
	coloring	22	22	22	0	3.6	3.6	0.0
	driver	7	7	7	0	5.2	5.0	0.2
	frb30-15	10	10	10	0	0.3	0.3	0.0
	frb35-17	10	10	10	0	0.5	0.5	0.0
	frb40-19	10	10	10	0	0.9	0.9	0.0
	frb45-21	10	10	10	0	1.4	1.4	0.0
	frb50-23	10	10	10	0	2.2	2.2	0.0
	frb53-24	10	10	10	0	2.7	2.8	0.0
	frb56-25	10	10	10	0	3.4	3.4	0.0
	frb59-26	10	10	10	0	4.2	4.2	0.0
	geom	100	100	100	0	0.7	0.7	0.0
	hanoi	5	5	5	0	0.2	2.0	-1.8
	langford	4	4	4	0	2.5	1.4	1.1
	lard	10	10	10	0	174.7	81.2	93.5
	marc	10	10	10	0	97.5	28.2	69.3
	os-taillard-4	30	30	30	0	129.5	127.7	1.9
	os-taillard-5	30	30	30	0	1,055.9	1,056.1	-0.2
rlfap	Graphs	14	10	10	0	105.5	90.2	15.3
rlfap	GraphsMod	12	7	8	1	12.9	12.0	0.9
rlfap	Scens11	12	12	12	0	34.6	29.2	5.4
rlfap	Scens	11	11	11	0	29.6	26.0	3.6
rlfap	ScensMod	13	13	13	0	11.2	8.7	2.5
	tightness0.1	100	100	100	0	0.2	0.2	0.0
	tightness0.2	100	100	100	0	0.3	0.3	0.0
	tightness0.35	100	100	100	0	0.4	0.5	0.0
	tightness0.5	100	100	100	0	0.9	0.9	0.0
	tightness0.65	100	100	100	0	1.4	1.4	0.0
	tightness0.8	100	100	100	0	4.4	4.4	0.0
	tightness0.9	100	100	100	0	28.5	28.6	0.0

Table B.12: Enforcing AC before σ-ΔPPC^{sup}. Pre-processing with AC allows us to complete more instances and generally saves time

			# instances			CPU time (s)		
			σ-ΔPPC^{sup}	**AC+σ-ΔPPC^{sup}**	**Gain**	**σ-ΔPPC^{sup}**	**AC+σ-ΔPPC^{sup}**	**Saving**
	Total	**1,317**	**1,300**	**1,309**	**9**	**2,217.9**	**1,658.8**	**559.0**
	BH-4-13	7	7	7	0	173.5	100.2	73.3
	BH-4-4	10	10	10	0	0.4	0.2	0.2
	BH-4-7	20	20	20	0	7.8	4.8	3.0
	QCP-10	15	15	15	0	0.8	0.4	0.4
	QCP-15	15	15	15	0	7.4	2.0	5.4
	QCP-20	15	15	15	0	42.1	8.5	33.6
	QCP-25	15	7	15	8	162.4	26.2	136.2
	QWH-10	10	10	10	0	0.6	0.2	0.4
	QWH-15	10	10	10	0	5.2	1.3	3.9
	QWH-20	10	10	10	0	30.3	5.6	24.7
	QWH-25	10	10	10	0	114.8	16.4	98.4
	bqwh-15-106	100	100	100	0	0.3	0.2	0.0
	bqwh-18-141	100	100	100	0	0.6	0.6	0.0
	coloring	22	22	22	0	3.7	3.7	0.0
	driver	7	7	7	0	3.8	3.6	0.2
	geom	100	100	100	0	0.7	0.7	0.0
	hanoi	5	5	5	0	0.2	2.0	-1.8
	langford	4	4	4	0	2.4	1.3	1.1
	lard	10	10	10	0	170.8	91.0	79.8
	marc	10	10	10	0	99.7	28.2	71.5
	os-taillard-4	30	30	30	0	131.9	129.8	2.1
	os-taillard-5	30	30	30	0	1,059.1	1,059.1	0.0
rlfap	Graphs	14	10	10	0	88.8	73.3	15.5
	GraphsMod	12	7	8	1	11.3	10.4	0.9
	Scens11	12	12	12	0	30.2	25.6	4.5
	Scens	11	11	11	0	27.0	23.4	3.5
	ScensMod	13	13	13	0	10.3	8.0	2.4
	tightness0.1	100	100	100	0	0.2	0.2	0.0
	tightness0.2	100	100	100	0	0.3	0.3	0.0
	tightness0.35	100	100	100	0	0.4	0.4	0.0
	tightness0.5	100	100	100	0	0.8	0.8	0.0
	tightness0.65	100	100	100	0	1.2	1.2	0.0
	tightness0.8	100	100	100	0	3.5	3.5	0.0
	tightness0.9	100	100	100	0	25.5	25.5	0.0

Table B.13: Enforcing AC before sDC2. Pre-processing with AC allows us to complete more instances and consistently saves time

			# instances			CPU time (s)		
			sDC2	AC+sDC2	Gain	sDC2	AC+sDC2	Saving
	Total	**1,317**	**1,286**	**1,294**	**8**	**9,172.8**	**8,950.5**	**222.3**
	BH-4-13	7	7	7	0	656.0	653.7	2.3
	BH-4-4	10	10	10	0	0.6	0.6	0.1
	BH-4-7	20	20	20	0	27.2	27.1	0.1
	QCP-10	15	15	15	0	0.4	0.3	0.1
	QCP-15	15	15	15	0	3.8	2.9	0.9
	QCP-20	15	15	15	0	22.2	18.1	4.1
	QCP-25	15	15	15	0	74.1	61.8	12.4
	QWH-10	10	10	10	0	1.2	1.1	0.1
	QWH-15	10	10	10	0	9.3	8.5	0.7
	QWH-20	10	10	10	0	47.3	43.6	3.7
	QWH-25	10	10	10	0	104.9	93.7	11.3
	bqwh-15-106	100	100	100	0	2.8	2.8	0.0
	bqwh-18-141	100	100	100	0	6.2	6.2	0.0
	coloring	22	22	22	0	2.4	2.4	0.0
	driver	7	7	7	0	49.6	49.5	0.1
	geom	100	100	100	0	2.5	2.5	0.0
	hanoi	5	3	5	2	4.1	0.0	4.0
	langford	4	4	4	0	1.8	1.8	0.0
	lard	10	10	10	0	34.2	34.1	0.0
	marc	10	10	10	0	23.7	23.7	0.0
	os-taillard-4	30	30	30	0	135.3	135.2	0.2
	os-taillard-5	30	30	30	0	1,187.2	1,187.3	-0.1
rlfap	Graphs	14	8	9	1	698.5	634.1	64.4
rlfap	GraphsMod	12	7	8	1	379.2	359.6	19.7
rlfap	Scens11	12	1	1	0	3,419.9	3,367.4	52.5
rlfap	Scens	11	5	9	4	500.9	470.9	30.1
rlfap	ScensMod	13	12	12	0	1,047.8	1,031.3	16.6
	tightness0.1	100	100	100	0	0.3	0.3	0.0
	tightness0.2	100	100	100	0	0.6	0.6	0.0
	tightness0.35	100	100	100	0	1.4	1.4	0.0
	tightness0.5	100	100	100	0	3.6	3.6	0.0
	tightness0.65	100	100	100	0	10.4	10.4	0.0
	tightness0.8	100	100	100	0	69.2	69.3	-0.1
	tightness0.9	100	100	100	0	644.3	644.9	-0.6

B.2 Propagation Queue: Edges versus Triangles

In this section, we report the detailed results comparing the impact of using triangles instead of edges in the propagation queue. Except for three instances where we run out of memory space, handling triangles generally saves CPU time (see discussion in Section 5.1.4). Otherwise, the number of deleted tuples is not affected.

Table B.14: AC+PPC+AP vs AC+ΔPPC (table 1 of 2)

			# instances			CPU time (s)		
			AC+PPC+AP	**AC+ΔPPC**	**Gain**	**AC+PPC+AP**	**AC+ΔPPC**	**Saving**
	Total	**2,288**	**2,288**	**2,285**	**-3**	**6,229.1**	**3,576.1**	**2,653.0**
	BH-4-13	7	7	7	0	147.6	79.4	68.2
	BH-4-4	10	10	10	0	0.3	0.2	0.1
	BH-4-7	20	20	20	0	6.8	3.9	2.8
	QCP-10	15	15	15	0	0.6	0.4	0.2
	QCP-15	15	15	15	0	3.4	2.0	1.4
	QCP-20	15	15	15	0	13.8	8.0	5.9
	QCP-25	15	15	15	0	40.4	22.7	17.7
	QWH-10	10	10	10	0	0.4	0.3	0.1
	QWH-15	10	10	10	0	2.2	1.5	0.7
	QWH-20	10	10	10	0	8.6	5.6	3.0
	QWH-25	10	10	10	0	25.0	15.3	9.7
	bqwh-15-106	100	100	100	0	0.5	0.3	0.1
	bqwh-18-141	100	100	100	0	1.1	0.7	0.4
	coloring	22	22	22	0	8.3	5.0	3.4
composed	25-1-2	10	10	10	0	0.1	0.1	0.0
composed	25-1-25	10	10	10	0	0.1	0.1	0.0
composed	25-1-40	10	10	10	0	0.1	0.1	0.0
composed	25-1-80	10	10	10	0	0.2	0.1	0.0
composed	25-10-20	10	10	10	0	0.6	0.4	0.2
composed	75-1-2	10	10	10	0	0.5	0.6	-0.1
composed	75-1-25	10	10	10	0	0.6	0.7	-0.1
composed	75-1-40	10	10	10	0	0.6	0.7	-0.1
composed	75-1-80	10	10	10	0	0.7	0.8	-0.1
	driver	7	7	7	0	7.5	5.9	1.6
	frb30-15	10	10	10	0	0.4	0.3	0.2
	frb35-17	10	10	10	0	0.8	0.5	0.3
	frb40-19	10	10	10	0	1.3	0.8	0.6
	frb45-21	10	10	10	0	2.3	1.2	1.1
	frb50-23	10	10	10	0	3.5	1.8	1.7
	frb53-24	10	10	10	0	4.5	2.2	2.2
	frb56-25	10	10	10	0	5.5	2.7	2.8
	frb59-26	10	10	10	0	6.9	3.4	3.5
	geom	100	100	100	0	1.0	0.6	0.4

Results continue in next table.

Table B.15: AC+PPC+AP vs AC+ΔPPC (table 2 of 2)

			# instances			CPU time (s)		
			AC+PPC+AP	**AC+ΔPPC**	**Gain**	**AC+PPC+AP**	**AC+ΔPPC**	**Saving**
graphColoring	hos	14	14	14	0	49.0	23.9	25.1
	full-insertion	40	40	37	-3	170.2	74.4	95.8
	k-insertion	32	32	32	0	26.0	13.4	12.6
	leighton-15	26	26	26	0	476.2	176.9	299.4
	leighton-25	31	31	31	0	1,084.6	385.1	699.4
	leighton-5	8	8	8	0	73.0	37.4	35.6
	mug	8	8	8	0	0.0	0.0	0.0
	myciel	16	16	16	0	0.9	0.4	0.4
	register-fpsol	37	37	37	0	175.3	80.4	94.9
	register-inithx	31	31	31	0	175.3	84.4	90.9
	register-mulsol	47	47	47	0	26.5	13.1	13.3
	register-zeroin	31	31	31	0	34.6	16.4	18.2
	school	8	8	8	0	280.9	105.2	175.7
	sgb-book	26	26	26	0	0.3	0.2	0.1
	sgb-games	4	4	4	0	0.8	0.4	0.4
	sgb-miles	42	42	42	0	79.6	33.9	45.7
	sgb-queen	50	50	50	0	56.9	21.0	35.9
	hanoi	5	5	5	0	2.0	2.0	0.0
	langford	4	4	4	0	1.9	1.3	0.6
	lard	10	10	10	0	127.2	58.9	68.3
	marc	10	10	10	0	27.4	25.2	2.2
	os-taillard-4	30	30	30	0	174.0	154.6	19.3
	os-taillard-5	30	30	30	0	1,502.9	1,256.9	246.0
rand	23-Feb	10	10	10	0	0.8	0.5	0.3
	24-Feb	10	10	10	0	1.0	0.6	0.4
	25-Feb	10	10	10	0	1.2	0.7	0.5
	26-Feb	10	10	10	0	1.5	0.9	0.6
	27-Feb	10	10	10	0	1.8	1.0	0.7
	2-30-15	50	50	50	0	0.4	0.3	0.2
	2-30-15-fcd	50	50	50	0	0.4	0.3	0.2
	2-40-19	50	50	50	0	1.3	0.7	0.6
	2-40-19-fcd	50	50	50	0	1.4	0.7	0.6
	2-50-23	50	50	50	0	3.6	1.8	1.8
	2-50-23-fcd	50	50	50	0	3.6	1.8	1.8
rlfap	Graphs	14	14	14	0	1,018.7	583.2	435.5
	GraphsMod	12	12	12	0	148.6	106.0	42.6
	Scens11	12	12	12	0	50.6	30.2	20.4
	Scens	11	11	11	0	39.4	26.7	12.7
	ScensMod	13	13	13	0	13.3	9.1	4.2
	tightness0.1	100	100	100	0	0.4	0.2	0.2
	tightness0.2	100	100	100	0	0.5	0.2	0.2
	tightness0.35	100	100	100	0	0.7	0.4	0.3
	tightness0.5	100	100	100	0	1.2	0.8	0.4
	tightness0.65	100	100	100	0	2.2	1.5	0.7
	tightness0.8	100	100	100	0	9.2	6.8	2.5
	tightness0.9	100	100	100	0	85.3	68.0	17.2

B.3 Propagating Along a PEO

Propagating along a PEO, without a propagation queue, allows us to save CPU time, memory space, and complete all 2,288 instances. Both algorithms, AC+ΔPPC and AC+σ-ΔPPC, remove the same number of tuples on completed instances. Further, they complete all instances of the BH, frb, composed, and rand benchmark and have comparable performance on those benchmarks (omitted from the tables below).

Table B.16: Following a PEO: AC+ΔPPC vs AC+σ-ΔPPC (table 1 of 2)

			# instances			CPU time (s)		
			AC+ΔPPC	**AC+σ-ΔPPC**	**Gain**	**AC+ΔPPC**	**AC+σ-ΔPPC**	**Saving**
	Total	**1,731**	**1,728**	**1,731**	**3**	**3,446.7**	**3,375.7**	**91.0**
	QCP-10	15	15	15	0	0.4	0.3	0.1
	QCP-15	15	15	15	0	2.0	1.4	0.6
	QCP-20	15	15	15	0	8.0	5.5	2.4
	QCP-25	15	15	15	0	22.7	15.7	7.0
	QWH-10	10	10	10	0	0.3	0.2	0.1
	QWH-15	10	10	10	0	1.5	1.0	0.4
	QWH-20	10	10	10	0	5.6	3.9	1.8
	QWH-25	10	10	10	0	15.3	10.3	5.0
	bqwh-15-106	100	100	100	0	0.3	0.3	0.1
	bqwh-18-141	100	100	100	0	0.7	0.5	0.2
	coloring	22	22	22	0	5.0	2.6	2.4
	driver	7	7	7	0	5.9	4.5	1.5
	geom	100	100	100	0	0.6	0.6	0.0
graphColoring	hos	14	14	14	0	23.9	19.0	4.9
graphColoring	full-insertion	40	37	40	3	74.4	55.8	18.6
graphColoring	k-insertion	32	32	32	0	13.4	8.5	4.9
graphColoring	leighton-15	26	26	26	0	176.9	161.3	15.6
graphColoring	leighton-25	31	31	31	0	385.1	372.3	12.8
graphColoring	leighton-5	8	8	8	0	37.4	23.5	13.9
graphColoring	mug	8	8	8	0	0.0	0.0	0.0
graphColoring	myciel	16	16	16	0	0.4	0.3	0.1
graphColoring	register-fpsol	37	37	37	0	80.4	79.8	0.6
graphColoring	register-inithx	31	31	31	0	84.4	83.4	1.0
graphColoring	register-mulsol	47	47	47	0	13.1	13.0	0.1
graphColoring	register-zeroin	31	31	31	0	16.4	16.3	0.1
graphColoring	school	8	8	8	0	105.2	99.1	6.1
graphColoring	sgb-book	26	26	26	0	0.2	0.2	0.0
graphColoring	sgb-games	4	4	4	0	0.4	0.3	0.1
graphColoring	sgb-miles	42	42	42	0	33.9	33.8	0.1
graphColoring	sgb-queen	50	50	50	0	21.0	19.8	1.2

Results continue in next table.

Table B.17: Following a PEO: AC+ΔPPC vs AC+σ-ΔPPC (table 2 of 2)

			# instances			CPU time (s)		
			AC+ΔPPC	**AC+σ-ΔPPC**	**Gain**	**AC+ΔPPC**	**AC+σ-ΔPPC**	**Saving**
	hanoi	5	5	5	0	2.0	2.0	0.0
	langford	4	4	4	0	1.3	1.2	0.1
	lard	10	10	10	0	58.9	58.7	0.2
	marc	10	10	10	0	25.2	25.0	0.2
	os-taillard-4	30	30	30	0	154.6	154.1	0.5
	os-taillard-5	30	30	30	0	1,256.9	1,262.8	-5.9
rlfap	Graphs	14	14	14	0	583.2	589.3	-6.1
rlfap	GraphsMod	12	12	12	0	106.0	103.3	2.7
rlfap	Scens11	12	12	12	0	30.2	30.3	-0.2
rlfap	Scens	11	11	11	0	26.7	27.0	-0.3
rlfap	ScensMod	13	13	13	0	9.1	8.9	0.1
	tightness0.1	100	100	100	0	0.2	0.2	0.0
	tightness0.2	100	100	100	0	0.2	0.2	0.0
	tightness0.35	100	100	100	0	0.4	0.4	0.0
	tightness0.5	100	100	100	0	0.8	0.8	0.0
	tightness0.65	100	100	100	0	1.5	1.6	0.0
	tightness0.8	100	100	100	0	6.8	6.8	0.0
	tightness0.9	100	100	100	0	68.0	69.9	-1.9

B.4 Comparing Types of Supports

We compare the performance AC+σ-ΔPPC^{sup2001}, which uses PC-2001 style supports and AC+σ-ΔPPC^{sup}, which users our new support structure.

Table B.18 isolates instances where PPC removes no tuples, thus, the effort of building support structures is wasted. In retrospect, one can create such structures only when the need arises.

Table B.18: Comparing the two styles of support structures when no filtering occurs

			CPU time (s)		
			AC+σ-ΔPPC^{sup2001}	**AC+σ-ΔPPC^{sup}**	**Saved**
	Total	**523**	**508.3**	**549.3**	**-41.1**
	coloring	22	3.6	3.7	-0.1
graphColoring	hos	14	26.2	26.7	-0.5
graphColoring	full-insertion	40	17.8	19.2	-1.3
graphColoring	k-insertion	32	12.2	12.8	-0.6
graphColoring	leighton-15	26	88.4	96.0	-7.6
graphColoring	leighton-25	31	56.8	60.9	-4.1
graphColoring	leighton-5	8	33.0	33.9	-0.9
graphColoring	mug	8	0.0	0.0	0.0
graphColoring	myciel	16	0.5	0.5	0.0
graphColoring	register-fpsol	37	32.2	34.2	-2.1
graphColoring	register-inithx	31	28.0	29.3	-1.3
graphColoring	register-mulsol	47	17.6	19.1	-1.5
graphColoring	register-zeroin	31	22.3	24.5	-2.2
graphColoring	school	8	115.9	128.4	-12.5
graphColoring	sgb-book	26	0.3	0.3	0.0
graphColoring	sgb-games	4	0.5	0.5	0.0
graphColoring	sgb-miles	42	22.8	25.6	-2.8
graphColoring	sgb-queen	50	26.0	29.4	-3.5
rand	23-Feb	10	0.6	0.6	0.0
rand	24-Feb	10	0.7	0.7	0.0
rand	25-Feb	10	0.8	0.9	0.0
rand	26-Feb	10	1.0	1.0	0.0
rand	27-Feb	10	1.2	1.3	0.0

Table B.19 reports the results when filtering occurs. Both algorithms complete all instances of the composed benchmark (omitted) with the same CPU time.

Table B.19: Comparing the two styles of support structures on benchmarks when filtering occurs

			CPU time (s)		
			AC+σ-ΔPPC^{sup2001}	**AC+σ-ΔPPC^{sup}**	**Saved**
	Total	**1,675**	**1,713.3**	**1,699.3**	**14.0**
	BH-4-13	7	95.7	100.2	-4.4
	BH-4-4	10	0.2	0.2	0.0
	BH-4-7	20	4.7	4.8	-0.1
	QCP-10	15	0.4	0.4	0.0
	QCP-15	15	2.1	2.0	0.1
	QCP-20	15	8.8	8.5	0.3
	QCP-25	15	26.9	26.0	0.9
	QWH-10	10	0.3	0.2	0.0
	QWH-15	10	1.4	1.3	0.1
	QWH-20	10	6.1	5.6	0.5
	QWH-25	10	17.5	16.4	1.0
	bqwh-15-106	100	0.3	0.2	0.1
	bqwh-18-141	100	0.6	0.6	0.1
	driver	7	5.0	3.6	1.4
	frb30-15	10	0.3	0.3	0.0
	frb35-17	10	0.5	0.5	0.0
	frb40-19	10	0.9	0.9	0.0
	frb45-21	10	1.4	1.5	0.0
	frb50-23	10	2.2	2.3	-0.1
	frb53-24	10	2.8	2.9	-0.1
	frb56-25	10	3.4	3.5	-0.2
	frb59-26	10	4.2	4.5	-0.3
	geom	100	0.7	0.7	0.0
	hanoi	5	2.0	2.0	0.0
	langford	4	1.4	1.3	0.1
	lard	10	81.2	91.0	-9.7
	marc	10	28.2	28.2	0.0
	os-taillard-4	30	127.7	129.8	-2.2
	os-taillard-5	30	1,056.1	1,059.1	-3.1
rand	2-30-15	50	0.3	0.3	0.0
	2-30-15-fcd	50	0.3	0.3	0.0
	2-40-19	50	0.9	0.9	0.0
	2-40-19-fcd	50	0.9	0.9	0.0
	2-50-23	50	2.2	2.3	-0.1
	2-50-23-fcd	50	2.2	2.3	-0.1
rlfap	Graphs	14	90.2	73.3	16.9
	GraphsMod	12	33.3	31.7	1.6
	Scens11	12	29.2	25.6	3.6
	Scens	11	26.0	23.4	2.5
	ScensMod	13	8.7	8.0	0.8
	tightness0.1	100	0.2	0.2	0.0
	tightness0.2	100	0.3	0.3	0.0
	tightness0.35	100	0.5	0.4	0.0
	tightness0.5	100	0.9	0.8	0.1
	tightness0.65	100	1.4	1.2	0.3
	tightness0.8	100	4.4	3.5	0.9
	tightness0.9	100	28.6	25.5	3.0

B.5 Benefits of Using Supports

Creating support structures uses up memory, but allows us, in general, to save on CPU time. When no filtering is possible, recording supports is wasteful of both time and space. We compare the performance of AC+σ-ΔPPC^{sup} and AC+σ-ΔPPC in Table B.20 (no filtering) and in Tables B.21 and B.22 (with filtering).

Table B.20: Drawbacks of using support structures on benchmarks with no filtering

			# instances			CPU time (s)		
			AC+σ-ΔPPC	**AC+σ-ΔPPC^{sup}**	**Gain**	**AC+σ-ΔPPC**	**AC+σ-ΔPPC^{sup}**	**Saved**
	Total	**523**	**523**	**436**	**-87**	**354.7**	**549.3**	**-194.6**
	coloring	22	22	22	0	2.6	3.7	-1.1
graphColoring	hos	14	14	14	0	19.0	26.7	-7.7
graphColoring	full-insertion	40	40	32	-8	12.2	19.2	-7.0
graphColoring	k-insertion	32	32	32	0	8.5	12.8	-4.4
graphColoring	leighton-15	26	26	10	-16	61.8	96.0	-34.2
graphColoring	leighton-25	31	31	6	-25	39.9	60.9	-21.0
graphColoring	leighton-5	8	8	8	0	23.5	33.9	-10.4
graphColoring	mug	8	8	8	0	0.0	0.0	0.0
graphColoring	myciel	16	16	16	0	0.3	0.5	-0.1
graphColoring	register-fpsol	37	37	26	-11	20.7	34.2	-13.6
graphColoring	register-inithx	31	31	13	-18	17.0	29.3	-12.3
graphColoring	register-mulsol	47	47	47	0	13.0	19.1	-6.1
graphColoring	register-zeroin	31	31	31	0	16.3	24.5	-8.2
graphColoring	school	8	8	5	-3	81.2	128.4	-47.2
graphColoring	sgb-book	26	26	26	0	0.2	0.3	-0.1
graphColoring	sgb-games	4	4	4	0	0.3	0.5	-0.2
graphColoring	sgb-miles	42	42	37	-5	16.6	25.6	-9.0
graphColoring	sgb-queen	50	50	49	-1	18.1	29.4	-11.4
rand	23-Feb	10	10	10	0	0.5	0.6	-0.1
rand	24-Feb	10	10	10	0	0.6	0.7	-0.1
rand	25-Feb	10	10	10	0	0.7	0.9	-0.1
rand	26-Feb	10	10	10	0	0.9	1.0	-0.2
rand	27-Feb	10	10	10	0	1.1	1.3	-0.2

Table B.21: Impact of using support structures on benchmarks with filtering (table 1 of 2)

			# instances			CPU time (s)		
			AC+σ-ΔPPC	AC+σ-ΔPPC^{sup}	Gain	AC+σ-ΔPPC	AC+σ-ΔPPC^{sup}	Saved
	Total	**1,765**	**1,765**	**1,757**	**-8**	**1,975.2**	**1,700.8**	**274.4**
	BH-4-13	7	7	7	0	79.4	100.2	-20.7
	BH-4-4	10	10	10	0	0.2	0.2	0.0
	BH-4-7	20	20	20	0	3.9	4.8	-0.9
	QCP-10	15	15	15	0	0.3	0.4	-0.1
	QCP-15	15	15	15	0	1.4	2.0	-0.6
	QCP-20	15	15	15	0	5.5	8.5	-2.9
	QCP-25	15	15	15	0	15.7	26.0	-10.3
	QWH-10	10	10	10	0	0.2	0.2	0.0
	QWH-15	10	10	10	0	1.0	1.3	-0.3
	QWH-20	10	10	10	0	3.9	5.6	-1.7
	QWH-25	10	10	10	0	10.3	16.4	-6.2
	bqwh-15-106	100	100	100	0	0.3	0.2	0.0
	bqwh-18-141	100	100	100	0	0.5	0.6	0.0
composed	25-1-2	10	10	10	0	0.1	0.0	0.0
	25-1-25	10	10	10	0	0.1	0.0	0.0
	25-1-40	10	10	10	0	0.1	0.1	0.0
	25-1-80	10	10	10	0	0.1	0.1	0.0
	25-10-20	10	10	10	0	0.4	0.4	0.0
	75-1-2	10	10	10	0	0.6	0.2	0.4
	75-1-25	10	10	10	0	0.6	0.2	0.4
	75-1-40	10	10	10	0	0.7	0.2	0.5
	75-1-80	10	10	10	0	0.7	0.3	0.4
	driver	7	7	7	0	4.5	3.6	0.9
	frb30-15	10	10	10	0	0.3	0.3	0.0
	frb35-17	10	10	10	0	0.5	0.5	0.0
	frb40-19	10	10	10	0	0.8	0.9	-0.1
	frb45-21	10	10	10	0	1.2	1.5	-0.2
	frb50-23	10	10	10	0	1.8	2.3	-0.5
	frb53-24	10	10	10	0	2.2	2.9	-0.6
	frb56-25	10	10	10	0	2.7	3.5	-0.9
	frb59-26	10	10	10	0	3.4	4.5	-1.1
	geom	100	100	100	0	0.6	0.7	-0.1
	hanoi	5	5	5	0	2.0	2.0	0.0
	langford	4	4	4	0	1.2	1.3	-0.1
	lard	10	10	10	0	58.7	91.0	-32.3
	marc	10	10	10	0	25.0	28.2	-3.2
	os-taillard-4	30	30	30	0	154.1	129.8	24.3
	os-taillard-5	30	30	30	0	1,262.8	1,059.1	203.7
rand	2-30-15	50	50	50	0	0.3	0.3	0.0
	2-30-15-fcd	50	50	50	0	0.3	0.3	0.0
	2-40-19	50	50	50	0	0.7	0.9	-0.1
	2-40-19-fcd	50	50	50	0	0.7	0.9	-0.1
	2-50-23	50	50	50	0	1.8	2.3	-0.5
	2-50-23-fcd	50	50	50	0	1.8	2.3	-0.5

Table B.22: Impact of using support structures on benchmarks with filtering (table 2 of 2)

			# instances			CPU time (s)		
			AC+σ-ΔPPC	**AC+σ-ΔPPC^{sup}**	**Gain**	**AC+σ-ΔPPC**	**AC+σ-ΔPPC^{sup}**	**Saved**
rlfap	Graphs	14	14	10	-4	141.1	73.3	67.8
rlfap	GraphsMod	12	12	8	-4	34.8	31.7	3.1
rlfap	Scens11	12	12	12	0	30.3	25.6	4.7
rlfap	Scens	11	11	11	0	27.0	23.4	3.6
rlfap	ScensMod	13	13	13	0	8.9	8.0	1.0
	tightness0.1	100	100	100	0	0.2	0.2	-0.1
	tightness0.2	100	100	100	0	0.2	0.3	-0.1
	tightness0.35	100	100	100	0	0.4	0.4	-0.1
	tightness0.5	100	100	100	0	0.8	0.8	0.0
	tightness0.65	100	100	100	0	1.6	1.2	0.4
	tightness0.8	100	100	100	0	6.8	3.5	3.3
	tightness0.9	100	100	100	0	69.9	25.5	44.4

B.6 Improving PC-8

All algorithms do the same filtering. AC+PC-8-Ordering has the smallest CPU time.

Table B.23: Assessing improvements to PC-8 (table 1 of 2)

			# instances				CPU time (s)			
			AC+PC-8	AC+PC-8+	AC+PC-8-Flag	AC+PC-8-Ordering	AC+PC-8	AC+PC-8+	AC+PC-8-Flag	AC+PC-8-Ordering
	Total:	**2,288**	**2,264**	**2,261**	**2,260**	**2,266**	**27,038**	**29,088**	**29,446**	**26,549**
	BH-4-13	7	*All instances complete*				536.2	573.8	586.1	524.8
	BH-4-4	10	*All instances complete*				0.5	0.5	0.5	0.5
	BH-4-7	20	*All instances complete*				20.9	22.2	22.7	20.4
	QCP-10	15	*All instances complete*				0.5	0.6	0.5	0.5
	QCP-15	15	*All instances complete*				3.5	3.8	3.8	3.5
	QCP-20	15	*All instances complete*				17.0	18.6	18.6	17.0
	QCP-25	15	*All instances complete*				57.8	63.6	63.3	58.1
	QWH-10	10	*All instances complete*				0.4	0.4	0.4	0.4
	QWH-15	10	*All instances complete*				2.7	2.9	3.0	2.7
	QWH-20	10	*All instances complete*				13.3	14.3	14.3	13.3
	QWH-25	10	*All instances complete*				44.4	48.7	48.6	44.6
	bqwh-15-106	100	*All instances complete*				1.0	1.1	1.1	1.0
	bqwh-18-141	100	*All instances complete*				2.0	2.1	2.1	1.9
	coloring	22	*All instances complete*				7.1	8.0	7.9	7.2
composed	25-1-2	10	*All instances complete*				0.2	0.2	0.2	0.2
composed	25-1-25	10	*All instances complete*				0.2	0.2	0.2	0.2
composed	25-1-40	10	*All instances complete*				0.2	0.2	0.2	0.2
composed	25-1-80	10	*All instances complete*				0.2	0.2	0.2	0.2
composed	25-10-20	10	*All instances complete*				12.9	13.2	13.6	12.7
composed	75-1-2	10	*All instances complete*				2.0	2.3	2.2	2.1
composed	75-1-25	10	*All instances complete*				2.0	2.3	2.2	2.1
composed	75-1-40	10	*All instances complete*				2.0	2.3	2.3	2.1
composed	75-1-80	10	*All instances complete*				2.1	2.3	2.3	2.1
	driver	7	*All instances complete*				94.6	81.2	87.3	80.7
	frb30-15	10	*All instances complete*				0.4	0.4	0.4	0.4
	frb35-17	10	*All instances complete*				0.7	0.7	0.7	0.7
	frb40-19	10	*All instances complete*				1.2	1.2	1.2	1.2
	frb45-21	10	*All instances complete*				2.0	2.1	2.1	2.0
	frb50-23	10	*All instances complete*				3.1	3.3	3.3	3.1
	frb53-24	10	*All instances complete*				3.9	4.2	4.2	3.9
	frb56-25	10	*All instances complete*				4.9	5.3	5.3	4.9
	frb59-26	10	*All instances complete*				6.1	6.6	6.6	6.1
	geom	100	*All instances complete*				2.2	2.4	2.4	2.2

Results continue in next table.

Table B.24: Assessing improvements to PC-8 (table 2 of 2)

			# instances				CPU time (s)			
			AC+PC-8	**AC+PC-8$^+$**	**AC+PC-8-FLAG**	**AC+PC-8-ORDERING**	**AC+PC-8**	**AC+PC-8$^+$**	**AC+PC-8-FLAG**	**AC+PC-8-ORDERING**
graphColoring	hos	14	14	13	13	14	2,786.7	3,121.7	3,107.0	2,801.8
	full-insertion	40	35	34	34	35	611.1	683.7	680.4	614.0
	k-insertion	32	*All instances complete*				304.8	341.6	340.3	306.8
	leighton-15	26	*All instances complete*				409.0	456.9	454.9	411.0
	leighton-25	31	*All instances complete*				1,114.6	1,248.9	1,242.7	1,120.3
	leighton-5	8	*All instances complete*				62.6	70.0	69.6	62.9
	mug	8	*All instances complete*				0.4	0.4	0.4	0.4
	myciel	16	*All instances complete*				3.1	3.5	3.5	3.2
	register-fpsol	37	*All instances complete*				1,109.1	1,238.3	1,233.0	1,114.6
	register-inithx	31	26	26	26	26	3,464.7	3,873.7	3,857.4	3,481.8
	register-mulsol	47	*All instances complete*				105.1	116.6	116.1	105.6
	register-zeroin	31	*All instances complete*				80.2	88.8	88.5	80.6
	school	8	*All instances complete*				322.6	359.6	358.3	324.3
	sgb-book	26	*All instances complete*				129.0	144.3	143.6	129.6
	sgb-games	4	*All instances complete*				3.3	3.7	3.7	3.4
	sgb-miles	42	*All instances complete*				101.3	111.9	111.4	101.7
	sgb-queen	50	*All instances complete*				34.1	38.0	37.8	34.2
	hanoi	5	*All instances complete*				3.7	3.7	3.7	3.7
	langford	4	*All instances complete*				1.8	1.6	1.8	1.6
	lard	10	*All instances complete*				67.7	72.7	72.4	67.9
	marc	10	*All instances complete*				25.0	25.0	25.0	25.0
	os-taillard-4	30	*All instances complete*				205.9	199.6	207.8	196.3
	os-taillard-5	30	*All instances complete*				1,563.0	1,498.5	1,575.8	1,475.5
rand	23-Feb	10	*All instances complete*				0.5	0.6	0.6	0.5
	24-Feb	10	*All instances complete*				0.6	0.7	0.7	0.7
	25-Feb	10	*All instances complete*				0.8	0.8	0.8	0.8
	26-Feb	10	*All instances complete*				0.9	1.0	1.0	0.9
	27-Feb	10	*All instances complete*				1.1	1.2	1.2	1.1
	2-30-15	50	*All instances complete*				0.3	0.3	0.4	0.3
	2-30-15-fcd	50	*All instances complete*				0.3	0.3	0.4	0.3
	2-40-19	50	*All instances complete*				1.1	1.2	1.2	1.1
	2-40-19-fcd	50	*All instances complete*				1.1	1.2	1.2	1.1
	2-50-23	50	*All instances complete*				3.0	3.2	3.2	3.0
	2-50-23-fcd	50	*All instances complete*				3.0	3.2	3.2	3.0
rlfap	Graphs	14	9	9	9	9	722.8	744.4	770.4	690.0
	GraphsMod	12	10	9	9	9	3,270.0	3,534.3	3,532.0	3,228.2
	Scens11	12	9	9	8	10	7,284.0	7,823.9	7,967.8	7,102.3
	Scens	11	9	9	9	9	846.7	833.1	891.8	775.5
	ScensMod	13	11	11	11	13	963.3	1,009.6	1,045.6	920.5
	tightness0.1	100	*All instances complete*				0.2	0.2	0.2	0.2
	tightness0.2	100	*All instances complete*				0.4	0.4	0.4	0.4
	tightness0.35	100	*All instances complete*				0.7	0.8	0.8	0.7
	tightness0.5	100	*All instances complete*				1.9	2.0	2.0	1.9
	tightness0.65	100	*All instances complete*				5.9	5.4	5.7	5.3
	tightness0.8	100	*All instances complete*				42.1	34.1	38.9	35.1
	tightness0.9	100	*All instances complete*				529.9	492.7	531.6	487.0

B.7 Improving PC-2001

All algorithms do the same filtering. They all complete the same number of instances (i.e., 2,128). AC+PC-2001-Ordering has the smallest CPU time.

Table B.25: Assessing improvements to PC-2001 (table 1 of 2)

			# instances	CPU time (s) AC+PC-2001	AC+PC-2001$^+$	AC+PC-2001-Flag	AC+PC-2001-Ordering
	Total	**2,288**	**2,128**	**5,372**	**5,633**	**5,697**	**5,302**
	BH-4-13	7	7	561.4	592.0	597.0	556.4
	BH-4-4	10	10	0.6	0.7	0.7	0.6
	BH-4-7	20	20	22.4	23.8	24.4	22.1
	QCP-10	15	15	0.6	0.6	0.6	0.6
	QCP-15	15	15	4.4	4.7	4.7	4.4
	QCP-20	15	15	22.1	23.7	23.7	22.0
	QCP-25	15	15	78.0	83.8	83.9	78.1
	QWH-10	10	10	0.5	0.5	0.5	0.4
	QWH-15	10	10	3.4	3.6	3.6	3.4
	QWH-20	10	10	17.4	18.4	18.5	17.3
	QWH-25	10	10	60.4	64.5	64.7	60.3
	bqwh-15-106	100	100	1.0	1.0	1.1	1.0
	bqwh-18-141	100	100	2.0	2.1	2.1	1.9
	coloring	22	22	7.6	8.4	8.4	7.6
composed	25-1-2	10	10	0.2	0.2	0.2	0.2
composed	25-1-25	10	10	0.2	0.2	0.2	0.2
composed	25-1-40	10	10	0.2	0.2	0.2	0.2
composed	25-1-80	10	10	0.2	0.2	0.2	0.2
composed	25-10-20	10	10	11.1	11.1	11.9	10.9
composed	75-1-2	10	10	2.2	2.4	2.4	2.2
composed	75-1-25	10	10	2.2	2.4	2.4	2.2
composed	75-1-40	10	10	2.2	2.4	2.4	2.2
composed	75-1-80	10	10	2.2	2.4	2.4	2.2
	driver	7	7	91.0	79.7	86.8	78.9
	frb30-15	10	10	0.4	0.4	0.4	0.4
	frb35-17	10	10	0.7	0.8	0.8	0.7
	frb40-19	10	10	1.3	1.3	1.3	1.3
	frb45-21	10	10	2.1	2.3	2.3	2.1
	frb50-23	10	10	3.3	3.6	3.6	3.3
	frb53-24	10	10	4.2	4.5	4.6	4.2
	frb56-25	10	10	5.3	5.7	5.7	5.3
	frb59-26	10	10	6.6	7.1	7.1	6.6
	geom	100	100	2.4	2.6	2.6	2.4

Results continue in next table.

Assessing improvements to PC-2001 (table 2 of 2)

				CPU time (s)			
			# instances	AC+PC-2001	AC+PC-2001^{+}	AC+PC-2001-Flag	AC+PC-2001-Ordering
graphColoring	hos	14	4	249.2	277.3	277.4	249.3
	full-insertion	40	25	67.5	75.1	75.1	67.3
	k-insertion	32	28	68.4	75.9	76.0	68.4
	leighton-15	26	14	248.5	276.4	276.5	248.5
	leighton-25	31	8	200.1	222.7	222.8	200.1
	leighton-5	8	8	66.1	73.5	73.5	66.1
	mug	8	8	0.4	0.5	0.5	0.4
	myciel	16	16	3.4	3.7	3.7	3.4
	register-fpsol	37	11	198.7	220.3	220.2	198.6
	register-inithx	31	3	144.3	160.4	160.2	144.4
	register-mulsol	47	47	112.1	123.6	123.7	112.1
	register-zeroin	31	31	85.7	94.3	94.3	85.7
	school	8	8	340.6	377.7	377.7	340.6
	sgb-book	26	22	25.3	28.1	28.1	25.3
	sgb-games	4	4	3.6	4.0	4.0	3.6
	sgb-miles	42	42	108.4	119.0	119.0	108.4
	sgb-queen	50	50	36.2	40.1	40.1	36.2
	hanoi	5	5	5.5	5.5	5.5	5.4
	langford	4	4	1.7	1.6	1.8	1.6
	lard	10	10	83.0	88.0	87.9	83.0
	marc	10	10	37.8	37.9	37.9	37.8
	os-taillard-4	30	30	153.5	155.1	156.1	153.0
	os-taillard-5	30	30	1,236.7	1,248.0	1,253.5	1,229.6
rand	23-Feb	10	10	0.6	0.6	0.6	0.6
	24-Feb	10	10	0.7	0.7	0.7	0.7
	25-Feb	10	10	0.8	0.9	0.9	0.8
	26-Feb	10	10	1.0	1.0	1.0	1.0
	27-Feb	10	10	1.2	1.2	1.2	1.2
	2-30-15	50	50	0.4	0.4	0.4	0.4
	2-30-15-fcd	50	50	0.4	0.4	0.4	0.4
	2-40-19	50	50	1.2	1.3	1.3	1.2
	2-40-19-fcd	50	50	1.2	1.3	1.3	1.2
	2-50-23	50	50	3.2	3.5	3.5	3.2
	2-50-23-fcd	50	50	3.2	3.5	3.5	3.2
rlfap	Graphs	14	8	272.7	271.5	278.4	263.1
	GraphsMod	12	2	0.1	0.1	0.1	0.1
	Scens11	12	0	-	-	-	-
	Scens	11	7	167.4	168.6	174.0	160.5
	ScensMod	13	7	141.1	139.7	150.0	132.5
	tightness0.1	100	100	0.2	0.2	0.2	0.2
	tightness0.2	100	100	0.4	0.4	0.4	0.4
	tightness0.35	100	100	0.8	0.9	0.9	0.8
	tightness0.5	100	100	2.1	2.2	2.2	2.0
	tightness0.65	100	100	5.9	5.8	6.0	5.5
	tightness0.8	100	100	35.2	31.7	34.2	31.4
	tightness0.9	100	100	335.4	331.4	349.0	320.1

B.8 Comparing Main Algorithms

Table B.26: Comparing AC+DPC, AC+PC-2001-ORDERING, AC+PC-8-ORDERING, σ-ΔPPC^{sup}, AC+sDC2, and sCDC1 (table 1 of 2)

			# instances						CPU time (s)					
			AC+DPC	**AC+PC-2001-ORDERING**	**AC+PC-8-ORDERING**	**AC+σ-ΔPPC^{sup}**	**AC+sDC2**	**sCDC1**	**AC+DPC**	**AC+PC-2001-ORDERING**	**AC+PC-8-ORDERING**	**AC+σ-ΔPPC^{sup}**	**AC+sDC2**	**sCDC1**
	Total:	**2,288**	**2,288**	**2,128**	**2,266**	**2,193**	**2,255**	**2,288**	**1,340**	**5,089**	**5,446**	**2,036**	**4,408**	**1,840**
	BH-4-13	7	*All instances complete*						37.1	556.4	524.8	100.2	653.7	182.3
	BH-4-4	10	*All instances complete*						0.2	0.6	0.5	0.2	0.6	0.2
	BH-4-7	20	*All instances complete*						2.3	22.1	20.4	4.8	27.1	8.1
	QCP-10	15	*All instances complete*						0.2	0.6	0.5	0.4	0.3	0.1
	QCP-15	15	*All instances complete*						0.8	4.4	3.5	2.0	2.9	0.5
	QCP-20	15	*All instances complete*						2.7	22.0	17.0	8.5	18.1	1.5
	QCP-25	15	*All instances complete*						7.6	78.1	58.1	26.0	61.8	3.5
	QWH-10	10	*All instances complete*						0.1	0.4	0.4	0.2	1.1	0.1
	QWH-15	10	*All instances complete*						0.5	3.4	2.7	1.3	8.5	0.4
	QWH-20	10	*All instances complete*						1.9	17.3	13.3	5.6	43.6	1.3
	QWH-25	10	*All instances complete*						5.2	60.3	44.6	16.4	93.7	2.9
	bqwh-15-106	100	*All instances complete*						0.1	1.0	1.0	0.2	2.8	0.1
	bqwh-18-141	100	*All instances complete*						0.2	1.9	1.9	0.6	6.2	0.2
	coloring	22	*All instances complete*						1.0	7.6	7.2	3.7	2.4	0.2
composed	25-1-2	10	*All instances complete*						0.0	0.2	0.2	0.0	0.2	0.1
composed	25-1-25	10	*All instances complete*						0.0	0.2	0.2	0.0	0.2	0.1
composed	25-1-40	10	*All instances complete*						0.1	0.2	0.2	0.1	0.2	0.1
composed	25-1-80	10	*All instances complete*						0.1	0.2	0.2	0.1	0.2	0.1
composed	25-10-20	10	*All instances complete*						0.2	10.9	12.7	0.4	9.0	0.7
composed	75-1-2	10	*All instances complete*						0.2	2.2	2.1	0.2	1.2	0.3
composed	75-1-25	10	*All instances complete*						0.2	2.2	2.1	0.2	1.3	0.3
composed	75-1-40	10	*All instances complete*						0.2	2.2	2.1	0.2	1.3	0.3
composed	75-1-80	10	*All instances complete*						0.3	2.2	2.1	0.3	1.3	0.3
	driver	7	*All instances complete*						1.1	78.9	80.7	3.6	49.5	5.6
	frb30-15	10	*All instances complete*						0.2	0.4	0.4	0.3	0.8	0.5
	frb35-17	10	*All instances complete*						0.3	0.7	0.7	0.5	1.4	0.8
	frb40-19	10	*All instances complete*						0.5	1.3	1.2	0.9	2.3	1.2
	frb45-21	10	*All instances complete*						0.7	2.1	2.0	1.5	3.6	1.8
	frb50-23	10	*All instances complete*						1.1	3.3	3.1	2.3	5.1	2.5
	frb53-24	10	*All instances complete*						1.4	4.2	3.9	2.9	6.3	3.0
	frb56-25	10	*All instances complete*						1.7	5.3	4.9	3.5	7.4	3.5
	frb59-26	10	*All instances complete*						2.0	6.6	6.1	4.5	9.2	4.1
	geom	100	*All instances complete*						0.5	2.4	2.2	0.7	2.5	1.1

Table B.27: Comparing AC+DPC, AC+PC-2001-Ordering, AC+PC-8-Ordering, σ-ΔPPC^{sup}, AC+sDC2, and sCDC1 (table 2 of 2)

			# instances						**CPU time (s)**					
			AC+DPC	**AC+PC-2001-Ordering**	**AC+PC-8-Ordering**	**AC+σ-ΔPPC^{sup}**	**AC+sDC2**	**sCDC1**	**AC+DPC**	**AC+PC-2001-Ordering**	**AC+PC-8-Ordering**	**AC+σ-ΔPPC^{sup}**	**AC+sDC2**	**sCDC1**
graphColoring	hos	14	14	4	14	14	14	14	1.8	249.3	237.5	5.9	32.7	0.9
	full-insertion	40	40	25	35	32	37	40	1.3	67.3	64.3	4.6	12.0	0.7
	k-insertion	32	32	28	32	32	32	32	1.1	68.4	65.1	4.1	9.9	0.4
	leighton-15	26	26	14	26	10	26	26	22.7	178.2	169.7	96.0	66.6	8.5
	leighton-25	31	31	8	31	6	31	31	14.8	160.4	152.6	60.9	56.6	7.4
	leighton-5	8	*All instances complete*						8.7	66.1	62.9	33.9	24.7	2.0
	mug	8	*All instances complete*						0.0	0.4	0.4	0.0	0.1	0.0
	myciel	16	*All instances complete*						0.2	3.4	3.2	0.5	1.7	0.3
	register-fpsol	37	37	11	37	26	37	37	6.5	198.6	188.7	17.1	92.0	37.2
	register-inithx	31	31	3	26	13	24	31	1.4	144.4	137.5	2.8	65.4	18.5
	register-mulsol	47	*All instances complete*						8.5	112.1	105.6	19.1	55.4	23.0
	register-zeroin	31	*All instances complete*						10.2	85.7	80.6	24.5	50.8	27.9
	school	8	8	8	8	5	8	8	32.1	283.4	270.0	128.4	161.1	44.6
	sgb-book	26	26	22	26	26	26	26	0.1	25.3	24.0	0.1	2.5	0.2
	sgb-games	4	*All instances complete*						0.2	3.6	3.4	0.5	1.3	0.1
	sgb-miles	42	42	42	42	37	42	42	9.1	66.4	62.2	25.6	41.4	23.0
	sgb-queen	50	50	50	50	49	50	50	7.0	33.0	31.2	29.4	15.6	3.5
	hanoi	5	*All instances complete*						2.0	5.4	3.7	2.0	2.0	2.0
	langford	4	*All instances complete*						0.7	1.6	1.6	1.3	1.8	2.1
	lard	10	*All instances complete*						35.9	83.0	67.9	91.0	34.1	32.3
	marc	10	*All instances complete*						24.2	37.8	25.0	28.2	23.7	23.7
	os-taillard-4	30	*All instances complete*						109.2	153.0	196.3	129.8	135.2	118.1
	os-taillard-5	30	*All instances complete*						924.7	1,229.6	1,475.5	1,059.1	1,187.3	1,012.1
rand	23-Feb	10	*All instances complete*						0.4	0.6	0.5	0.6	0.9	0.9
	24-Feb	10	*All instances complete*						0.5	0.7	0.7	0.7	1.1	1.1
	25-Feb	10	*All instances complete*						0.6	0.8	0.8	0.9	1.3	1.3
	26-Feb	10	*All instances complete*						0.7	1.0	0.9	1.0	1.6	1.6
	27-Feb	10	*All instances complete*						0.8	1.2	1.1	1.3	1.9	1.9
	2-30-15	50	*All instances complete*						0.1	0.4	0.3	0.3	0.7	0.5
	2-30-15-fcd	50	*All instances complete*						0.1	0.4	0.3	0.3	0.8	0.5
	2-40-19	50	*All instances complete*						0.4	1.2	1.1	0.9	2.2	1.2
	2-40-19-fcd	50	*All instances complete*						0.4	1.2	1.1	0.9	2.2	1.2
	2-50-23	50	*All instances complete*						1.1	3.2	3.0	2.3	4.9	2.6
	2-50-23-fcd	50	*All instances complete*						1.1	3.2	3.0	2.3	4.9	2.6
rlfap	Graphs	14	14	8	9	10	9	14	12.6	263.1	367.9	19.0	375.6	34.9
	GraphsMod	12	12	2	9	8	8	12	0.1	0.1	0.1	0.1	1.0	1.0
	Scens11	12	12	0	10	12	1	12	-	-	-	-	-	-
	Scens	11	11	7	9	11	9	11	13.9	160.5	141.4	14.6	78.5	16.0
	ScensMod	13	13	7	13	13	12	13	1.7	132.5	142.1	2.2	100.8	4.8
	tightness0.1	100	*All instances complete*						0.1	0.2	0.2	0.2	0.3	0.4
	tightness0.2	100	*All instances complete*						0.1	0.4	0.4	0.3	0.6	0.4
	tightness0.35	100	*All instances complete*						0.2	0.8	0.7	0.4	1.4	0.6
	tightness0.5	100	*All instances complete*						0.4	2.0	1.9	0.8	3.6	1.1
	tightness0.65	100	*All instances complete*						0.5	5.5	5.3	1.2	10.4	2.2
	tightness0.8	100	*All instances complete*						1.6	31.4	35.1	3.5	69.3	12.6
	tightness0.9	100	*All instances complete*						9.6	320.1	487.0	25.5	644.9	134.3

Appendix C

Results of Randomly Generated Problems

In this appendix, we include the results of our experiments on three randomly generated data sets:

- $\langle 50, 50, t, 20\% \rangle$
- $\langle 50, 25, t, 40\% \rangle$
- $\langle 50, 50, t, 40\% \rangle$

A summary and detailed description of the $\langle 50, 25, t, 20\% \rangle$ data set can be found in Section 5.2.

C.1 $\langle 50, 25, t, 40\% \rangle$

The complete results are summarized in Table C.1

Table C.1: Comparing CPU times on random problems $\langle 50, 25, t, 40\% \rangle$

Tightness	**AC+DPC**	**AC+σ-ΔPPC^{sup}**	**sCDC1**	**AC+PC-2001-**Ordering	**AC+PC-2001^{+}**	**AC+PC-2001**	**AC+σ-ΔPPC^{sup2001}**	**AC+PC-2001-**Flag	AC+PC-8^{+}	**AC+PC-8-**Ordering	**AC+PC-8-**Flag	**AC+PC-8**	**AC+sDC2**	**AC+σ-ΔPPC**	**AC+ΔPPC**	**AC+PPC+AP**	**AC+PC-2**
46.0%	3.6	11.5	17.3	14.5	15.6	15.0	13.8	15.8	14.8	13.7	15.3	14.2	35.4	13.2	13.2	23.2	16.5
48.0%	3.6	11.8	20.7	15.3	16.0	15.6	15.5	16.4	15.4	14.5	16.0	15.4	38.5	15.5	15.6	25.7	19.2
50.0%	3.6	12.3	25.9	16.1	17.0	17.3	17.8	17.8	16.8	16.2	18.1	18.1	49.8	19.3	19.1	29.2	24.2
52.0%	3.6	13.5	35.3	18.3	18.9	20.3	22.7	20.5	19.9	19.7	22.4	23.1	70.6	27.5	26.8	38.2	34.9
53.0%	3.7	19.6	73.5	27.3	27.7	29.9	49.3	29.8	34.6	35.4	38.1	39.8	157.9	72.8	71.5	94.0	97.0
53.5%	3.7	25.0	34.5	52.8	53.2	55.6	56.2	56.6	79.2	80.4	83.2	84.9	85.8	86.2	88.0	100.5	139.5
54.0%	3.7	19.0	17.0	45.3	46.0	48.6	30.4	49.4	65.6	66.3	70.0	71.4	41.0	41.3	43.8	46.0	80.3
56.0%	3.8	11.9	3.9	25.0	26.1	25.8	12.2	27.0	30.6	29.5	31.9	31.7	9.3	12.8	14.8	9.3	31.5
58.0%	3.9	8.9	1.6	11.4	12.2	11.4	9.3	12.3	11.2	10.4	11.2	10.4	3.7	6.7	7.8	5.4	10.0
60.0%	4.1	3.9	0.9	5.8	6.1	5.8	4.0	6.1	5.1	4.9	5.1	4.9	2.1	3.3	4.0	3.5	4.7

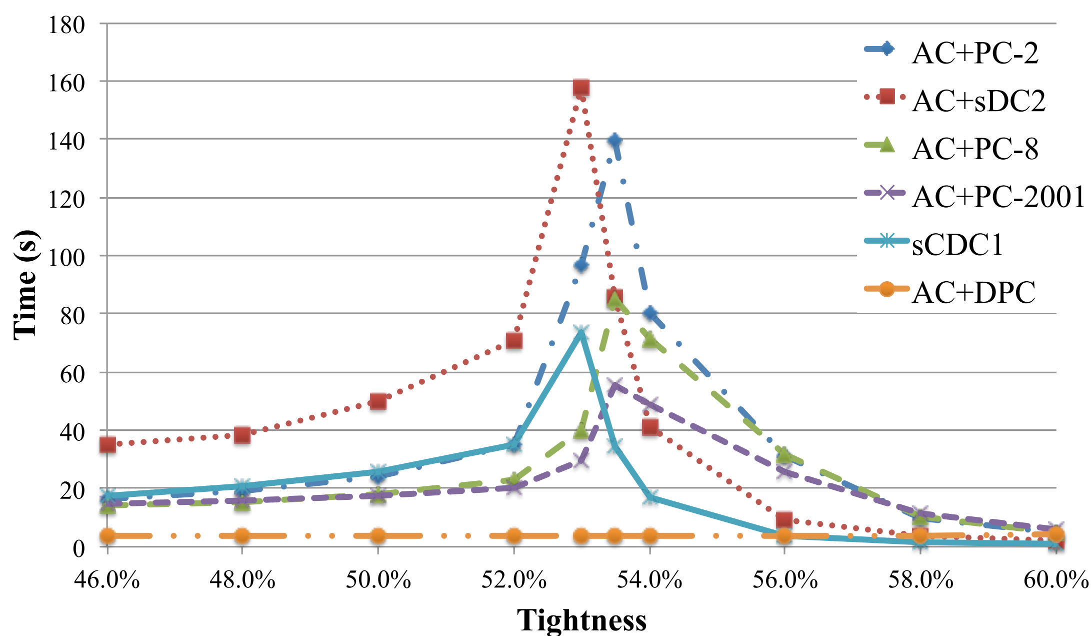


Figure C.1: $\langle 50, 25, t, 40\% \rangle$: Comparing the best previously known algorithms

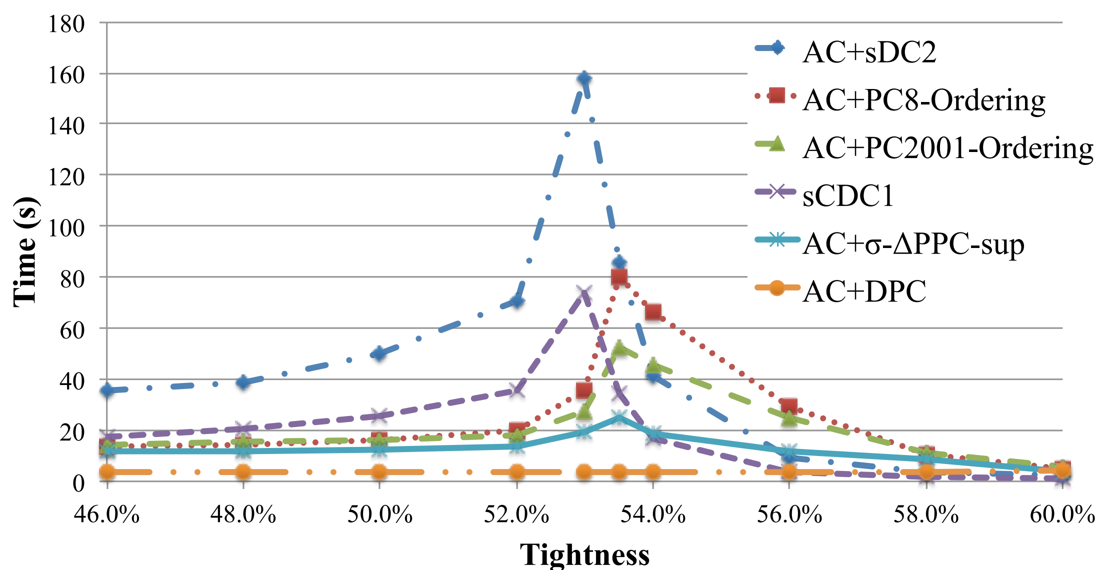


Figure C.2: $\langle 50, 25, t, 40\% \rangle$: Comparing the most competitive algorithms

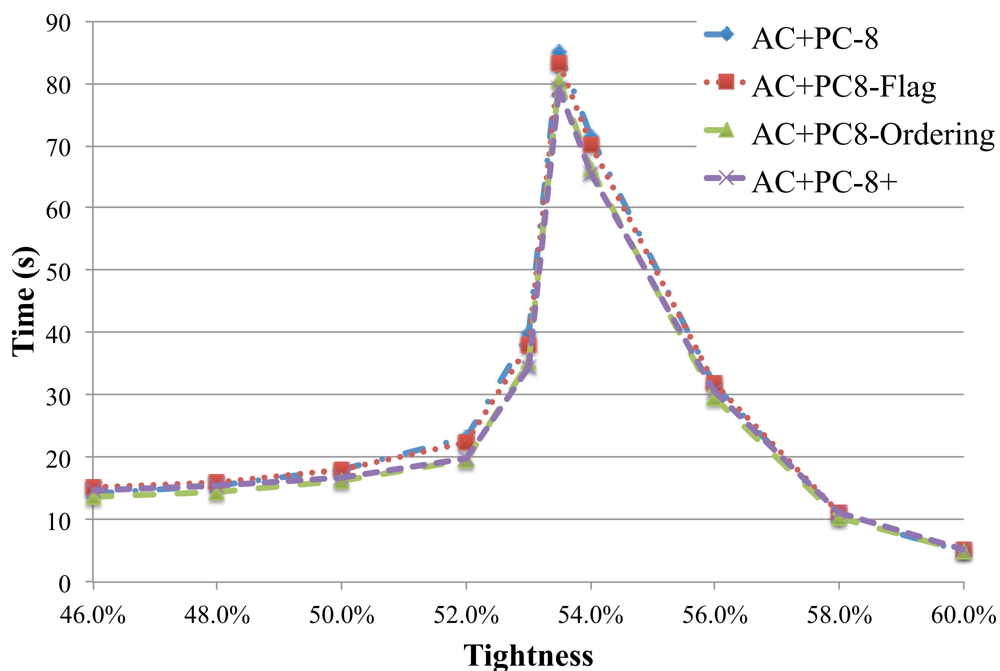


Figure C.3: $\langle 50, 25, t, 40\% \rangle$: Improving PC-8

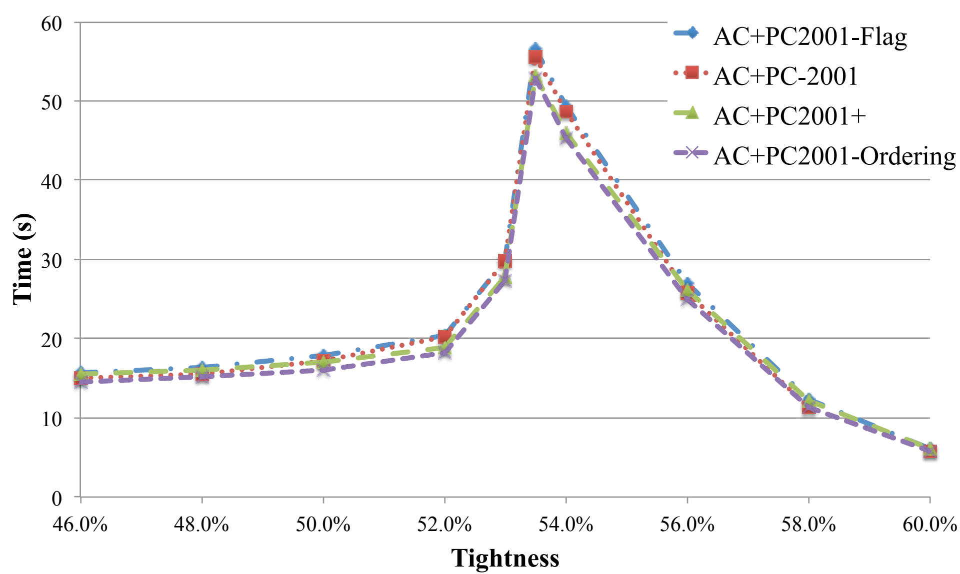


Figure C.4: $\langle 50, 25, t, 40\% \rangle$: Improving PC-2001

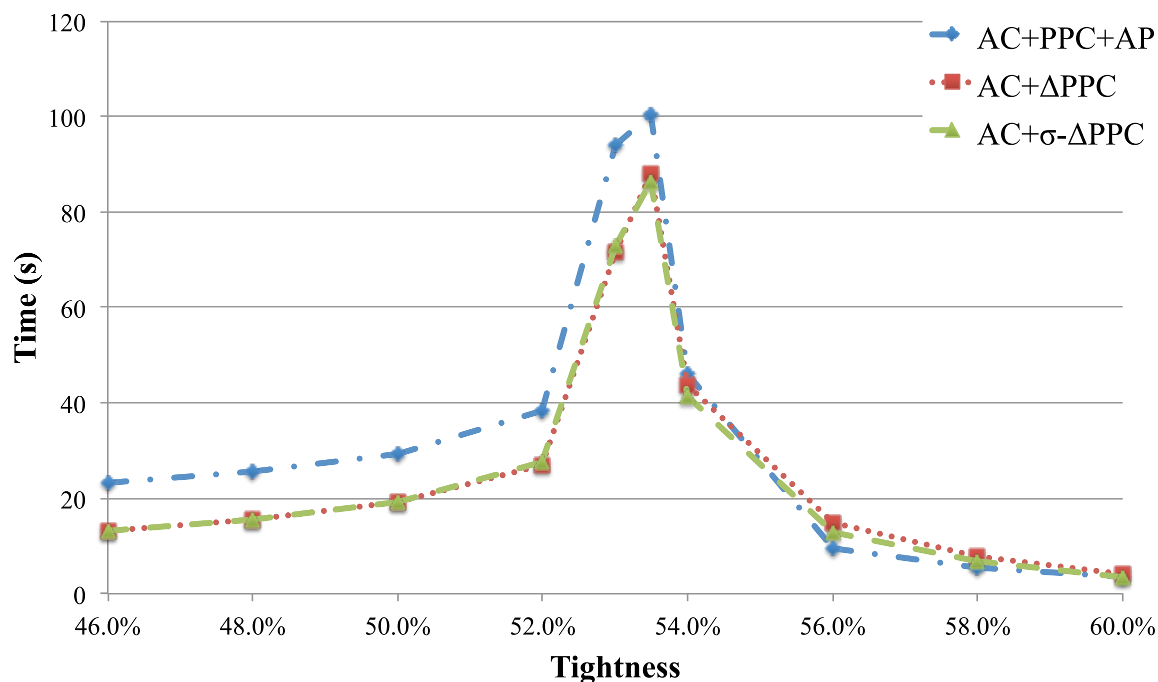


Figure C.5: $\langle 50, 25, t, 40\% \rangle$: Impact of the propagation queue and PEO

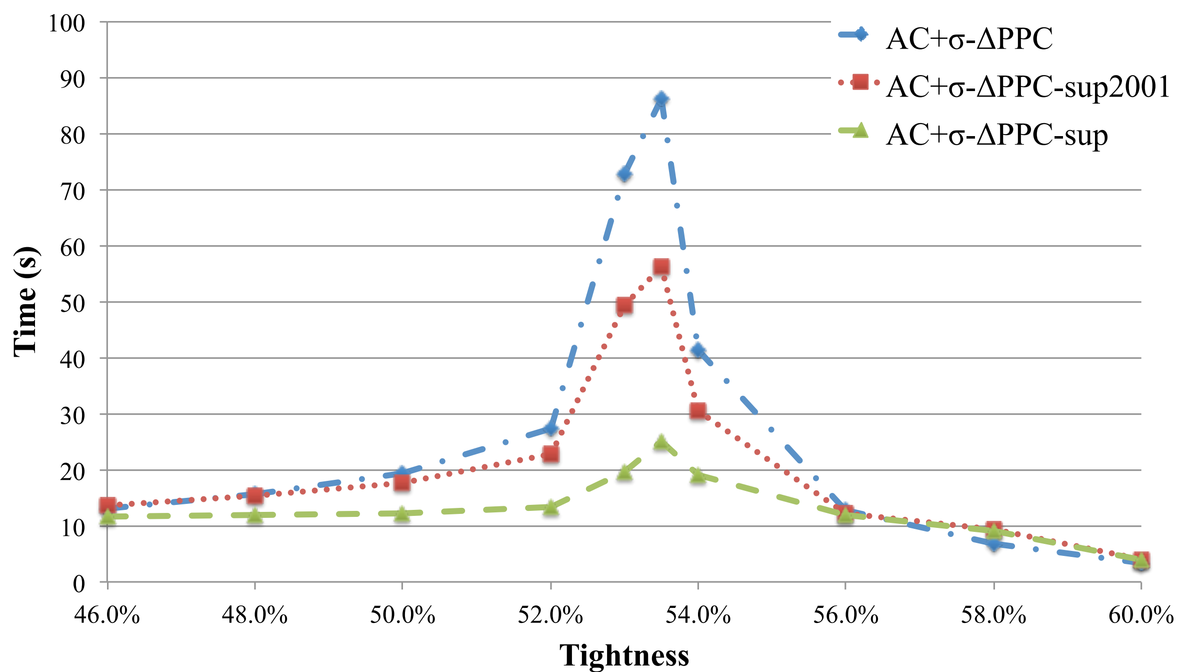


Figure C.6: $\langle 50, 25, t, 40\% \rangle$: Impact of support structures

C.2 $\langle 50, 50, t, 20\% \rangle$

The complete results are summarized in Table C.2

Table C.2: Comparing CPU times on random problems $\langle 50, 50, t, 20\% \rangle$

Tightness	**AC+DPC**	**AC+σ-ΔPPC^{sup}**	**sCDC1**	**AC+σ-ΔPPC^{sup2001}**	**AC+PC-2001**$^{+}$	**AC+PC-2001-**ORDERING	**AC+PC-2001-**FLAG	**AC+PC-2001**	**AC+ΔPPC**	**AC+σ-ΔPPC**	**AC+PPC+AP**	AC+PC-8^{+}	**AC+PC-8-**ORDERING	**AC+PC-8-**FLAG	**AC+PC-8**	**AC+sDC2**	**AC+PC-2**
66.0%	9.3	31.2	39.4	43.5	71.2	69.5	75.1	75.8	47.9	49.0	68.5	70.6	69.9	76.5	80.8	151.0	101.8
68.0%	9.6	33.6	51.9	51.9	80.5	79.9	87.4	91.0	63.4	65.0	88.0	83.5	86.4	96.0	105.7	205.9	137.2
70.0%	9.8	40.8	93.6	81.2	103.6	104.7	114.5	120.8	117.9	121.5	154.6	121.4	130.2	142.4	158.6	357.4	266.4
71.0%	10.1	69.4	104.5	156.4	241.5	242.5	251.6	259.6	264.8	278.2	300.6	365.8	387.3	388.9	416.8	420.1	700.8
72.0%	10.3	49.8	25.3	60.6	184.9	184.4	201.8	199.6	86.0	81.6	82.9	265.4	268.5	290.0	296.6	108.4	263.4
74.0%	11.0	33.2	7.8	33.3	109.2	103.1	98.4	93.4	33.3	31.3	19.5	121.7	115.8	106.1	102.6	34.4	102.6
76.0%	11.8	20.2	2.5	20.8	49.6	46.5	49.6	46.6	20.1	22.2	13.2	45.1	42.1	45.0	42.0	12.3	38.2
78.0%	12.4	8.8	1.4	9.1	22.1	21.1	22.1	21.1	9.6	10.1	9.4	17.9	16.9	17.8	16.9	7.9	15.8

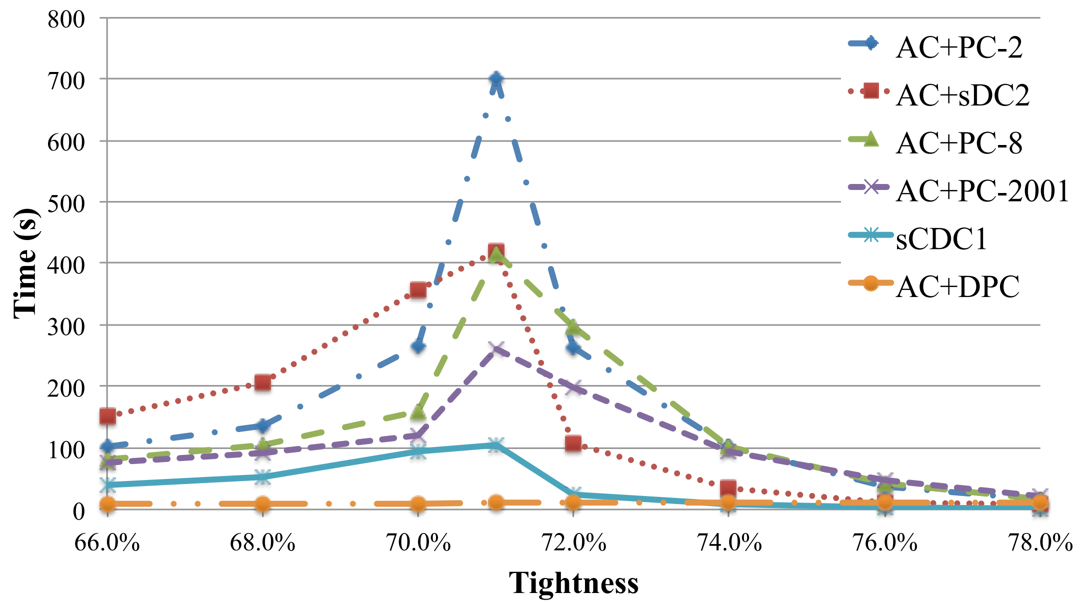


Figure C.7: $\langle 50, 50, t, 20\% \rangle$: Comparing the best previously known algorithms

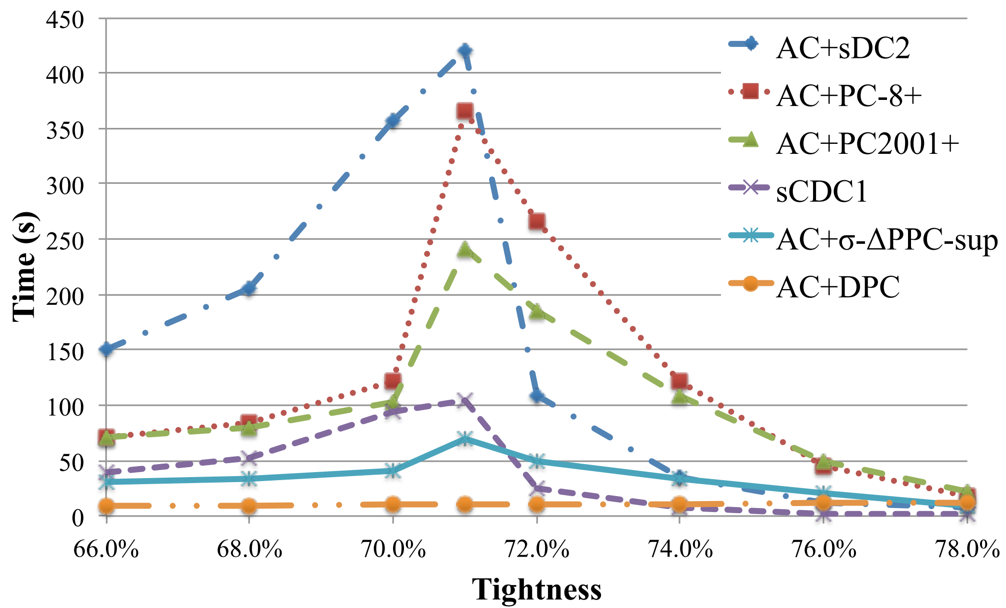


Figure C.8: $\langle 50, 50, t, 20\% \rangle$: Comparing the most competitive algorithms

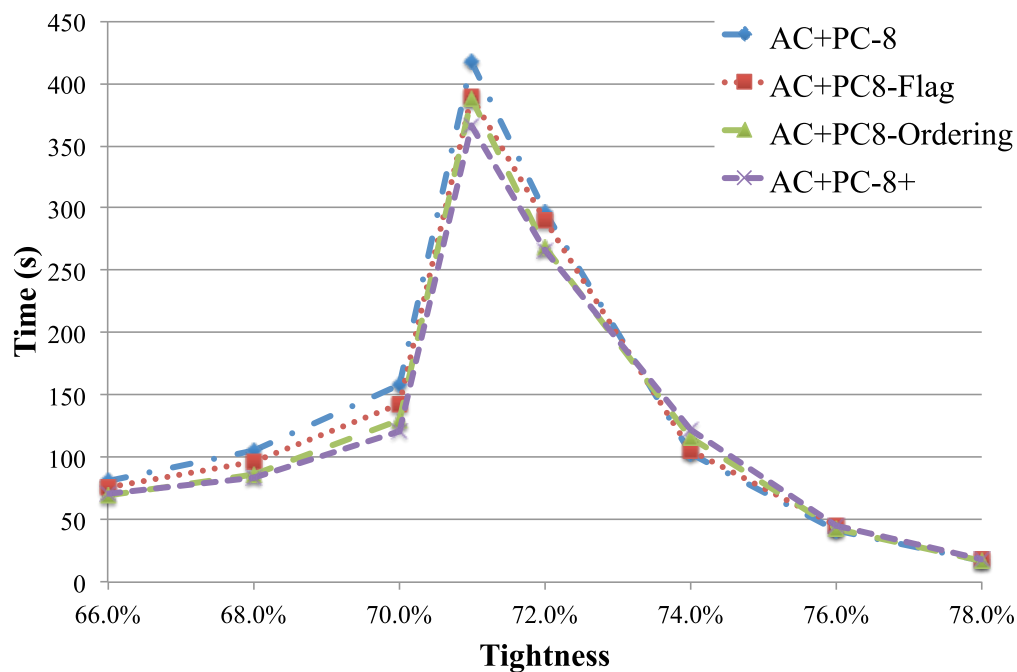


Figure C.9: $\langle 50, 50, t, 20\% \rangle$: Improving PC-8

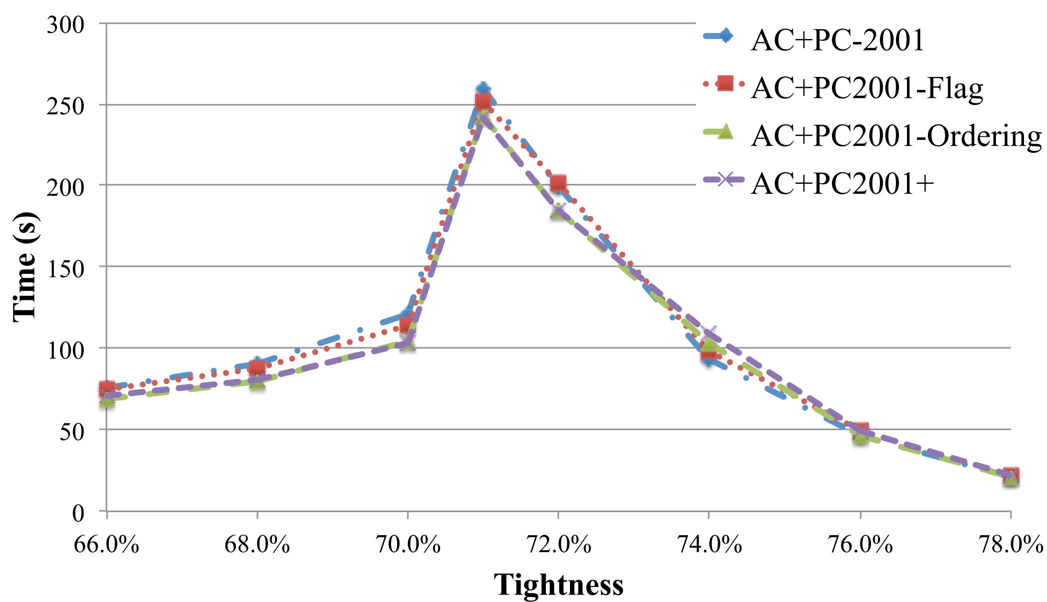


Figure C.10: $\langle 50, 50, t, 20\% \rangle$: Improving PC-2001

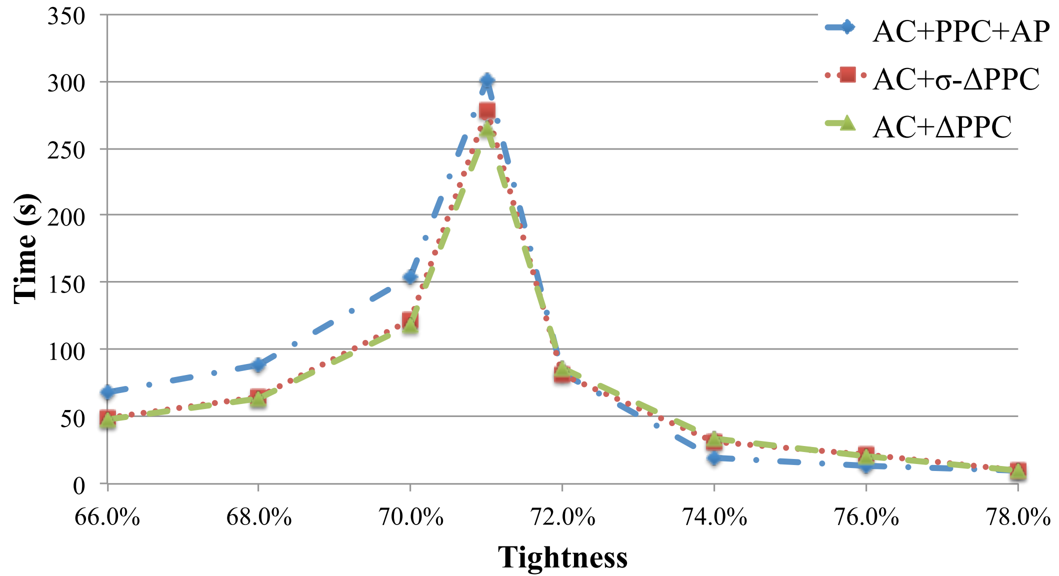


Figure C.11: $\langle 50, 50, t, 20\% \rangle$: Impact of the propagation queue and PEO

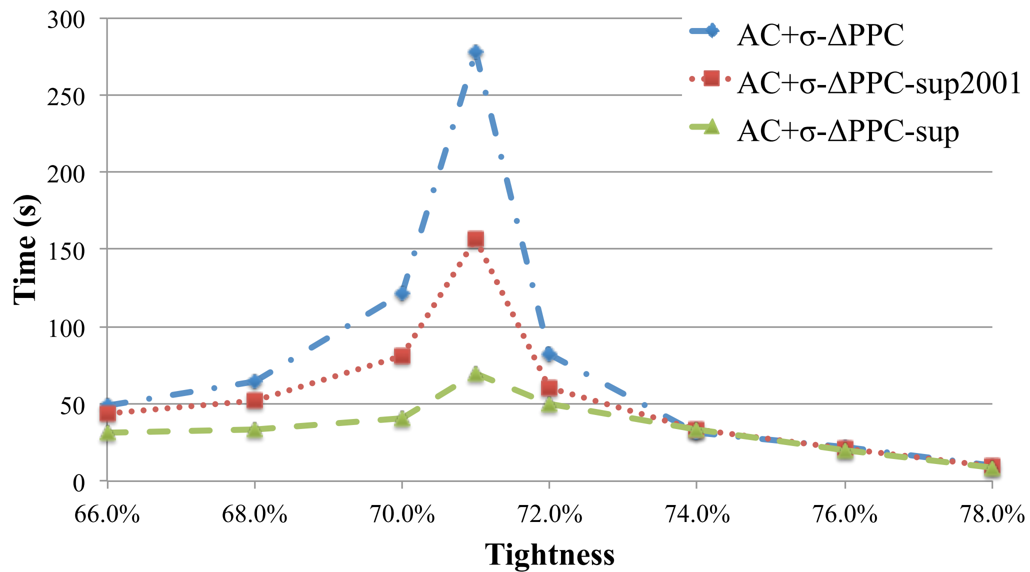


Figure C.12: $\langle 50, 50, t, 20\% \rangle$: Impact of support structures

C.3 $\langle 50, 50, t, 40\% \rangle$

The complete results are summarized in Table C.3

Table C.3: Comparing CPU times on random problems $\langle 50, 50, t, 40\% \rangle$

Tightness	**AC+DPC**	**AC+σ-ΔPPC^{sup}**	**AC+PC-2001-Ordering**	**AC+PC-2001^{+}**	**AC+PC-2001-Flag**	**AC+PC-2001**	**AC+PC-8^{+}**	**AC+PC-8-Ordering**	**AC+PC-8-Flag**	**AC+PC-8**	**AC+σ-ΔPPC^{sup2001}**	**sCDC1**	**AC+σ-ΔPPC**	**AC+ΔPPC**	**AC+sDC2**	**AC+PPC+AP**	**AC+PC-2**
60.0%	15.6	50.7	65.3	69.6	70.7	67.1	67.1	63.5	69.0	66.9	63.9	88.8	71.4	71.2	164.5	120.9	87.4
62.0%	16.8	55.0	73.4	76.2	79.2	77.2	75.9	74.1	81.3	82.8	75.0	110.4	92.8	91.7	199.3	147.8	113.7
64.0%	17.2	59.0	83.0	84.6	91.2	91.9	90.8	92.3	103.6	111.4	90.5	147.6	128.8	125.8	282.0	190.6	164.1
65.0%	17.7	64.2	93.2	93.5	101.8	104.4	107.9	112.0	126.2	137.4	108.3	191.9	171.2	165.8	355.5	242.1	217.3
66.0%	18.0	124.8	191.3	192.2	198.8	205.8	300.2	326.3	326.4	358.6	364.6	410.1	804.8	809.2	875.8	1,020.9	1,110.7
66.5%	18.3	96.9	240.7	240.7	249.8	254.8	404.2	426.9	431.7	458.8	153.9	93.6	281.3	298.8	214.8	326.0	531.9
67.0%	18.4	79.1	188.9	188.8	201.7	201.1	295.7	303.7	323.3	334.4	97.6	48.4	148.6	162.2	110.4	162.6	306.1
68.0%	18.8	61.7	128.5	132.5	134.5	131.0	175.2	170.9	181.4	180.9	63.9	17.9	77.0	84.8	41.5	55.3	168.7
70.0%	20.0	39.5	54.5	57.9	57.6	54.1	53.6	50.1	53.5	50.1	41.3	5.3	29.3	36.9	13.0	26.8	46.8

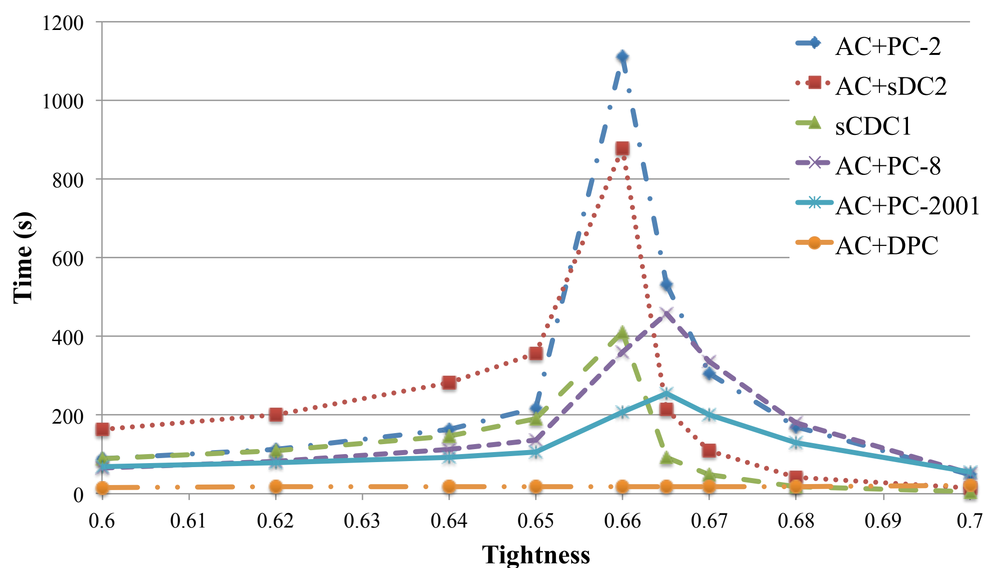


Figure C.13: $\langle 50, 50, t, 40\% \rangle$: Comparing the best previously known algorithms

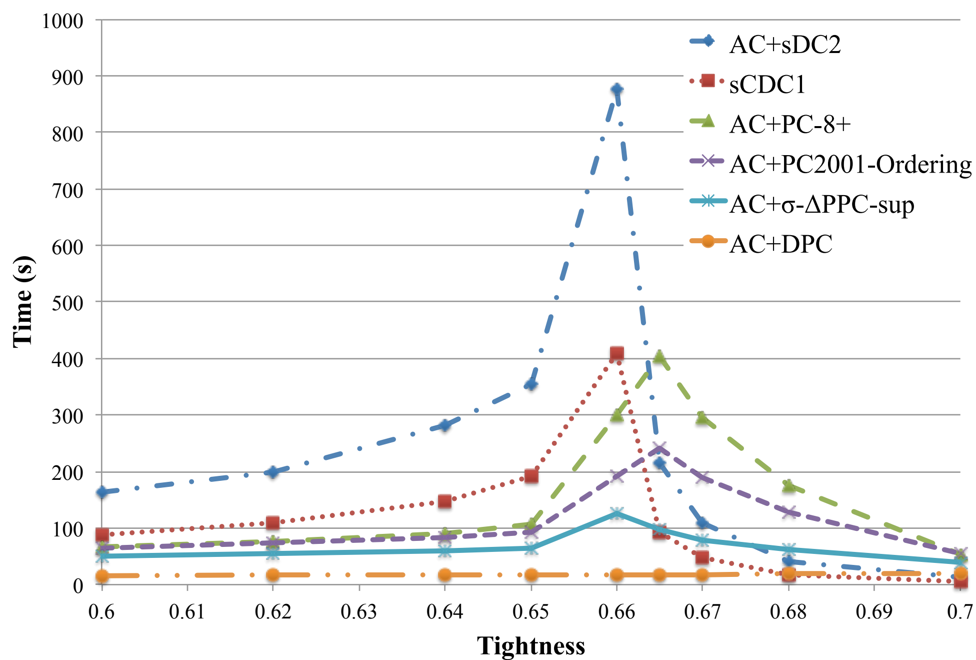


Figure C.14: $\langle 50, 50, t, 40\% \rangle$: Comparing the most competitive algorithms

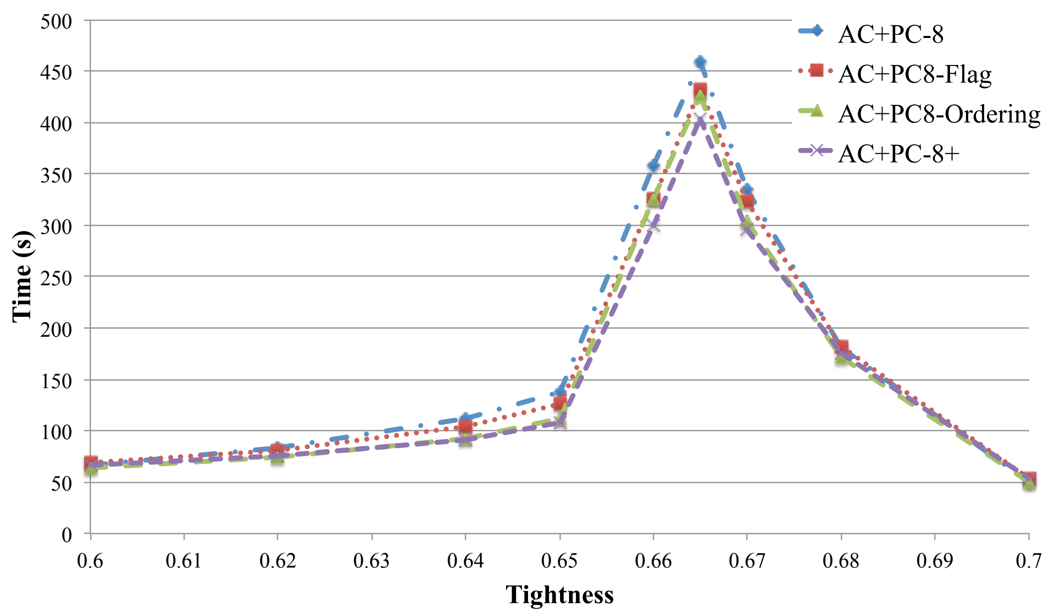


Figure C.15: $\langle 50, 50, t, 40\% \rangle$: Improving PC-8

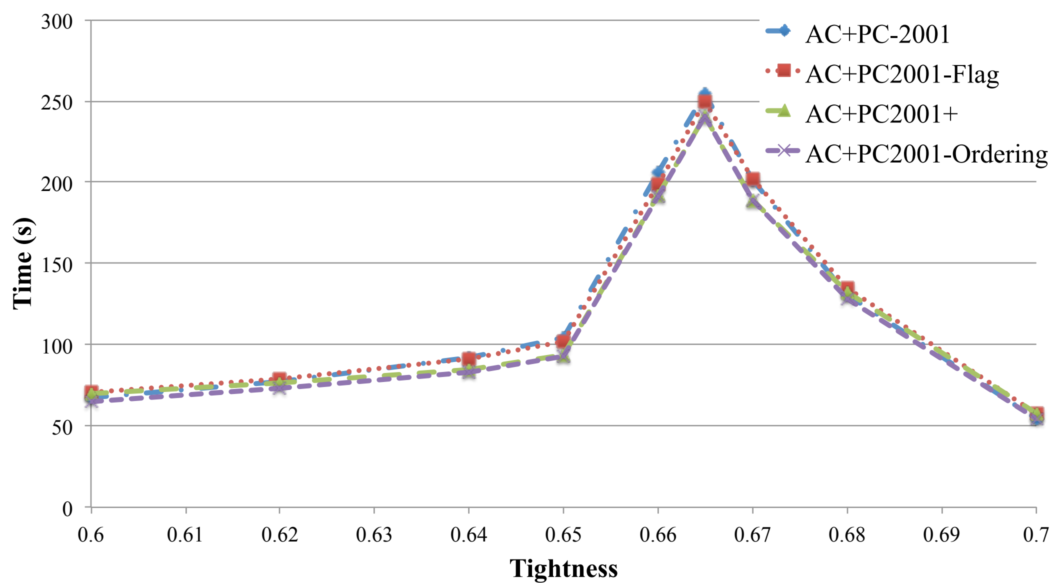


Figure C.16: $\langle 50, 50, t, 40\% \rangle$: Improving PC-2001

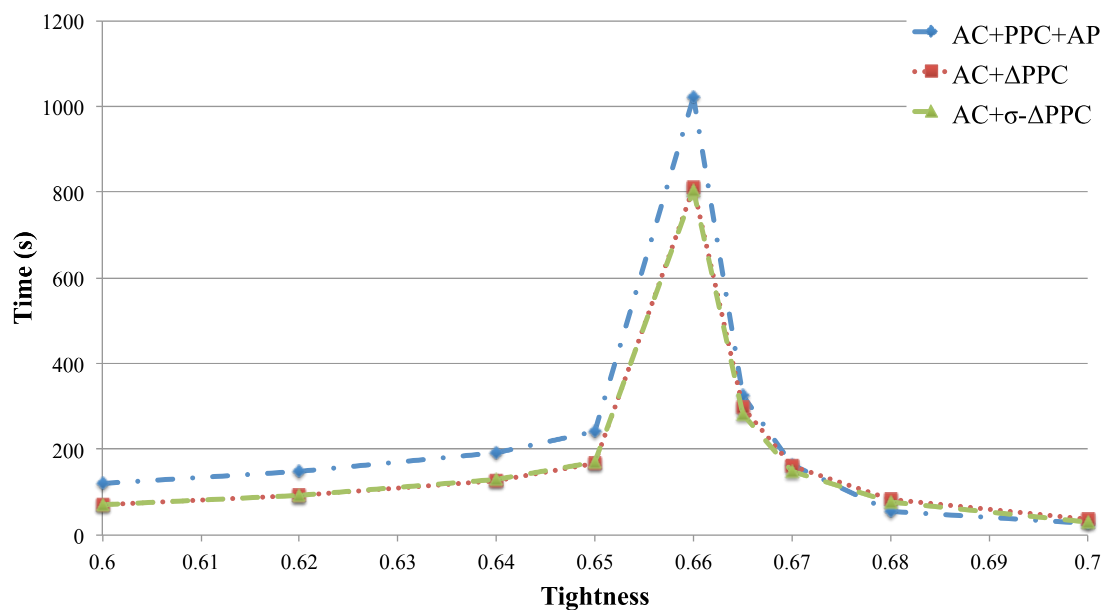


Figure C.17: $\langle 50, 50, t, 40\% \rangle$: Impact of the propagation queue and PEO

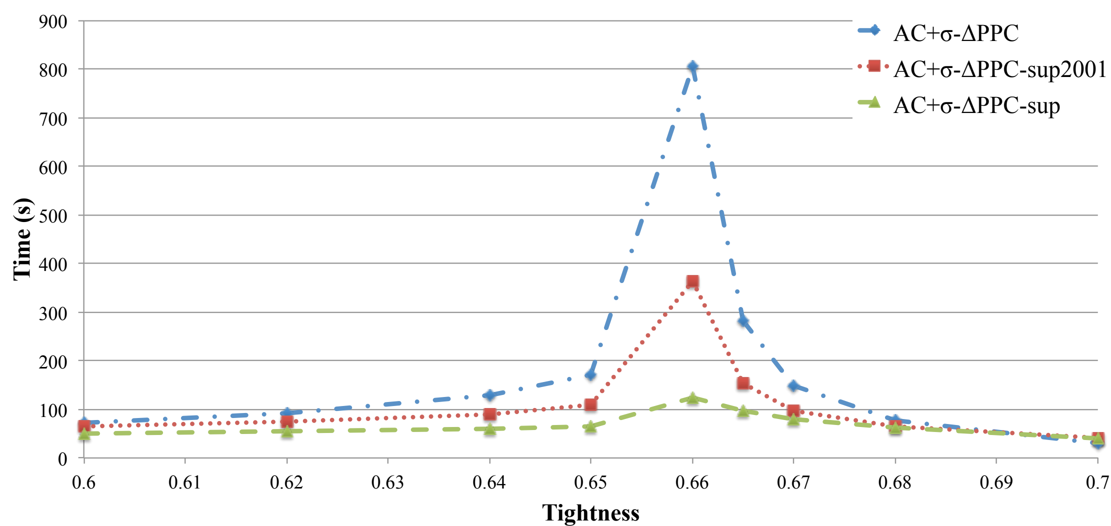


Figure C.18: $\langle 50, 50, t, 40\% \rangle$: Impact of support structures

Bibliography

[Arnborg, 1985] Stefan A. Arnborg. Efficient Algorithms for Combinatorial Problems on Graphs with Bounded Decomposability–A Survey. *BIT*, 25:2–23, 1985.

[Bessière and Régin, 2001] Christian Bessière and Jean-Charles Régin. Refining the Basic Constraint Propagation Algorithm. In *Proceedings of the 17 th International Joint Conference on Artificial Intelligence*, pages 309–315, 2001.

[Bessière *et al.*, 2005] Christian Bessière, Jean-Charles Régin, Roland H.C. Yap, and Yuanlin Zhang. An Optimal Coarse-Grained Arc Consistency Algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

[Bessière *et al.*, 2008] Christian Bessière, Kostas Stergiou, and Toby Walsh. Domain Filtering Consistencies for Non-Binary Constraints. *Artificial Intelligence*, 172:800–822, 2008.

[Bessiere, 2006] Christian Bessiere. *Handbook of Constraint Programming*, chapter Constraint Propagation, pages 29–83. Elsevier, 2006.

[Bliek and Sam-Haroud, 1999] Christian Bliek and Djamilla Sam-Haroud. Path Consistency for Triangulated Constraint Graphs. In *Proceedings of the 16 th International Joint Conference on Artificial Intelligence*, pages 456–461, 1999.

[Bui *et al.*, 2007] Hung H. Bui, Mabry Tyson, and Neil Yorke-Smith. Efficient Message Passing and Propagation of Simple Temporal Constraints: Results on Semi-Structured Networks. In *Workshop on Spatial and Temporal Reasoning held during AAAI 2007*, pages 1–8, 2007.

[Cheeseman *et al.*, 1991] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the Really Hard Problems Are. In *Proceedings of the 12 th International Joint Conference on Artificial Intelligence*, pages 331–337, 1991.

[Chmeiss and Jégou, 1996] Assef Chmeiss and Philippe Jégou. Path-Consistency: When Space Misses Time. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 196–201, 1996.

[Chmeiss and Jégou, 1998] Assef Chmeiss and Philippe Jégou. Efficient Path-Consistency Propagation. *International Journal on Artificial Intelligence Tools*, 7(2):121–142, 1998.

[Chmeiss, 1996] Assef Chmeiss. Sur la Consistance de Chemin et ses Formes Partielles. In *Actes du Congrès AFCET-RFIA 96*, pages 212–219, 1996.

[Debruyne and Bessière, 1997] Romuald Debruyne and Christian Bessière. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In *Proceedings of the 15 th International Joint Conference on Artificial Intelligence*, pages 412–417, 1997.

[Debruyne, 1999] Romuald Debruyne. A Strong Local Consistency for Constraint Satisfaction. In *Proceedings of ICTAI 99*, pages 202–209, 1999.

[Dechter and Pearl, 1988] Rina Dechter and Judea Pearl. Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence*, 34:1–38, 1988.

[Dechter, 2003] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[Eén and Biere, 2005] Niklas Eén and Armin Biere. *Effective Preprocessing in SAT Through Variable and Clause Elimination*, volume 3569 of *LNCS*, pages 61–75. Springer, Berlin, 2005.

[Freuder, 1978] Eugene C. Freuder. Synthesizing Constraint Expressions. *Communications of the ACM*, 21 (11):958–966, 1978.

[Freuder, 1982] Eugene C. Freuder. A Sufficient Condition for Backtrack-Free Search. *JACM*, 29 (1):24–32, 1982.

[Fulkerson and Gross, 1965] D. R. Fulkerson and O. A. Gross. Incidence Matrices and Interval Graphs. *Pacific Journal of Mathematics*, 15 (3):835–855, 1965.

[Geschwender *et al.*, 2016] Daniel J. Geschwender, Robert J. Woodward, Berthe Y. Choueiry, and Stephen D. Scott. A Portfolio Approach for Enforcing Minimality in a Tree Decomposition. In *Proceeding of the International Conference on Principles and Practice of Constraint Programming*, pages 1–10, 2016.

[Gottlob, 2011] Georg Gottlob. On Minimal Constraint Networks. In *Proceedings of 17th International Conference on Principle and Practice of Constraint Programming (CP 11)*, volume 6876 of *Lecture Notes in Computer Science*, pages 325–0339. Springer, 2011.

[Han and Lee, 1988] C.-C. Han and C.-H. Lee. Comments on Mohr and Henderson's Path Consistency Algorithm. *Artificial Intelligence*, 36:125–130, 1988.

[Haralick and Elliott, 1980] Robert M. Haralick and Gordon L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

[Hopcroft and Tarjan, 1973] John E. Hopcroft and Robert Endre Tarjan. Efficient Algorithms for Graph Manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[Karakashian *et al.*, 2010] Shant Karakashian, Robert Woodward, Christopher Reeson, Berthe Y. Choueiry, and Christian Bessiere. A First Practical Algorithm for High Levels of Relational Consistency. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence*, pages 101–107, 2010.

[Karakashian *et al.*, 2013] Shant Karakashian, Robert Woodward, and Berthe Y. Choueiry. Improving the Performance of Consistency Algorithms by Localizing and Bolstering Propagation in a Tree Decomposition. In *Proceedings of the 27th Conference on Artificial Intelligence*, pages 466–473, 2013.

[Kjærulff, 1990] U. Kjærulff. Triagulation of Graphs - Algorithms Giving Small Total State Space, 1990.

[Lecoutre *et al.*, 2007a] Christophe Lecoutre, Stéphane Cardon, and Julien Vion. Conservative Dual Consistency. In *Proceedings of the 22nd Conference on Artificial Intelligence*, pages 237–242, 2007.

[Lecoutre *et al.*, 2007b] Christophe Lecoutre, Stéphane Cardon, and Julien Vion. Path Consistency by Dual Consistency. In *Proceedings of the 13th International Conference on Principle and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 448–452. Springer Verlag, 2007.

[Lecoutre *et al.*, 2011] Christophe Lecoutre, Stéphane Cardon, and Julien Vion. Second-Order Consistencies. *Journal of Artificial Intelligence Research (JAIR)*, 40:175–219, 2011.

[Long *et al.*, 2016] Zhiguo Long, Michael Sioutis, and Sanjiang Li. Efficient Path Consistency Algorithm for Large Qualitative Constraint Networks. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, pages 1202–1208, 2016.

[Mackworth and Freuder, 1984] Alan K. Mackworth and Eugene C. Freuder. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence*, (25) 1:65–74, 1984.

[Mackworth, 1977] Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.

[Boerkoel Jr. and Durfee, 2013] James C. Boerkoel Jr. and Edmund H. Durfee. Distributed Reasoning for Multiagent Simple Temporal Problems. *J. Artif. Intell. Res. (JAIR)*, 47:95–156, 2013.

[McGregor, 1979] J. J. McGregor. Relational Consistency Algoritms and their Application in Finding Subgraph and Graph Isomorphims. *Information Sciences*, 19:229–259, 1979.

[Mohr and Henderson, 1986] Roger Mohr and Thomas C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Montanari, 1974] Ugo Montanari. Networks of Constraints: Fundamental Properties and Application to Picture Processing. *Information Sciences*, 7:95–132, 1974.

[Planken *et al.*, 2008] Léon Planken, Mathijs de Weerdt, and Roman van der Krogt. P3C: A New Algorithm for the Simple Temporal Problem. In *Processings of the 18th International Conference on Automated Planning & Scheduling*, pages 256–263, 2008.

[Schneider *et al.*, 2014] Anthony Schneider, Robert J. Woodward, Berthe Y. Choueiry, and Christian Bessiere. Improving Relational Consistency Algorithms Using Dynamic Relation Partitioning . In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, volume 8656 of *LNCS*, pages 688–704. Springer, 2014.

[Singh, 1996] Moninder Singh. Path Consistency Revisited. *International Journal on Artificial Intelligence Tools*, 5(1-2):127–142, 1996.

[Woodward *et al.*, 2011] Robert Woodward, Shant Karakashian, Berthe Y. Choueiry, and Christian Bessiere. Solving Difficult CSPs with Relational Neighborhood Inverse Consistency. In *Proceedings of 25th AAAI Conference on Artificial Intelligence (AAAI 11)*, pages 112–119, 2011.

[Woodward *et al.*, 2012] Robert J. Woodward, Shant Karakashian, Berthe Y. Choueiry, and Christian Bessiere. Revisiting Neighborhood Inverse Consistency on Binary CSPs. In *Eighteenth International Conference on Principles and Practice of Constraint Programming (CP 2012)*, volume 7514 of *Lecture Notes in Computer Science*, pages 688–703. Springer, 2012.

[Woodward *et al.*, 2014] Robert J. Woodward, Anthony Schneider, Berthe Y. Choueiry, and Christian Bessiere. Adaptive Parameterized Consistency for Non-Binary CSPs by Counting Supports. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, volume 8656 of *LNCS*, pages 755–764. Springer, 2014.

[Xu and Choueiry, 2003] Lin Xu and Berthe Y. Choueiry. A New Efficient Algorithm for Solving the Simple Temporal Problem. In Mark Reynolds and Abdul Sattar, editors, *Proceedings of the 10^{t}h International Symposium on Temporal Representa-*

tion and Reasoning and Fourth International Conference on Temporal Logic, pages 212–222. IEEE Computer Society Press, 2003.

[Yannakakis, 1981] Mihalis Yannakakis. Computing the Minimum Fill-In is NP-Complete. *SIAM Journal on Algebraic and Discrete Methods*, 2 (1):77–79, 1981.

[Yorke-Smith, 2005] Neil Yorke-Smith. Exploiting the Structure of Hierarchical Plans in Temporal Constraint Propagation. In *Proceedings of the 20th National Conference on Artificial Intelligence*, pages 1223–1228, 2005.

[Zhang and Yap, 2001] Yualin Zhang and Roland H.C. Yap. Making AC-3 an Optimal Algorithm. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 316–321, 2001.