University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

Computer Science and Engineering: Theses, Dissertations, and Student Research | Computer Science and Engineering, Department of

5-2018

Effectively Enforcing Minimality During Backtrack Search

Daniel J. Geschwender
University of Nebraska-Lincoln, geschd23@gmail.com

Follow this and additional works at: https://digitalcommons.unl.edu/computerscidiss

Part of the Artificial Intelligence and Robotics Commons

Geschwender, Daniel J., "Effectively Enforcing Minimality During Backtrack Search" (2018). *Computer Science and Engineering: Theses, Dissertations, and Student Research*. 151.
https://digitalcommons.unl.edu/computerscidiss/151

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in Computer Science and Engineering: Theses, Dissertations, and Student Research by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

EFFECTIVELY ENFORCING MINIMALITY DURING BACKTRACK SEARCH

by

Daniel J. Geschwender

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

May, 2018

EFFECTIVELY ENFORCING MINIMALITY DURING BACKTRACK SEARCH

Daniel J. Geschwender, M.S.

University of Nebraska, 2018

Adviser: Berthe Y. Choueiry

Constraint Processing is an expressive and powerful framework for modeling and solving combinatorial decision problems. Enforcing consistency during backtrack search is an effective technique for reducing thrashing in a large search tree. The higher the level of the consistency enforced, the stronger the pruning of inconsistent subtrees. Recently, high-level consistencies (HLC) were shown to be instrumental for solving difficult instances. In particular, minimality, which is guaranteed to prune all inconsistent branches, is advantageous even when enforced locally. In this thesis, we study two algorithms for computing minimality and propose three new mechanisms that significantly improve performance. Then, we integrate the resulting algorithms in a portfolio that operates both locally and dynamically during search. Finally, we empirically evaluate the performance of our approach on benchmark problems.

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Berthe Choueiry, for her invaluable support throughout my graduate studies. She opened the doors of many academic opportunities for me, helped to guide and refine my research, and provided extensive feedback in the revision of this thesis. I also want to thank the members of my committee, Professor Stephen Scott and Professor Hongfeng Yu, for their valuable comments, suggestions, and ideas for future work.

Tony Schneider and Robert Woodward have helped me tremendously throughout this research. Tony initiated the task of developing STAMPEDE, the solver on which I built all my algorithms. He put considerable effort into creating a modular solver that accommodates our research-oriented needs. I developed several of the structures and algorithms discussed in this thesis (Section 3.1 and Section 3.4.1) in collaboration with Tony. Similarly, Robert made substantial contributions to the STAMPEDE solver of which I have benefited. In particular, the cluster-computer job submission and results formatting scripts that Robert wrote were instrumental in the collection of my experimental data.

I would also like to thank Shant Karakashian and Professor Stephen Scott for the collaboration that led to the work in Chapter 6, Dr. Laura Damuth for her help during the NSF GRFP application-process, and Professor Kent Eskridge of the Department of Statistics of UNL for introducing me to factorial experimental design.

Finally, I want to express my gratitude to my friends and family for their support.

This research was partially supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1041000 and NSF Grant No. RI-1619344. Experiments were conducted on the equipment at the Holland Computing Center at UNL.

Contents

Contents iv

List of Figures viii

List of Tables x

1 Introduction 1

1.1 Motivation 2

1.2 Contributions 2

1.3 Thesis Structure 4

2 Background 5

2.1 The Constraint Satisfaction Problem 5

2.2 Solving CSPs 7

2.3 Graphical Representations 8

2.3.1 Constraint Networks 8

2.3.2 Minimal Dual Graphs 10

2.3.3 Tree-Structured Constraint Networks 11

2.3.4 Graham Reduction 11

2.3.5 Tree Decomposition 12

2.4 Consistency Properties 14
2.4.1 Arc Consistency 15
2.4.2 Pairwise Consistency 15
2.4.3 Minimality 16
2.5 Consistency Algorithms 17
2.5.1 Lookahead 17
2.5.2 STR2 to Enforce Generalized Arc Consistency 18
2.5.3 PW-AC2 to Enforce Pairwise Consistency (PWC) 18
2.5.4 PerTuple, AllSol to Enforce Constraint Minimality 19
2.6 Cluster-Based Minimality 21

3 Improving Minimality Algorithms 24

3.1 Unmarked-First (UF) Ordering Heuristic 25
3.2 Dangle Identification 26
3.3 Dom/wdeg Variable Ordering 29
3.4 Maintaining Consistency 30
3.4.1 The DualFC Algorithm 31
3.4.2 The DualRFL Algorithm 33
3.4.3 The DualDangleFC Algorithm 38
3.5 Minimal Dual Graph 40
3.6 Correctness 41
3.7 Experimental Evaluation 43
3.7.1 Setup 44
3.7.2 Results 45
3.7.2.1 Performance of the 64 Configurations 45
3.7.2.2 Comparing the Original and Best Configurations . . 50

3.7.2.3 ANOVA Results 53
3.7.3 Discussion 58

4 Which Minimal Dual Graph **61**

4.1 A Minimal Dual Graph in PerTuple 61
4.2 The MaxDeg Heuristic for a Minimal Dual Graph 63
4.3 Metrics for Dangle Identification 65
4.3.1 Normalized Average Dangle Level 65
4.3.2 Average Percent Dangles Identified 66
4.4 Experimental Evaluation 66
4.4.1 Setup 66
4.4.2 Results 67
4.4.3 Discussion 72

5 Weight Update in High-Level Consistencies **73**

5.1 Weight-Update Strategies: Motivation 73
5.2 Weight-Update Parameters 74
5.2.1 Occurrence 75
5.2.2 Distribution 76
5.2.3 Scale 77
5.3 Experimental Evaluation 78
5.3.1 Setup 79
5.3.2 Results 80
5.3.3 Discussion 83

6 Dynamic Portfolio for Cluster-Based Minimality **84**

6.1 Collecting Training Data 85

6.1.1 Features 85

6.1.1.1 General Features 85

6.1.1.2 Graph Features 87

6.1.1.3 Tree-Decomposition Features 88

6.1.2 Aggregate Functions 89

6.2 Decision-Tree Classifier 90

6.2.1 Labels and Weights for Classification 90

6.2.2 Training 93

6.2.3 Training Results 93

6.2.4 Trained Decision Tree 95

6.2.5 Alternate Classifiers 96

6.3 Experimental Evaluation 98

6.3.1 Setup 98

6.3.2 Cluster-Minimality Algorithms Using dom/deg 100

6.3.3 Cluster-Minimality Algorithms Using dom/wdeg 110

6.3.4 Discussion 115

7 Conclusions and Future Work **116**

7.1 Conclusions 116

7.2 Future Work 117

A Results of Experiments in Section 6.3 **120**

Bibliography **161**

List of Figures

2.1 Relations from Example 1 . . . 6
2.2 Two equivalent hypergraph representations of Example 1 . . . 8
2.3 Primal graph of Example 1 . . . 8
2.4 Dual graph of Example 1 . . . 9
2.5 Incidence graph of Example 1 . . . 10
2.6 An example dual graph before redundancy removal . . . 10
2.7 An example dual graph after redundancy removal . . . 10
2.8 Triangulated primal graph of Example 1 and its maximal cliques . . . 13
2.9 Tree decomposition of Example 1 . . . 13
2.10 Cluster-based minimality . . . 21

3.1 Instance completions over time . . . 51
3.2 PerTuple: runtime comparison per instance . . . 52
3.3 AllSol: runtime comparison per instance . . . 52

4.1 Dual graphs of an example instance . . . 64
4.2 PerTuple's runtime on original vs. minimal dual graph with MinDeg . . . 69
4.3 AllSol's runtime on original vs. minimal dual graph with MinDeg . . . 69
4.4 PerTuple's runtime on minimal dual graph with MinDeg versus MaxDeg . . . 70
4.5 AllSol's runtime on minimal dual graph with MinDeg versus MaxDeg . . . 70

4.6 PerTuple's runtime on original vs. minimal dual graph with MaxDeg 71

4.7 AllSol's runtime on original vs. minimal dual graph with MaxDeg 71

6.1 The weighted and labeled training data 92

6.2 Confusion matrix of the decision-tree classifier 94

6.3 The trained decision-tree classifier 95

6.4 Confusion matrices of two alternate classifiers 96

6.5 Instance completions over time with dom/deg 103

6.6 Instance completions over time with dom/wdeg 112

List of Tables

3.1 The six tested factors 44
3.2 Results summary for all tested configurations 46
3.3 ANOVA results sorted by generalized eta squared 54

4.1 Degree statistics of the dual graphs in Figure 4.1 65
4.2 Results summary for tested dual graph types 68

5.1 The three factors tested 79
5.2 Results summary for all tested weight-update strategies 81
5.3 Weight update ANOVA results sorted by generalized eta squared 82

6.1 Classifier training results 94
6.2 Benchmarks where a given algorithm performs best (dom/deg) 101
6.3 Performance summary using dom/deg 101
6.4 Per-benchmark performance using dom/deg 105
6.5 Benchmarks where a given algorithm performs best (dom/wdeg) 110
6.6 Performance summary using dom/wdeg 111
6.7 Per-benchmark performance using dom/wdeg 113

A.1 Results summary for STR2 and cluster-minimality algorithms 122
A.2 STR2 and cluster-minimality algorithms using dom/deg 125

A.3 STR2 and cluster-minimality algorithms using dom/wdeg 143

List of Algorithms

1 PerTuple($\mathcal{P}_D$) [Karakashian *et al.*, 2012] 20
2 AllSol($\mathcal{P}_D$) [Karakashian *et al.*, 2012] 20
3 IdentifyDangles(*unremoved*,*dangleVs*,*dangleEs*,*level*) 28
4 RestoreDangles(*unremoved*,*dangleVs*,*dangleEs*,*level*) 29
5 DualFC(*assignedCon*,*unassigned*) 31
6 ReviseFC(*reviseCon*,*againstCon*) 32
7 SaveIntersection(*domain*,*block*) 32
8 RemoveDifference(*domain*,*block*,*subscope*) 32
9 DualRFL(*assignedCon*,*unassigned*) 34
10 ReviseRFL(*reviseCon*,*againstCon*) 35
11 CheckForDeadBlock(*domain*, *block*, *subscope*) 36
12 SearchBlockForLivingTuple(*domain*, *block*) 37
13 SearchLivingForBlockTuple(*domain*, *block*, *subscope*) 37
14 DualDangleFC(*assignedCon*,*unassigned*,*unremoved*,*dangleEs*,*level*) . . . 39
15 UpCurrentDangles(*dangleEs*,*level*) 40
16 DownAllDangles(*dangleEs*) 40

Chapter 1

Introduction

Consistency properties and algorithms for enforcing them are central to research in Constraint Processing (CP). Consistency properties range from the local and efficiently computable ones, such as the popular arc consistency, to the global and likely intractable ones, such as minimality and decomposability. In practice, enforcing consistency during search (i.e., inference) can significantly reduce the size of the search tree. Further, the higher the level of the consistency enforced, the stronger the pruning of inconsistent subtrees. Although high-level consistency (HLC) properties are often costly to enforce, they were shown to be beneficial for solving difficult problem instances. We study algorithms for enforcing constraint minimality on clusters of a tree decomposition of a Constraint Satisfaction Problem (CSP) in continuation of early investigations initiated in the Constraint Systems Laboratory [Karakashian *et al.*, 2013; Karakashian, 2013].

1.1 Motivation

High-level consistencies are powerful tools that enable us to solve difficult problems that are otherwise unsolvable within reasonable amount of time. Constraint minimality is one such property: It ensures that every tuple in the relation of a constraint appear in a solution to the CSP. Enforcing minimality is known to be NP-complete [Gottlob, 2011] and rarely used in practice. However, recent research has identified situations where it is beneficial [Bayer *et al.*, 2007; Karakashian *et al.*, 2013; Bessiere *et al.*, 2013].

We believe that high-level consistencies, in particular constraint minimality, have much to contribute to CSP solvers. For this reason, we seek to improve the efficiency of the algorithms for enforcing minimality, adapt search ordering heuristics to this context, and design strategies for enforcing it where it is most beneficial.

1.2 Contributions

In this thesis, we describe five primary contributions. The first three contributions aim at improving the performance of two algorithms for enforcing constraint minimality, namely AllSol and PerTuple; the fourth one investigates weight updates in the dom/wdeg ordering heuristic; and the last one is a portfolio algorithm for locally enforcing minimality on subproblems defined by a tree decomposition of the CSP.

Unmarked-first ordering heuristic (UF). In order to determine that a value in the domain of a variable (alternatively, a tuple in the relation of a constraint) is minimal, algorithms for enforcing minimality conduct a backtrack search to ensure that the value (tuple) appear in a solution. Further, they also mark the other values (tuples) that appear in the solution as minimal, thus, extending the benefits of

the search beyond the tested value (tuple). The UF ordering heuristic prioritizes using values (tuples) that are unmarked during the search and yields performance improvement.

Dangle identification. As search proceeds in AllSol and PerTuple, the constraint network becomes increasingly sparser. Tree-structured subgraphs emerge and can be efficiently identified. We propose to dynamically identify these 'dangles' as tractable subproblems. Indeed, enforcing directional arc consistency is sufficient to to guarantee all the values (alternatively, tuples) that appear in them are minimal provided the current search path successfully terminates. This mechanism not only reduces the search effort by reducing the size of the search space but also can determine the minimality of a significantly larger number of values (tuples).

The MaxDeg heuristic for minimal dual graph. For algorithms for constraint minimality, we show that using a minimal dual graph (which is never denser than the dual graph) boosts the benefits of dangle identification. However, a minimal dual graph is not unique. We investigate how to exploit the algorithm proposed by Janssen *et al.* [1989] for computing a minimal dual graph in order to promote the appearance of dangles during search. We introduce the MaxDeg heuristic for this algorithm and show that the resulting minimal dual graph is more conducive to dangle identification.

Weight-updates of dom/wdeg in the context of high-level consistencies. The highly successful dom/wdeg variable ordering heuristic relies on updating the weight of the constraint that detects inconsistency when enforcing Generalized Arc Consistency (GAC) during search [Boussemart *et al.*, 2004]. In the context of a high-level consistency, such as constraint minimality, there is not a definitive method for attributing blame in the event of a wipeout. We propose a framework for constraint-

weight updates that is suitable for high-level consistencies and evaluate a wide variety of such heuristics.

A cluster-level portfolio for enforcing cluster minimality. In continuation of previous work [Karakashian *et al.*, 2013; Karakashian, 2013], we enforce constraint minimality on the clusters of a tree decomposition of a CSP. Further, with our widely improved versions of AllSol and PerTuple, we advocate a fine-grain portfolio approach at the level of a cluster during search by which we dynamically choose the 'most appropriate' minimality algorithm every time we process a cluster. We argue the novelty of this approach and its effectiveness in solving difficult problems. Preliminary results of this contribution have been published. [Geschwender *et al.*, 2013; Geschwender *et al.*, 2016]

1.3 Thesis Structure

The thesis is structured as follows. Chapter 2 discusses relevant background information. Chapter 3 introduces two of our three improvements of the algorithms for enforcing constraint minimality, AllSol and PerTuple. These improvements are the unmarked-first heuristic and the dangle-identification mechanism. Chapter 4 introduces the MaxDeg heuristic for choosing a minimal dual graph and evaluates its impact on dangle identification. Chapter 5 discusses our weight-update framework for dom/wdeg in the context of high-level consistencies. Chapter 6 introduces our cluster-level portfolio and evaluates its effectiveness. Chapter 7 concludes this thesis.

Chapter 2

Background

In this chapter, we review background information that is relevant to the remainder of the thesis.

2.1 The Constraint Satisfaction Problem

A *Constraint Satisfaction Problem* (CSP) is given by $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X} = \{x_1, ..., x_n\}$ is a set of n *variables*, each with an associated finite *domain* in $\mathcal{D} = \{D_1, ..., D_n\}$ and $\mathcal{C} = \{C_1, ..., C_e\}$ is a set of *constraints* that restrict how values can be assigned to variables. Each constraint C_i is defined by a *relation* R_i over a subset of variables, which is the *scope* of C_i. The relation R_i is a subset of the Cartesian product of the domains of the variables in the scope of C_i. In this thesis, we consider *table constraints*, where the relation R_i of a constraint C_i is given by a set of allowed tuples or *supports*. The *arity* of a constraint is the cardinality of its scope. A *binary* CSP is a CSP containing only constraints of arity two. A *solution* to a CSP is an assignment to each variable a value from its domain such that all constraints are satisfied.

Example 1. *Consider* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ *where:*

- $\mathcal{X} = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N\}$
- $\mathcal{D} = \{D_A, D_B, D_C, D_D, D_E, D_F, D_G, D_H, D_I, D_J, D_K, D_L, D_M, D_N\}$, *with* $D_{i \in \mathcal{X}} = \{0, 1\}$
- $\mathcal{C} = \{R_1, R_2, R_3, R_4, R_5, R_6, R_7\}$ *with relations given in Figure 2.1.*

The tuples highlighted in Figure 2.1 correspond to the solution obtained from the following assignments:

$$(A, 1), (B, 0), (C, 0), (D, 0), (E, 1), (F, 1), (G, 1),$$

$$(H, 1), (I, 1), (J, 1), (K, 0), (L, 1), (M, 0), (N, 0).$$

R_1			
A	***B***	***C***	***N***
0	0	1	1
1	0	0	0
1	0	1	0
1	1	1	1

R_2		
I	***M***	***N***
0	0	0
0	0	1
0	1	0
1	0	0

R_3		
I	***J***	***K***
0	1	1
1	0	1
1	1	0

R_4		
A	***K***	***L***
0	0	0
0	0	1
0	1	1
1	0	1
1	1	1

R_5			
B	***D***	***E***	***F***
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

R_6		
C	***D***	***H***
0	0	1
1	1	0

R_7		
F	***G***	***H***
0	0	0
0	1	0
0	1	1
1	1	0
1	1	1

Figure 2.1: Relations from Example 1

2.2 Solving CSPs

Determining whether or not a CSP has a solution is NP-complete. Backtrack search is a sound and complete algorithm for finding a solution to a given CSP. It is a systematic, constructive procedure that instantiates variables, one by one, in a depth-first manner, until a solution is found or a dead-end reached. At each variable instantiation, some consistency property is enforced on the remaining subproblem. When the subproblem is found to be inconsistent, the assignment is undone and a different assignment is attempted for the variable. Once all values in a variable's domain are exhausted, the search backtracks to the immediately previous level and undoes the assignment made at that level. When the search procedure backtracks past the first variable without finding a solution, no solution exists.

The order in which the variables (and values) are instantiated is determined by *ordering heuristics*. The baseline heuristic (which is typically used as a tie-breaker) is a lexicographic ordering. For variable ordering, numerous heuristics have been proposed. The dom/deg heuristic selects the variable with the minimum ratio of current domain size to current degree in the constraint graph (described in Section 2.3). A variant of this heuristic, dom/wdeg is particularly effective and currently the most popular variable ordering heuristic [Boussemart *et al.*, 2004]. Dom/wdeg is computed similarly to dom/deg but uses, instead of the current degree, the 'weighted current degree,' which is the sum of the weights of incident constraints. A constraint's weight is initialized to one and incremented by one whenever the constraint is responsible for detecting an inconsistent future subproblem. This heuristic is designed to prioritize assigning highly constrained variables as an implementation of the general principle of 'most-constrained variable first.'

2.3 Graphical Representations

Below we review the main graphical representations of a CSP.

2.3.1 Constraint Networks

Several graphical representations of CSPs exist:

- *Hypergraph* (Figure 2.2): The vertices represent variables. Hyperedges represent constraints and connect the variables in the scope of the constraints.

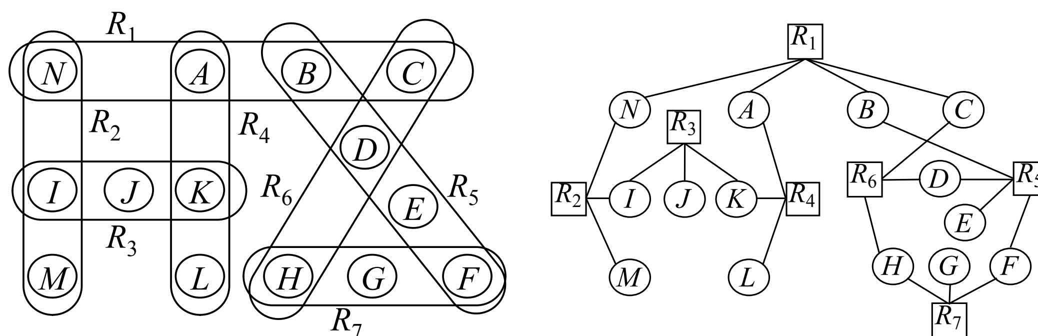


Figure 2.2: Two equivalent hypergraph representations of Example 1

- *Primal graph* (Figure 2.3): In this graph, the vertices represent variables and edges connect all the pairs of variables such that both variables are in the scope of some constraint.

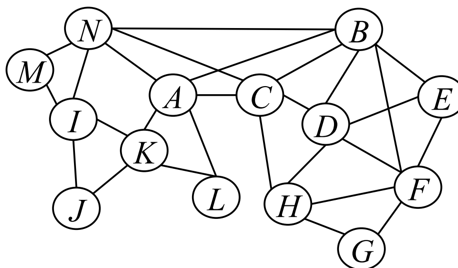


Figure 2.3: Primal graph of Example 1

- *Dual graph* (Figure 2.4): Vertices represent the constraints and edges connect constraints that share variables in their scope. The dual graph is of particular

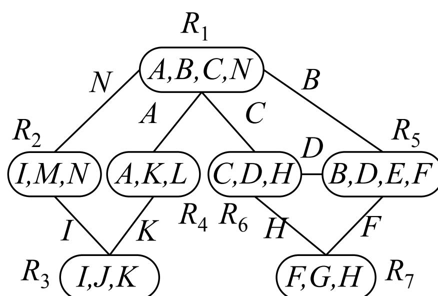


Figure 2.4: Dual graph of Example 1

interest because it presents a dual CSP, where vertices (the original constraints) represent dual variables whose domains are the tuples of the relations of their respective constraints. The edges are *equality* constraints forcing the assigned tuples to agree on the values in their shared CSP variables. All constraints are binary in the dual CSP. Throughout this thesis, we make extensive use of the dual graph and dual CSP. When referring to the variables and constraints of a dual CSP, we refer respectively to the constraints of the original CSP and the new equality constraints introduced between the (dual) variables in the dual CSP. Further, we designate by *subscope* the set of CSP variables shared by two constraints.

- *Incidence graph* (Figure 2.5): The graph represents both variables and constraints as vertices. Edges link vertices representing constraints to the vertices representing the variables in their scope. Thus, the incidence graph is a bipartite graph with vertices representing constraints in one part and vertices representing variables in the other part.

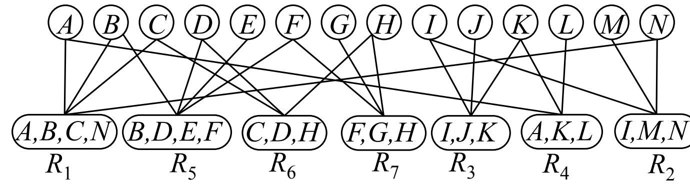


Figure 2.5: Incidence graph of Example 1

2.3.2 Minimal Dual Graphs

The dual-graph representation may be simplified by removing redundant edges to obtain an equivalent network [Janssen *et al.*, 1989; Dechter, 2003]. An edge is redundant if its removal does not change the solutions to the problem. Redundant edges can occur when the scopes of three or more constraints have shared CSP variables. In the dual graph, the vertices representing these overlapping constraints are fully connected with edges representing equality constraints. However, because of the transitivity of equality, a chain of these constraints is sufficient to enforce this equality. A dual graph with no redundant edges is a *minimal dual graph*. Figure 2.6 shows an example of a dual graph with redundant edges. The relations R_1, R_3, and R_4 share the variable A.

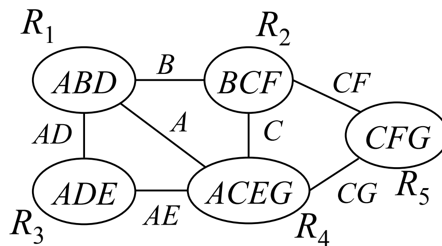


Figure 2.6: An example dual graph before redundancy removal

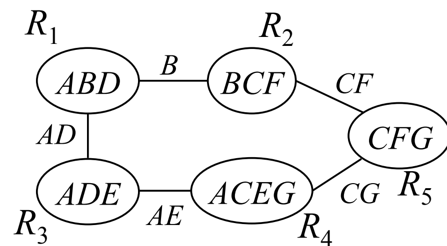


Figure 2.7: An example dual graph after redundancy removal

Thus, their respective vertices form a clique in the dual graph. Similarly, relations R_2, R_4, and R_5 share the variable C. Figure 2.7 shows the dual graph after removing redundant edges, namely, the edge (R_1,R_4) labeled 'A' and the edge (R_2,R_4) labeled 'C', making it a minimal dual graph.

A given dual graph may have many minimal dual graphs, all of which are guaranteed to have the same number of edges. Janssen *et al.* [1989] introduced an efficient algorithm to compute a minimal dual graph. Different heuristics for edge selection can easily be integrated in this algorithm in order to produce different minimal dual graphs.

2.3.3 Tree-Structured Constraint Networks

Trees are a known tractable structure in Constraint Processing: A tree-structured constraint network can be solved in a backtrack-free manner after enforcing arc consistency [Freuder, 1982] or even directional arc consistency [Dechter and Pearl, 1988] (arc consistency is described in Section 2.4.1). A constraint network that is not tree-structured may become tree structured after some assignments have been made.

The Cycle-Cutset method proposed by Dechter and Pearl [1987] identifies a set of vertices in the network that, when removed, leave the network as a forest. The removed vertices corresponding to the cutset vertices. After finding a solution to the cutset vertices and enforcing AC on the remaining trees, we can then solve the trees in a backtrack-free manner.

2.3.4 Graham Reduction

The Graham reduction is an algorithm used in databases to determine the acyclicity of a database scheme [Maier, 1983]. The database scheme is expressed as a hypergraph $H = (N, E)$, where N is the set of nodes and E is the set of edges. Two reductions are repeatedly applied until neither can be applied further:

- Hyperedge removal: if edges $e, f \in E$ are such that e is properly contained in f, remove e from E.

- Node removal: if node $a \in N$ appears in only one edge $e \in E$, remove a from N and e from E.

The Graham reduction can be applied to any hypergraph to remove all acyclic portions of the hypergraph.

2.3.5 Tree Decomposition

A *tree decomposition* of a CSP is a tree embedding of its constraint network. The tree vertices are *clusters* of variables and constraints from the CSP. The set of variables of a cluster cl is denoted $\chi(cl) \subseteq \mathcal{X}$, and the set of constraints $\psi(cl) \subseteq \mathcal{C}$. A tree decomposition must satisfy two conditions:

1. Each constraint appears in at least one cluster and the variables in its scope must appear in this cluster; and
2. For every variable, the clusters where the variable appears induce a connected subtree.

Many techniques for generating a tree decomposition of a CSP exist [Dechter and Pearl, 1989; Jeavons *et al.*, 1994; Gottlob *et al.*, 1999]. We use an adaption for non-binary CSPs of the tree-clustering technique [Dechter and Pearl, 1989]. First, we triangulate the primal graph of the CSP using the min-fill heuristic [Kjærulff, 1990]. Then, we identify the maximal cliques in the resulting chordal graph using the MaxCliques algorithm [Golumbic, 1980] and use the identified maximal cliques to form the clusters of the tree decomposition. We build the tree by connecting the clusters using the JoinTree algorithm [Dechter, 2003]. While any cluster can be chosen as the root of the tree, we choose the cluster that minimizes the longest chain

from the root to a leaf. Figure 2.8 shows a triangulated primal graph of the example in Figure 2.3.

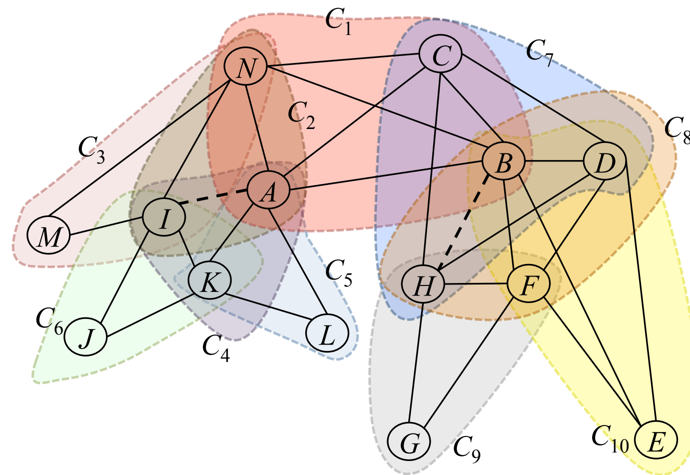


Figure 2.8: Triangulated primal graph of Example 1 and its maximal cliques

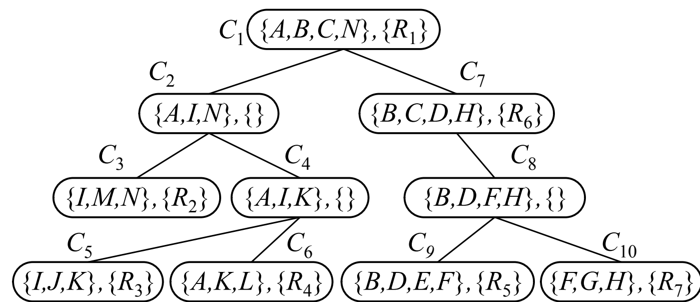


Figure 2.9: Tree decomposition of Example 1

The dotted edges (B, H) and (A, I) in Figure 2.8 are fill-in edges generated by the triangulation algorithm. The ten maximal cliques of the triangulated graph are highlighted with 'blobs.' The resulting tree decomposition is shown in Figure 2.9.

A *separator* of two adjacent clusters is the set of variables that are associated with both clusters. A given tree decomposition is characterized by its *treewidth*, which is the maximum number of variables in a cluster minus one. The complexity of solving a CSP using a given tree decomposition can be bound in time by the treewidth of the decomposition and in space by the size of its largest separator. The *treewidth of*

a constraint network is the minimum treewidth of all its decompositions; computing it is known to be NP-hard [Arnborg, 1985].

In order to guarantee perfect 'message passing' across clusters, one would have to generate a unique constraint over all the variables of a separator, which is prohibitively expensive in space (i.e., the size of the constraint's table grows exponentially with the number of variables in the separator). As an approximation, and in order to enhance constraint propagation between two adjacent clusters, we use the projection schema described by Karakashian *et al.* [2013]. According to this bolstering strategy, we add to each cluster the projection on the variables of the cluster of all the constraints (from outside the cluster), then we normalize the constraints in the cluster by merging all two constraints where the scope of one is a subset the other's.

2.4 Consistency Properties

Consistency properties guarantee that a constraint network meets some condition and serve as the basis for inference (i.e., reasoning) on a CSP. These properties are defined:

- Globally or locally, and
- On the domains of the variables (i.e., domain-based consistencies) or on the relations of the constraints (i.e., relation-based consistencies).

Local consistency properties are defined on sub-problems, of a given fixed size, of the original problem (e.g., two variables or two constraints). Because the size is fixed, local consistencies can be efficiently enforced. Below, we discuss the consistency properties used in this thesis.

2.4.1 Arc Consistency

Arc consistency (AC) is the most widely used local consistency property. A variable is AC if every value in its domain is compatible with at least one value in every neighboring variable. A CSP is AC if every variable is AC. A CSP is directionally arc consistent (DAC) with respect to a given fixed ordering if every variable is AC with respect to all variables that follow it in the ordering [Dechter and Pearl, 1988]. Arc consistency is extended to non-binary constraints in *generalized arc consistency* (GAC) [Waltz, 1975; Mackworth, 1977].

Definition 1. *Generalized Arc Consistency* (GAC) [Waltz, 1975; Mackworth, 1977].

- A value $d_i \in D_i$ is GAC with respect to $C_j \in \mathcal{C}$ where $x_i \in scope(C_j)$ iff there exists a valid tuple $\tau_j \in C_j$ such that $\pi_{x_i}(\tau_j) = d_i$.
- A variable $x_i \in \mathcal{X}$ is GAC with respect to $C_j \in \mathcal{C}$ where $x_i \in scope(C_j)$ iff all values $d_i \in D_i$ are GAC with respect to C_j.
- A constraint $C_j \in \mathcal{C}$ is GAC iff all variables $x_i \in scope(C_j)$ are GAC with respect to C_j.
- A CSP is GAC iff all constraints $C_j \in \mathcal{C}$ are GAC.

2.4.2 Pairwise Consistency

Pairwise consistency (PWC) [Gyssens, 1986] is local consistency property defined over the relations of the constraints of the CSP. The property ensures that every tuple of every relation has a supporting tuple in overlapping relations. PWC is analogous to arc consistency on the dual CSP.

Definition 2. *Pairwise Consistency* (PWC) [Gyssens, 1986]. A tuple $\tau_i \in C_i$ is PWC iff $\forall C_j \in \mathcal{C}$, $\exists \tau_j \in C_j$ such that $\pi_{scope(C_i) \cap scope(C_j)}(\tau_i) = \pi_{scope(C_i) \cap scope(C_j)}(\tau_j)$. We say that τ_i and τ_j are PWC and a PW-support of one another. A CSP is PWC iff every tuple of every constraint has a PW-support.

A CSP is PWC+GAC (full PWC) iff it is both PWC and GAC [Debruyne and Bessière, 2001].

2.4.3 Minimality

Minimality is a global consistency property that was first introduced for binary CSPs as the 'central problem' [Montanari, 1974]. It ensures that every tuple of the relation of a constraint appears in at least one solution to the problem.

Definition 3. *Minimal network* [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 argued that when a CSP has this property, a number of NP-hard queries can be answered in polynomial time, but also showed that:

1. Deciding whether or not a constraint network is minimal is NP-complete.
2. Finding a solution to a minimal network is also NP-complete [Gottlob, 2011],

Thus, proving earlier conjectures by Dechter and Pearl [1992].

In the literature, minimality corresponds to (relational) m-wise consistency [Gyssens, 1986] and relational $(1, m)$-consistency [Dechter and van Beek, 1997] where m is number of constraints in the CSP.

Although minimality was first introduced as a property of the constraints' relations, it is also used as a property of the variables' domains [Bayer *et al.*, 2007].

Domain minimality ensures that every value of every variable appears in at least one solution to the CSP. It has appeared in the literature under a variety of names:

- It is a special case of k-inverse consistency, denoted (1,$k-1$)-consistency with $k = n$ and n is the number of variables in the CSP [Freuder, 1985; Freuder and Elfe, 1996].
- Variable completability [Freuder, 1991].
- Domain m-wise consistency where m is number of constraints in the CSP [Mairy *et al.*, 2014].
- Global inverse consistency [Bessiere *et al.*, 2013].

2.5 Consistency Algorithms

Consistency algorithms are central to Constraint Processing. They enforce a given consistency property by removing, from the domains of a variable, the values that do not satisfy the domain-consistency property (respectively, from the relation of a constraint, the tuples that do not satisfy the relational-consistency property) because the removed values (respectively, tuples) are guaranteed to not appear in any solution.

There could be any number of algorithms for enforcing a given consistency property. Below, we review the mechanisms and algorithms that we use in this thesis to enforce consistency during search.

2.5.1 Lookahead

Consistency properties are often enforced during search as a *lookahead* strategy. One of the weakest lookahead strategies in terms of filtering is *forward checking* (FC): after

each assignment, FC ensures that all future variables that are adjacent to the instantiated variable are arc consistent with respect to the current assignment [Haralick and Elliott, 1980].

Currently, it is more common to enforce, after each instantiation, a given consistency property on the entire future subproblem in a *real-full lookahead* (RFL) strategy [Haralick and Elliott, 1980]. Typically, the maintained consistency property is a lightweight local consistency such as AC. We refer to the corresponding strategy as *maintaining arc consistency* (MAC) [Sabin and Freuder, 1997].

2.5.2 STR2 to Enforce Generalized Arc Consistency

We use the STR2 algorithm to enforce GAC during search [Lecoutre, 2011]. STR2 is an improved version of the Simple Tabular Reduction (STR) algorithm [Ullmann, 2007]. STR2 operates on constraints specified by tables of supporting tuples (i.e., positive tables), which are the type of constraints considered in this thesis. We choose STR2 because it filters not only the domains of the variables to enforce GAC but also updates accordingly the relevant relations. As a result, our relational consistency algorithms benefit from having the content of the relations and domains 'synchronized' without any additional overhead.

2.5.3 PW-AC2 to Enforce Pairwise Consistency (PWC)

For enforcing Pairwise Consistency, we use the PW-AC2 algorithm,[1] an optimization of the PieceWise Arc Consistency (PW-AC) algorithm [Samaras and Stergiou, 2005]. Both algorithms take advantage of the piecewise functional property of the equality

[1] PW-AC2 is an implementation by Anthony Schneider in the Stampede constraint solver implemented at the Constraint Systems Laboratory. Publication forthcoming.

constraints of a dual CSP, which partitions the tuples of a relation into equivalence classes of tuples that are handled as blocks and deleted together.

2.5.4 PerTuple, AllSol to Enforce Constraint Minimality

Enforcing minimality is known to be NP-complete [Gottlob, 2011]. Domain minimality algorithms have been proposed for interactive problem solving [Bayer *et al.*, 2007; Bessiere *et al.*, 2013]. Constraint minimality algorithms, restricted to subproblems, have been used for lookahead during search [Karakashian *et al.*, 2013].

We consider two algorithms for enforcing constraint minimality: PerTuple [Karakashian *et al.*, 2010] and AllSol [Karakashian, 2013]. Both algorithms operate by running a backtrack search on the dual CSP and identify the tuples that appear in a solution and mark them as minimal tuples. All tuples are initially unmarked.

PerTuple (Algorithm 1) systematically runs one backtrack search (Line 7) for *each* unmarked tuple in *each* relation, stopping after finding the first solution or determining that no solution exists. If no solution is found, the tuple is deleted (Line 9). If a solution is found, all the tuples in the solution are marked (Line 11)

AllSol (Algorithm 2) performs a single backtrack search, enumerating *all* solutions (Line 5). For each solution found, all involved tuples are marked as used (Line 7). After concluding the search, all unmarked tuples are deleted from the corresponding relations (Line 10).

A particularly effective refinement of AllSol is to backtrack whenever all the tuples in *both* the current path and the domains of all future variables are marked (as minimal). Indeed, continuing the search is useless and wasteful.

Algorithm 1: PerTuple($\mathcal{P}_D$) [Karakashian *et al.*, 2012]

```
   Input: P_D
   Output: Minimal Network of P_D
1  foreach R_i ∈ P_D do
2      foreach τ_i ∈ R_i do SetMark(τ_i, false)
3  foreach R_i ∈ P_D do
4      foreach τ_i ∈ R_i, do
5          if Marked(τ_i) = false then
6              Assign(R_i, τ_i)
               /* Backtrack search for a solution */
7              sol ←BTsearchOneSol(P_D)
8              if sol = false then
9                  Delete(τ_i)
10             else
11                 foreach τ_j ∈ sol do SetMark(τ_j, true)
```

Algorithm 2: AllSol($\mathcal{P}_D$) [Karakashian *et al.*, 2012]

```
   Input: P_D
   Output: Minimal Network of P_D
1  foreach R_i ∈ P_D do
2      foreach τ_i ∈ R_i do SetMark(τ_i, false)
3  sol ← false
4  while sol = false do
5      sol ←BTsearchNextSol(P_D)
6      if sol ≠ false then
7          foreach τ_i ∈ sol do SetMark(τ_i, true)
8  foreach R_i ∈ P_D do
9      foreach τ_i ∈ R_i do
10         if Marked(τ_i) = false then Delete(τ_i)
```

2.6 Cluster-Based Minimality

Because enforcing minimality can be prohibitively costly, Karakashian *et al.* propose to localize minimality to the clusters defined by a tree decomposition of the CSP [Karakashian *et al.*, 2013; Karakashian, 2013]. Below, we summarize their approach, as illustrated in Figure 2.10.

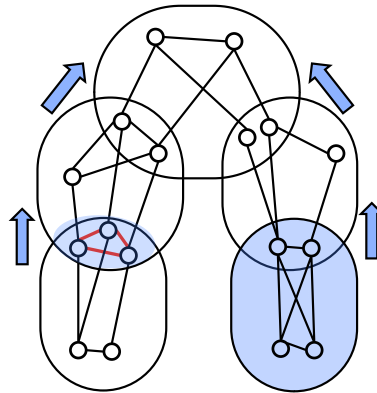


Figure 2.10: Cluster-based minimality

First, we generate a tree decomposition of the CSP as described in Section 2.3.5 while organizing the clusters of the decomposition along the MaxCliques ordering (in a total ordering) and generating projects constraints in separators between clusters to bolster constraint propagation along the tree. To enforce minimality, both at the preprocessing step and during search, we execute the following steps:

1. First, we update the constraints' relations by synchronizing them with the variables' domains by executing a selection operation.

2. We enforce minimality on the clusters along the MaxCliques ordering from the leaves the tree to its root, back and forth, until reaching a fixpoint.

3. When processing, we first synchronize its constraints with the constraints of neighboring clusters by enforcing directional pairwise consistency on the con-

straints in the cluster. Then, we enforce constraint minimality on the cluster.

4. After reaching a fixpoint, we update the domains of the variables by projecting the constraints on the domains of the variables.

We consider a few variations on this approach:

- We impose a time limit when enforcing minimality on cluster. When the limit is reached, the minimality algorithm is halted. As a result, we avoid getting stuck on a particularly difficult cluster but no longer guarantee cluster minimality. Note that when using PerTuple, early interruption may still cause some tuple deletions and, thus, yield some benefits. However, AllSol must reach completion before tuple can be deleted and the search is wasted.

- We consider strategies that choose to bypass certain clusters.

- Prior to processing the clusters, we enforce GAC over the *entire* problem because is it computationally cheap and strictly weaker than cluster-based minimality. This refinement is known to improve performance in general.

- As stated in Section 2.5.2, we use the STR2 algorithm to enforce GAC. Because STR2 filters the relations, we do not need to synchronize the relations with the domains (Step 1 above).

- After processing the clusters, we need to again enforce GAC over the entire problem, which is necessary given that we may skip some clusters or interrupt minimality on others, which may cause the CSP to no longer be GAC.

Summary

In this chapter, we provided background information relevant to the thesis, including general information about CSPs, their graphical representation, consistency properties and their algorithms as used in this thesis. Most importantly, we discussed how to enforce cluster-based minimality during search, which is the main concern of this thesis.

Chapter 3

Improving Minimality Algorithms

Now we focus on the two algorithms for enforcing constraint minimality, namely, PerTuple and AllSol. Both algorithms are based on searching the dual CSP and operate by finding and marking all tuples that participate in a solution, which may or may not require finding all the solutions. We introduce two new techniques for improving the performance of these algorithms:

1. *Unmarked-first ordering heuristic* (UF). A value-ordering heuristic appropriate for minimality algorithms.

2. *Dangle identification.* Identification of 'dangling' trees as tractable subproblems during search by application of the Graham reduction on the dual graph.

In order to further enhance the performance of PerTuple and AllSol, we also use three more known techniques that were *not* previously used in this context:

3. *dom/wdeg variable ordering.* Dom/wdeg is currently the most competitive variable ordering heuristic used during search [Boussemart *et al.*, 2004]. However, it was not used before in PerTuple or AllSol.

4. *Maintaining consistency.* The previous implementation and evaluation of PerTuple and AllSol is based on forward checking [Karakashian, 2013]. We propose to implement real-full lookahead.

5. *Minimal dual graph.* PerTuple and AllSol run search on the dual CSP. We propose to run them on a minimal dual CSP, corresponding to the dual CSP obtained after removing the redundant constraints.

We first discuss the improvements then empirically assess their impact.

3.1 Unmarked-First (UF) Ordering Heuristic

Our algorithms for enforcing constraint minimality, namely, PerTuple and AllSol, need to guarantee that every tuple in a relation appears in some solution to the dual CSP. To this end, whenever we find a solution, we mark all the tuples that appear in it. Because all tuples must eventually be checked, it seems beneficial to prioritize instantiating unmarked tuples over already marked ones. This rationale inspires our value-ordering heuristic '*Unmarked First*' (UF), which selects first, for instantiation, the unmarked tuples. Importantly, the heuristic is applicable to computing both domain minimality and constraint minimality.

For an efficient processing, we implement the domain of a dual variable (i.e., the set of tuples of a relation) as a reversible sparse-set [le Clément de Saint-Marcq *et al.*, 2013; Demeulenaere *et al.*, 2016].[1] This data structure allows constant-time operations on the domains of the dual variables (e.g., delete a tuple, restore a group of previously deleted tuples, check whether or not a tuple is alive). In addition, we use intrusive lists to handle iteration over the current domain and the sets of tuples re-

[1]The data structures are implemented by Anthony Schneider in Stampede, the constraint solver of the Constraint Systems Laboratory.

moved during search (i.e., reductions [Prosser, 1993]). However, and as a consequence of these structures, the tuples in any given domain are stored in an arbitrary order, which is affected by the sequence of deletion and restoration operations. In order to efficiently implement the UF heuristic, we use two additional bookkeeping structures for the domain of each dual variable:

1. A sparse-set to track all unmarked tuples in the domain (i.e., regardless of whether or not the tuple is alive).

2. For each variable, we split the intrusive list representing its current domain and each list representing one of its reductions into two intrusive lists: one for marked tuples and one for unmarked tuples. Within these lists, the order of tuples is arbitrary. Separating these lists allows us, upon backtracking, to update the current domain by merging the marked (respectively, unmarked) tuples of the relevant reduction and of the current domain.

During search, when instantiating a dual variable, we choose in priority an unmarked tuple over a marked one.

3.2 Dangle Identification

We propose to dynamically identify and remove, during search, all dangling tree structures in order to reduce the search space and effort. Whenever a variable is instantiated, it is as if the vertex representing the variable is erased from the hyperedges in the constraint network induced by the unassigned (i.e., future) variables. As search proceeds, this network becomes increasingly sparser and dangling subtrees may appear in the network. These dangling subtrees can be exploited in three ways:

1. When, upon dangle identification, a single pass of directional arc consistency (DAC) [Dechter and Pearl, 1988] from the leaves of each subtree to its root finds a subtree to be inconsistent, we can force backtracking.

2. It DAC successfully completes, the subtrees are guaranteed to have a solution provided the current path has a solution. It can thus be safely removed from the search process, thus reducing the search space and potential thrashing.

3. If the search successfully completes the current path, enforcing DAC from the root to the leaves of each dangling subtree effectively guarantees the minimality of the subtree. Consequently, all the values remaining in the domains of its variables can be 'collectively' marked as minimal, thus reducing the number of operations in PerTuple or AllSol.

The procedures IdentifyDangles (Algorithm 3) and RestoreDangles (Algorithm 4) handle the identification and restoration of dangles during search, respectively. They use the following parameters as input: *dangleVs*, *dangleEs*, *unremoved*, and *level*. Each structure *dangleVs* and *dangleEs* is a vector indexed by search level. For each level, the vector stores an ordered list of vertices (for the former) and edges (for the latter) of the dual graph. These structures track the dangles identified at each level of the current search path. The structure *unremoved* tracks the set of dual variables that are neither instantiated nor removed by dangle identification at the current level of the search, which is specified by the parameter *level*.

IdentifyDangles (Algorithm 3) is executed both at preprocessing and as soon as a variable is selected for assignment but before it is instantiated in order to avoid repeating the operation for every value assignment. IdentifyDangles performs the Graham reduction on the graph [Maier, 1983]. This reduction alternates between identifying degree-one vertices (Lines 2–5) and removing these vertices (Lines 7–13)

until no further degree-one vertices are found (Line 6). Every vertex v removed from the graph is added to the list at $dangleVs[level]$. The edge that connects the vertex to the graph is also removed from the graph and added to the list at $dangleEs[level]$. Because the vertices and edges are added in the order in which they are identified, they are ordered from the leaves to the root. RemoveV (Line 13) modifies the graph removing the corresponding vertex and updating the degree of its neighbor.

Algorithm 3: IdentifyDangles(*unremoved*,*dangleVs*,*dangleEs*,*level*)

```
   Input: unremoved: set of dual variables
          dangleVs, dangleEs: vectors of lists of dual variables/edges
          level: integer
1  while true do
2      degOneVs ← ∅
3      foreach v ∈ unremoved do
4          if degree(v) ≤ 1 then
5              degOneVs ← degOneVs ∪ {v}
6      if degOneVs = ∅ then break
7      foreach v ∈ degOneVs do
8          if degree(v) = 1 then
9              neighbor ← Neighbor(v)
10             Push(⟨v,neighbor⟩,dangleEs[level])
11         Push(v,dangleVs[level])
12         unremoved ← unremoved \ {v}
13         RemoveV(v)
```

RestoreDangles (Algorithm 4) is executed upon backtracking from *level* and undoes the separation of dangles at *level*. RestoreDangles iterates over the vertices stored in $dangleVs[level]$. It calls AddV to add a vertex back to the graph and update the degree its neighboring vertex. The lists stored in *dangleEs* and *dangleVs* for *level* are then cleared.

Algorithm 4: RestoreDangles(*unremoved*,*dangleVs*,*dangleEs*,*level*)

Input: *unremoved*: set of dual variables
dangleVs, *dangleEs*: vectors of lists of dual variables/edges
level: integer

1 **foreach** $v \in dangleVs[level]$ **do**
2   AddV(v)
3     $unremoved \leftarrow unremoved \cup \{v\}$
4 $dangleEs[level] \leftarrow \emptyset$
5 $dangleVs[level] \leftarrow \emptyset$

3.3 Dom/wdeg Variable Ordering

The original implementations of AllSol and PerTuple used the dom/deg dynamic variable ordering heuristic [Karakashian *et al.*, 2010; Karakashian *et al.*, 2013; Karakashian, 2013]. The more recent dom/wdeg ordering heuristic is known to have, in general, the best performance [Boussemart *et al.*, 2004]. Adapting dom/wdeg to the search on dual CSPs requires assigning weights to the equality constraints between the dual variables (i.e., the dual graph's edges). Starting from a weight of one for each equality constraint, we increase by one the weight of an equality constraint whenever it yields a domain wipeout in AllSol or PerTuple during lookahead.

Further, when enforcing minimality on the clusters of a tree decomposition during lookahead, we maintain, for each cluster, two sets of weights for the equality constraints in the cluster: one for PerTuple and one for AllSol. Each set persists across calls to a given minimality algorithm.

Note that the weights of the equality constraints as used when enforcing minimality on a cluster do not interfere with the weights of the constraints in the search for solving the CSP.

3.4 Maintaining Consistency

Previous implementations of PerTuple and AllSol use only forward checking (FC) [Karakashian, 2013]. The current state of the art recommends using the more aggressive real-full lookahead (RFL), that is, maintaining consistency.

Lookahead on the dual CSP enforces pairwise consistency (analogous to arc consistency on the dual CSP), either partially (for FC) or completely (for RFL). However, each equality constraint in the dual CSP is piecewise functional and partitions the two relations to which it applies into coarse blocks of equivalent tuples [Samaras and Stergiou, 2005; Schneider *et al.*, 2014]. Our lookahead algorithms DualFC and DualRFL for lookahead on the dual CSP exploit this property of the equality constraints.[2] We compute once the partitions of the domains of the dual variables and store them for use in our algorithms.

Below, we first describe DualFC and DualRFL, then introduce DualDangleFC as another FC algorithm for the dual CSP that operates with dangle identification. Our algorithms use the following 'accessors' and helper functions:

- Neighbors(*dualVar*) gives the list dual variables adjacent to *dualVar* in the dual graph.
- GetDomain(*dualVar*) gives the list of tuples in the domain of *dualVar*.
- GetTheLivingTuple(*dualVar*) gives the (unique) living tuple in the domain of *dualVar*.
- GetCoarseBlock($dualVar_a$, $dualVar_b$, *tuple*) gives the set of shared CSP variables by $dualVar_a$ and $dualVar_b$ (i.e., the subscope) and the list of tuples of $dualVar_a$ that are consistent with *tuple* of $dualVar_b$.

[2] DualFC is developed in collaboration with Anthony Schneider.

- WasAlive(*domain*, *tuple*) returns true if *tuple* was alive after the previous instantiation.

- EnqueueFrom(*queue*, $dualVar_a$, *dualVars*) takes three parameters: the propagation queue, a dual variable, and a set of dual uninstantiated variables *dualVars*. It pushes in *queue* all ordered pairs $(dualVar_b, dualVar_a)$ where $dualVar_b \in dualVars \cap$ Neighbors$(dualVar_a)$.

3.4.1 The DualFC Algorithm

We implement forward checking on the dual graph with the following four algorithms: DualFC (Algorithm 5), ReviseFC (Algorithm 6), SaveIntersection (Algorithm 7), and RemoveDifference (Algorithm 8).[3]

Algorithm 5: DualFC(*assignedCon*,*unassigned*)

Input: *assignedCon*: dual variable
unassigned: set of dual variables
Output: *consistent*: Boolean

```
1 foreach v ∈Neighbors(assignedCon) do
2     if v ∈ unassigned then
3         ⟨consistent, filtered⟩ ←ReviseFC(v,assignedCon)
4     if consistent = false then break
5 return consistent
```

After instantiating a dual variable, DualFC (Algorithm 5) is called on the assigned constraint *assignedCon* and all future constraints *unassigned*. It calls ReviseFC on the unassigned neighbors (Line 3). ReviseFC (Algorithm 6) exploits the coarse blocks of the constraint on which it is called and the fact that *againstCon* has a single living tuple (i.e., the assigned tuple). Thus, only the compatible coarse

[3] Note that we enforce forward checking in DualFC (Algorithm 5), in the original PerTuple algorithm [Karakashian *et al.*, 2010], and in the PerFB algorithm [Schneider *et al.*, 2014] in qualitatively different ways.

Algorithm 6: ReviseFC(*reviseCon*,*againstCon*)

```
   Input: reviseCon, againstCon: dual variables
   Output: consistent, filtered: Booleans
1  reviseDomain ←GetDomain(reviseCon)
2  initialSize ← |reviseDomain|
3  tuple ←GetTheLivingTuple(againstCon)
4  ⟨subscope, block⟩ ←GetCoarseBlock(reviseCon,againstCon,tuple)
5  if |block| ≤ |reviseDomain| · |subscope| then
6  |   SaveIntersection(reviseDomain,block)
7  else RemoveDifference(reviseDomain,block,subscope)
8  consistent ← |reviseDomain| ≠ 0
9  filtered ← |reviseDomain| < initialSize
10 return ⟨consistent, filtered⟩
```

Algorithm 7: SaveIntersection(*domain*,*block*)

```
   Input: domain: domain of dual variable
          block: coarse block
1  domain ← ∅
2  foreach tuple ∈ block do
3  |   if WasAlive(domain,tuple) then
4  |   |   domain ← domain ∪ {tuple}
```

Algorithm 8: RemoveDifference(*domain*,*block*,*subscope*)

```
   Input: domain: domain of dual variable
          block: coarse block
          subscope: set of shared CSP variables
1  blockTuple ← block[0]
2  foreach tuple ∈ domain do
3  |   foreach var ∈ subscope do
4  |   |   if π_var(tuple) ≠ π_var(blockTuple) then
5  |   |   |   domain ← domain \ {tuple}
6  |   |   |   break
```

block in the *reviseCon* needs to be saved (i.e., $reviseDomain \cap block$). For the sake of efficiency, we implement this intersection operation in two different ways (Lines 6 and 7):

1. If the size of the block is smaller than that of the current domain (multiplied by the number of CSP variables shared by the two constraints), we call, in Line 6, SaveIntersection (Algorithm 7). This algorithm deletes the entire domain (in constant time). Then, it loops through the block and restores that tuples that were previously alive. (Tuple lookup can be done in a constant using our domain structures.) Thus, this entire operation runs in time $O(|block|)$.

2. If the block size is larger than that of the current domain (multiplied by the number of CSP variables shared by the two constraints), we call, in Line 7, RemoveDifference (Algorithm 8). This algorithm loops over the current domain, removing, from the domain, the tuples not in the block. The block consists of tuples with the same values for the shared CSP variables. In order to determine membership in a block, we check the shared CSP variables have the same value in the two tuples (Line 4). Thus, this entire operation runs in time $O(|domain| \cdot |subscope|)$.

If ReviseFC results in an empty domain, we interrupt DualFC and immediately return *false*, triggering a backtrack in the search procedure.

3.4.2 The DualRFL Algorithm

Our real-full lookahead procedure is similar to the AC-3 algorithm [Mackworth, 1977]. We implement it with the following five algorithms: DualRFL (Algorithm 9), ReviseRFL (Algorithm 10), CheckForDeadBlock (Algorithm 11), SearchBlockForLivingTuple (Algorithm 12), and SearchLivingForBlockTuple (Algorithm 13).

DualRFL (Algorithm 9) uses a *queue* to track dual edges that must be revised. *queue* begins empty (Line 2) and is filled with all outgoing edges from *assignedCon*

Algorithm 9: DUALRFL(*assignedCon*,*unassigned*)

Input: *assignedCon*: dual variable
unassigned: set of dual variables
Output: *consistent*: Boolean

```
1 consistent ← true
2 queue ← ∅
3 EnqueueFrom(queue,assignedCon,unassigned)
4 while queue ≠ ∅ do
5 |   ⟨reviseCon, againstCon⟩ ←Pop(queue)
6 |   ⟨consistent, filtered⟩ ←ReviseRFL(reviseCon,againstCon)
7 |   if consistent = false then break
8 |   if filtered = true then EnqueueFrom(queue,reviseCon,unassigned)
9 return consistent
```

to dual variables in *unassigned* (Line 3). Lines 4–8 loop through the *queue* until it is empty. We pop an edge from the *queue* (Line 5) and call REVISERFL on the edge (Line 6). The edges are directed: the first vertex (*reviseCon*) is revised with respect to the second vertex (*againstCon*). If REVISERFL detects inconsistency, we immediately return *false* (Line 7). If REVISERFL filters a constraint, outgoing arcs from that constraint are added to the queue (Line 8). At preprocessing, we initialize the queue with all directed edges of the dual CSP and start DUALRFL from Line 4.

REVISERFL (Algorithm 10) deletes tuples in the domain of *reviseCon* that are inconsistent with respect to *againstCon*. It uses several shortcuts enabled by the coarse blocks. First of all, if the domain of *againstCon* (the constraint we are revising against) has a single tuple, it is cheaper to perform REVISEFC (Algorithm 6) rather than checking the consistency of each tuple in *reviseCon* (Lines 2–3). Otherwise, we loop through every tuple of *reviseDomain* (Lines 7–14). We identify the tuple's supporting *block* in *againstCon* (Line 8). We then determine if *block* has any living tuples remaining. The structure *blockIsDead* is a map indexed by coarse blocks and used to track whether a block is alive or dead, avoiding repeated checks. We

Algorithm 10: REVISERFL(*reviseCon*,*againstCon*)

Input: *reviseCon*, *againstCon*: dual variables
Output: *consistent*, *filtered*: Booleans

```
1  againstDomain ←GetDomain(againstCon)
2  if |againstDomain| = 1 then
3      return ReviseFC(reviseCon,againstCon)
4  reviseDomain ←GetDomain(reviseCon)
5  initialSize ← |reviseDomain|
6  blockIsDead ← ∅
7  foreach tuple ∈ reviseDomain do
8      ⟨subscope,block⟩ ←GetCoarseBlock(againstCon,reviseCon,tuple)
9      if blockIsDead[block] = NULL then
10         if CheckForDeadBlock(againstDomain,block,subscope) = true
            then
11             blockIsDead[block] ← true
12         else blockIsDead[block] ← false
13     if blockIsDead[block] = true then
14         reviseDomain = reviseDomain \ {tuple}
15 consistent ← |reviseDomain| ≠ 0
16 filtered ← |reviseDomain| < initialSize
17 return (consistent,filtered)
```

initialize *blockIsDead* to be empty (Line 6) and populate it with entries mapping a coarse block to a Boolean (Lines 11 and 12). If no entry is found for *block* (Line 9), CHECKFORDEADBLOCK (Algorithm 11) verifies the existence of a living tuple in the block (Line 10). Whenever, in the loop (Lines 7–14), the call to GETCOARSEBLOCK at Line 8 returns a block 'seen' during the loop, the tuples in the subsequent calls are saved or deleted with a constant-time lookup (Lines 13–14).

CHECKFORDEADBLOCK (Algorithm 11) determines whether *block* is *dead* (i.e., contains no living tuples). Similarly to REVISEFC, two methods perform this operation (Lines 2 and 3) for the sake of performance. These two methods are SEARCHBLOCKFORLIVINGTUPLE (Algorithm 12) and SEARCHLIVINGFORBLOCKTUPLE (Algorithm 13):

1. If the size of the block is smaller than that of the current domain (multiplied by the number of CSP variables shared by the two constraints), we call, in Line 2 of CheckForDeadBlock, SearchBlockForLivingTuple (Algorithm 12). This algorithm iterates over *block* to find an alive tuple in *domain* (Lines 5–14). To speed this process, we use the structure *lastAlive*. This structure maps the 'id' of a coarse block to the index of the last tuple found alive in the block. Initially, the *lastAlive* entries are empty (Line 2) and the default *aliveIndex* value, zero, is used (Line 3). The first for-loop iterates over the tuples in *block*, beginning at *aliveIndex* and ending at the last tuple in the block (Lines 5–9). The second for-loop (Lines 10–14) iterates over the first portion of *block*, from the first tuple up until the *aliveIndex*. If a living tuple is found, the algorithm records the tuple index in *lastAlive* and returns *true*. If not, the algorithm returns *false* and the block is identified as dead. The whole operation runs in time $O(|block|)$.

Algorithm 11: CheckForDeadBlock(*domain*, *block*, *subscope*)

Input: *domain*: domain of dual variable
block: coarse block
subscope: set of shared CSP variables
Output: *dead*: Boolean

```
1 if |block| ≤ |domain| · |subscope| then
2 |  dead ← not SearchBlockForLivingTuple(domain,block)
3 else dead ← not SearchLivingForBlockTuple(domain,block,subscope)
4 return dead
```

2. If the block size is larger than that of the current domain (multiplied by the number of CSP variables shared by the two constraints), we call, in Line 3 of CheckForDeadBlock, SearchLivingForBlockTuple (Algorithm 13). This algorithm iterates over the tuples in *domain* (Lines 3–11) and determines whether or not they are members of *block* (Lines 5–8). Similarly to RemovedD-

Algorithm 12: SearchBlockForLivingTuple(*domain*, *block*)

```
   Input: domain: domain of dual variable
          block: coarse block
   Output: living: Boolean
1  living ← false
2  if lastAlive[block] = NULL then
3  |  aliveIndex ← 0
4  else aliveIndex ← lastAlive[block]
5  for i ← aliveIndex to |block| − 1 do
6  |  if block[i] ∈ domain then
7  |  |  lastAlive[block] ← i
8  |  |  living ← true
9  |  |  return living
10 for i ← 0 to aliveIndex − 1 do
11 |  if block[i] ∈ domain then
12 |  |  lastAlive[block] ← i
13 |  |  living ← true
14 |  |  return living
15 return living
```

Algorithm 13: SearchLivingForBlockTuple(*domain*, *block*, *subscope*)

```
   Input: domain: domain of dual variable
          block: coarse block
          subscope: set of shared CSP variables
   Output: living: Boolean
1  living ← false
2  blockTuple ← block[0]
3  foreach tuple ∈ domain do
4  |  match ← true
5  |  foreach var ∈ subscope do
6  |  |  if π_var(tuple) ≠ π_var(blockTuple) then
7  |  |  |  match ← false
8  |  |  |  break
9  |  if match = true then
10 |  |  living ← true
11 |  |  return living
12 return living
```

IFFERENCE (Algorithm 8), we determine block membership by checking whether the shared CSP variables have the same value in the two tuples (Line 6). If the match is successful, then the tuple belongs to *block* and the block as alive (Lines 9–11). The time complexity of SEARCHLIVINGFORBLOCKTUPLE is $O(|domain| \cdot |subscope|)$.

3.4.3 The DUALDANGLEFC Algorithm

DUALRFL enforces AC on all the dual variables in the dual graph, including those identified as dangles by IDENTIFYDANGLES (Algorithm 3). However, when using dangle identification with forward checking, a specialized lookahead algorithm is needed. We implement this procedure in the following three algorithms: DUALDANGLEFC (Algorithm 14), UPCURRENTDANGLES (Algorithm 15), and DOWNALLDANGLES (Algorithm 16). DUALDANGLEFC (Algorithm 14) takes five inputs:

1. *assignedCon* the dual variable just instantiated
2. *unassigned* the set of uninstantiated dual variables
3. *unremoved* the set of dual variables that are neither instantiated nor removed by dangle identification at the current level of the search
4. *dangleEs* the vector of lists of edges identified as dangles by IDENTIFYDANGLES (Algorithm 3)
5. *level* the current search level

DUALDANGLEFC first executes DUALFC (Algorithm 5) in Line 1. If it does not detect an inconsistency, it calls UPCURRENTDANGLES to enforce directional arc consistency from the leaves to the root on all newly identified dangles (Line 3). Finally,

---

Algorithm 14: DUALDANGLEFC(*assignedCon*,*unassigned*,*unremoved*,*dangleEs*,*level*)

Input: *assignedCon*: dual variable
unassigned, *unremoved*: sets of dual variables
dangleEs: vector of lists of dual edges
level: integer
Output: *consistent*: Boolean

1 *consistent* ←DUALFC(*assignedCon*,*unassigned*)
2 **if** *consistent* = *false* **then return** *consistent*
3 *consistent* ←UPCURRENTDANGLES(*dangleEs*,*level*)
4 **if** *consistent* = *consistent* **then return** *consistent*
5 **if** *unremoved* = ∅ **then**
6   *consistent* ←DOWNALLDANGLES(*dangleEs*)
7 **return** *consistent*

---

if only dangles are left unassigned (i.e., we have reached the end of the search path), we call DOWNALLDANGLES to enforce directional arc consistency on all dangles from the root to the leaves (Lines 5–6).

UPCURRENTDANGLES (Algorithm 15) takes, as input, *dangleEs* and *level*. The for-loop in Lines 2–5 iterates over the dangle edges at the current level (i.e., *dangleEs*[*level*]) and executes REVISERFL on the two dual variables of each edge revising the second dual variable of the edge with respect to the first (i.e., revise the variable closer to the root). Upon detecting inconsistency, it immediately returns *false* (Line 5). UP-CURRENTDANGLES enforces directional arc consistency along a width-one ordering, which ensures that the dangle can be solved backtrack free.

DOWNALLDANGLES (Algorithm 16) takes, as input, *dangleEs*. The outer for-loop iterates over the search levels in reverse order, that is, from the deepest to the shallowest (Lines 2–6). The inner for-loop iterates over the dangle edges identified at *level* in reverse of the order in which they were identified (Lines 3–6). It executes REVISERFL on each edge (Line 4) revising the first dual variable of the edge with respect to the second (i.e., revise the variable further from the root).

Algorithm 15: UpCurrentDangles(*dangleEs*,*level*)

Input: *dangleEs*: vector of lists of dual edges
level: integer
Output: *consistent*: Boolean

```
1 consistent ← true
2 for i ← 0 to |dangleEs[level]| − 1 do
3 |   ⟨againstCon, reviseCon⟩ ← dangleEs[level][i]
4 |   ⟨consistent, filtered⟩ ←ReviseRFL(reviseCon,againstCon)
5 |_  if consistent = false then return consistent
6 return consistent
```

Algorithm 16: DownAllDangles(*dangleEs*)

Input: *dangleEs*: vector of lists of dual edges
Output: *consistent*: Boolean

```
1 consistent ← true
2 for level ← |dangleEs| − 1 downto 0 do
3 |   for i ← |dangleEs[level]| − 1 downto 0 do
4 |   |   ⟨reviseCon, againstCon⟩ ← dangleEs[level][i]
5 |   |   ⟨consistent, filtered⟩ ←ReviseRFL(reviseCon,againstCon)
6 |   |_  if consistent = false then return consistent
  |_
7 return consistent
```

By running UpCurrentDangles upon identifying new dangles and DownAllDangles at the conclusion of the current search path, we ensure directional arc consistency is enforced in both directions. Thus, all living tuples are minimal and are marked as such.

3.5 Minimal Dual Graph

To enforce minimality, PerTuple and AllSol conduct search on the dual graph. Janssen *et al.* [1989] and Dechter [2003] observe that, in the dual graph, an edge between two vertices is *redundant* if there exists an alternate path between the two vertices such that the shared CSP variables appear in every vertex in the path. Re-

dundant edges can be removed without affecting the set of solutions. A *minimal dual graph* is one with no such redundant edges. Many minimal graphs may exist, but all are guaranteed to have the same number of edges.

Janssen *et al.* introduce an efficient algorithm for computing a minimal dual graph [1989]. In order to yield different minimal dual graphs, we propose to add to this algorithm various edge-selection heuristics when the algorithm connects two connected components. As a first heuristic, we choose to prioritize using the edge that minimizes the sum of the degrees of the two vertices it connects. We call this heuristic MinDeg.[4]

3.6 Correctness

In this section, we prove the correctness of several of our proposed improvements.

Theorem 2. *The search procedure conducted with forward checking maintains correctness when performed on any minimal dual graph.*

Proof. By definition, any minimal dual is equivalent to the original dual (i.e., both have the same set of solutions). Thus, forward checking (or any complete search procedure) will maintain correctness when performed on any minimal dual graph. □

Although correctness is maintained, the performance of search may be significantly altered by the use of a minimal dual graph during forward checking. By removing edges (equality constraints), wipeouts that would have otherwise been detected early may be delayed until remaining dual variables along the path of equality constraints have been assigned.

[4] The same heuristic is used in the previous work for computing minimal dual graphs [Karakashian *et al.*, 2010; Karakashian *et al.*, 2013; Karakashian, 2013].

Theorem 3. *When all dual variables are either instantiated or identified as dangles, if the dangles are arc consistent, every living tuple participates in a solution (i.e., is minimal).*

Proof. After the search has completed, all uninstantiated dual variables are identified as dangles. Consequently, the dual graph is a forest of independent, arc-consistent trees (i.e., of width one). An arc-consistent CSP of width one is solvable backtrack free [Freuder, 1982]. Thus, any living tuple can be extended to some solution. □

Because of Theorem 3, upon reaching this point in the search of AllSol and PerTuple, we can safely mark all living tuples without explicitly enumerating the solutions in which they appear.

Theorem 4. *After all dual variables are either instantiated or identified as dangles,* DualDangleFC *guarantees that the dangles are arc consistent.*

Proof. All uninstantiated dual variables are identified as dangles. Consequently, the graph is a forest of independent trees (i.e., of width one). In order to enforce AC on such a graph, it is sufficient to enforce DAC both in the forward and reverse directions of a width-one ordering, d [Dechter and Pearl, 1988]. The width-one ordering, d, is obtained by traversing the structure *dangleVs* in 'reverse' of the order in which it was built:

1. From the deepest level to the shallowest level

2. At a given level, from the last identified dual variable to the first

By construction, each of the dual variables stored in *dangleVs* is a dangle because, at the identification step, it had no more than one neighbor. Thus, when taken in reverse order of identification, each is guaranteed to have no more than one parent in

the ordering. By the construction of *dangleEs*, each edge connects a child in ordering d either to its parent in the ordering or to an instantiated dual variable. When traversing *dangleEs* in reverse order as described above, we necessarily encounter the edge between a dual variable and its parent before the edges between a dual variable and its children.

Now, we show that UpCurrentDangles enforces DAC along the ordering d and that DownAllDangles enforces DAC along the reverse of d. UpCurrentDangles is executed in stages as dangles are identified at each level. In order to enforce DAC along an ordering, beginning with the last variable, each variable revises the domain of its parent variable. Because d is ordered in reverse of *dangleEs*, following the original order of *dangleEs* and revising the second dual variable in each ordered pair results in revising the parent dual variables in the correct order. The correctness of this process is not affected by splitting it across levels of search.

Upon reaching a point in search where all uninstantiated dual variables are dangles, DualDangleFC executes UpCurrentDangles on the final level of dangles, thus, enforcing DAC along d. Then, DownAllDangles is executed, proceeding backwards through all of *dangleEs* and revising edges in the opposite direction. This process enforces DAC along the reverse ordering of d. Because all dangles are of width one, the resulting problem is guaranteed arc consistent. □

3.7 Experimental Evaluation

We conduct a series of experiments to evaluate the impact of our improvements on the performance of both AllSol and PerTuple. In order to efficiently and completely observe the impact of each change as well as their interactions, we set up our experiments in a factorial design [Box *et al.*, 1978].

3.7.1 Setup

We consider six factors, each of two levels:

1. Algorithm: AllSol, PerTuple
2. Weighted ordering heuristic: on, off
3. Unmarked first: on, off
4. Real-full lookahead: on, off
5. Minimal dual graph: on, off
6. Dangle identification: on, off

We test all 64 configurations of the factors. To denote these configurations, we use the following scheme (see Table 3.1). A configuration is first specified by its algorithm name in full, followed by a dash and a sequence of five upper and lowercase letters (e.g., AllSol-UDwrM). Each letter corresponds to one of the remaining five factors. Uppercase indicates the factor is 'on' while lowercase is 'off'.

Table 3.1: The six tested factors

Factor	**Abbrev.**	**Level**		**Reference**
		-	+	
Algorithm	Alg	PerTuple	AllSol	Section 2.5.4
Unmarked First	UF	off: u	on: U	Section 3.1
Dangle Identification	DI	off: d	on: D	Section 3.2
Weighted variable ordering	Weight	off: w	on: W	Section 3.3
Real-Full Lookahead	RFL	off: r	on: R	Section 3.4
Minimal Dual	MinD	off: m	on: M	Section 3.5

In this thesis, we advocate to enforce minimality for RFL locally, on the clusters of of a tree decomposition [Karakashian *et al.*, 2013]. For this reason, and in order to evaluate the impact of the above listed factors, we execute the 64 configurations of our algorithms on single clusters taken from tree decompositions of full CSP instances. We randomly sample 10 clusters from decompositions of all the instances of a set of 175 benchmarks from the XCSP library, which includes a mix of binary and non-binary random, quasi-random, academic, Boolean, patterned, and real-world instances.[5] This sampling yields 1,684 total clusters.[6] We executed each of the 64 configurations on all 1,684 clusters for a total of 107,776 runs.

We run our experiments on a computer cluster of Intel Xeon E5-2670 2.60 GHz processors. Each run is allocated 30 minutes and 12GB of memory. To account for load variations on the cluster computer, we measure instruction count and convert it to runtime using a standardized measure of instructions per cycle and clock speed. For each run, we first enforce GAC prior to enforcing minimality.

3.7.2 Results

In this section, we report the results, describing our tables and figures. We delay the discussion of these results to Section 3.7.3.

3.7.2.1 Performance of the 64 Configurations

In Table 3.2, we summarize the performance of each of the 64 configurations. In the table, we report, for each configuration: the level of each of the six factors; the number of instances completed (out of the 1,684 instances in total); the average CPU time;

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

[6] When the primal graph of a CSP instance is complete, then a tree decomposition has a single cluster and the entire instance is selected. When such a benchmark has less than ten instances, we end up with fewer total clusters than expected (i.e., 1,684 instead of 1,750).

Table 3.2: Results summary for all tested configurations

Configuration	Alg	UF	DI	Weight	RFL	MinD	Instances completed	Average time (ms) by one	Average time (ms) by all	Avg NV by all
PerTuple-udwrm	-	-	-	-	-	-	1,187	300,480.2	2,190.7	28,515.1
PerTuple-udwrM	-	-	-	-	-	+	715	898,788.8	19,938.3	5,392,161.7
PerTuple-udwRm	-	-	-	-	+	-	1,149	383,225.9	5,117.3	15,123.2
PerTuple-udwRM	-	-	-	-	+	+	1,221	285,803.7	5,198.7	14,975.6
PerTuple-udWrm	-	-	-	+	-	-	1,239	237,906.5	2,183.4	25,602.2
PerTuple-udWrM	-	-	-	+	-	+	816	774,711.5	6,707.6	1,539,655.7
PerTuple-udWRm	-	-	-	+	+	-	1,151	380,338.7	5,088.8	15,140.4
PerTuple-udWRM	-	-	-	+	+	+	1,228	276,859.2	5,171.0	14,974.2
PerTuple-uDwrm	-	-	+	-	-	-	1,184	305,828.0	1,077.8	13,098.0
PerTuple-uDwrM	-	-	+	-	-	+	739	866,740.7	11,227.3	1,377,239.7
PerTuple-uDwRm	-	-	+	-	+	-	1,146	382,318.0	2,442.3	3,996.6
PerTuple-uDwRM	-	-	+	-	+	+	1,242	241,890.7	1,803.2	1,032.6
PerTuple-uDWrm	-	-	+	+	-	-	1,228	248,779.0	1,075.6	12,362.0
PerTuple-uDWrM	-	-	+	+	-	+	827	760,624.7	5,401.7	594,037.4
PerTuple-uDWRm	-	-	+	+	+	-	1,148	379,569.7	2,448.3	4,015.4
PerTuple-uDWRM	-	-	+	+	+	+	1,246	235,177.2	1,804.8	1,047.1

Table 3.2: Results summary for all tested configurations (continued)

Configuration	Alg	UF	DI	Weight	RFL	MinD	Instances completed	Average time (ms)		Avg NV
								by one	by all	by all
PerTuple-Udwrm	-	+	-	-	-	-	1,245	233,532.9	2,041.7	21,553.2
PerTuple-UdwrM	-	+	-	-	-	+	724	887,317.6	18,081.8	4,431,705.5
PerTuple-UdwRm	-	+	-	-	+	-	1,176	332,286.2	3,614.1	10,979.0
PerTuple-UdwRM	-	+	-	-	+	+	1,239	243,753.6	3,468.9	10,865.0
PerTuple-UdWrm	-	+	-	+	-	-	**1,276**	**193,436.9**	2,041.7	20,447.6
PerTuple-UdWrM	-	+	-	+	-	+	834	751,406.9	5,894.8	1,210,568.4
PerTuple-UdWRm	-	+	-	+	+	-	1,174	332,982.0	3,610.3	10,982.6
PerTuple-UdWRM	-	+	-	+	+	+	1,249	232,511.8	3,458.8	10,873.9
PerTuple-UDwrm	-	+	+	-	-	-	1,237	241,425.3	**1,004.2**	11,504.8
PerTuple-UDwrM	-	+	+	-	-	+	753	852,844.7	12,017.8	1,517,151.2
PerTuple-UDwRm	-	+	+	-	+	-	1,179	330,673.2	1,865.3	2,928.8
PerTuple-UDwRM	-	+	+	-	+	+	1,254	218,715.3	1,587.1	**878.0**
PerTuple-UDWrm	-	+	+	+	-	-	1,268	201,358.0	1,006.1	11,291.1
PerTuple-UDWrM	-	+	+	+	-	+	842	739,499.9	4,218.9	471,417.5
PerTuple-UDWRm	-	+	+	+	+	-	1,177	330,241.0	1,872.8	2,954.3
PerTuple-UDWRM	-	+	+	+	+	+	1,264	210,453.8	1,587.0	890.2

Table 3.2: Results summary for all tested configurations (continued)

Configuration	Alg	UF	DI	Weight	RFL	MinD	Instances completed	Average time (ms) by one	Average time (ms) by all	Avg NV by all
AllSol-udwrm	+	-	-	-	-	-	813	782,840.6	8,940.5	3,476,098.6
AllSol-udwrM	+	-	-	-	-	+	680	943,504.4	32,553.0	12,608,059.3
AllSol-udwRm	+	-	-	-	+	-	1,082	454,824.1	1,946.1	554,179.6
AllSol-udwRM	+	-	-	-	+	+	1,122	397,917.4	2,274.5	556,817.8
AllSol-udWrm	+	-	-	+	-	-	850	736,930.3	3,168.5	1,305,591.7
AllSol-udWrM	+	-	-	+	-	+	729	887,060.7	20,904.9	9,027,779.2
AllSol-udWRm	+	-	-	+	+	-	1,100	432,623.4	1,951.8	554,152.3
AllSol-udWRM	+	-	-	+	+	+	1,152	363,251.0	2,277.7	556,525.9
AllSol-uDwrm	+	-	+	-	-	-	814	779,238.1	3,769.0	616,690.2
AllSol-uDwrM	+	-	+	-	-	+	708	903,853.3	14,710.3	1,575,871.2
AllSol-uDwRm	+	-	+	-	+	-	1,079	456,240.1	1,673.0	14,873.6
AllSol-uDwRM	+	-	+	-	+	+	1,123	393,522.5	1,712.5	5,238.8
AllSol-uDWrm	+	-	+	+	-	-	846	742,042.1	1,882.5	159,374.9
AllSol-uDWrM	+	-	+	+	-	+	734	872,185.1	9,571.7	902,129.4
AllSol-uDWRm	+	-	+	+	+	-	1,098	433,944.0	1,676.2	14,885.1
AllSol-uDWRM	+	-	+	+	+	+	1,149	359,515.8	1,711.2	5,129.5

Table 3.2: Results summary for all tested configurations (continued)

Configuration	Alg	UF	DI	Weight	RFL	MinD	Instances completed	Average time (ms) by one	Average time (ms) by all	Avg NV by all
AllSol-Udwrm	+	+	-	-	-	-	818	778,630.7	7,338.8	3,457,579.9
AllSol-UdwrM	+	+	-	-	-	+	681	941,690.1	29,693.0	12,564,995.1
AllSol-UdwRm	+	+	-	-	+	-	1,087	448,040.4	1,435.6	553,110.1
AllSol-UdwRM	+	+	-	-	+	+	1,125	389,906.3	1,565.8	556,042.4
AllSol-UdWrm	+	+	-	+	-	-	859	730,162.8	2,821.1	1,316,591.2
AllSol-UdWrM	+	+	-	+	-	+	729	885,366.7	18,196.0	8,610,953.9
AllSol-UdWRm	+	+	-	+	+	-	1,112	423,845.1	1,439.4	553,068.4
AllSol-UdWRM	+	+	-	+	+	+	1,155	353,314.1	1,567.9	555,764.4
AllSol-UDwrm	+	+	+	-	-	-	818	775,624.1	3,273.8	619,570.7
AllSol-UDwrM	+	+	+	-	-	+	710	902,545.4	14,343.4	1,575,725.0
AllSol-UDwRm	+	+	+	-	+	-	1,088	449,940.5	**1,194.6**	14,608.8
AllSol-UDwRM	+	+	+	-	+	+	1,133	381,115.6	1,280.9	5,215.6
AllSol-UDWrm	+	+	+	+	-	-	852	737,131.0	1,760.6	159,166.0
AllSol-UDWrM	+	+	+	+	-	+	734	870,800.0	8,602.7	882,606.3
AllSol-UDWRm	+	+	+	+	+	-	1,103	428,902.8	1,195.9	14,622.5
AllSol-UDWRM	+	+	+	+	+	+	**1,156**	**350,595.4**	1,281.3	**5,113.6**

and the average number of nodes visited by the search of the minimality algorithm, reporting only for instances completed by all configurations. We report the average CPU time in two ways:

1. Over instances completed by at least one configuration (by one). If a configuration does not complete an instance within the allocated 30 minutes, the CPU time is considered to be the time limit (i.e., 1,800,000 ms).

2. Over instances completed by every configuration (by all)

In each of the last four columns (i.e., instances completed, average time, and average nodes visited), we typeset the best entry for each of PerTuple and AllSol in boldface. Finally, we put a border around the configuration that completes the largest number of instances, for each of PerTuple and AllSol.

3.7.2.2 Comparing the Original and Best Configurations

Next, we compare the basic configuration of an algorithm (i.e., factors udwrm) to its configuration that completes the largest number of instances:

1. PerTuple-udwrm versus PerTuple-UdWrm

2. AllSol-udwrm versus AllSol-UDWRM

Figure 3.1 is a cactus plot of all four configurations, displaying the cumulative number of instances completed over time. We see that the improvement of AllSol is quite dramatic, altering the slope of its curve to more closely match that of PerTuple. The improvement of PerTuple is also significant, although to a lesser extent.

Figure 3.2 and 3.3 are scatter plots comparing the run time of individual instances of the two chosen configurations of each algorithm, providing a per-instance performance comparison. Each point represents a single instance. The axes indicate the

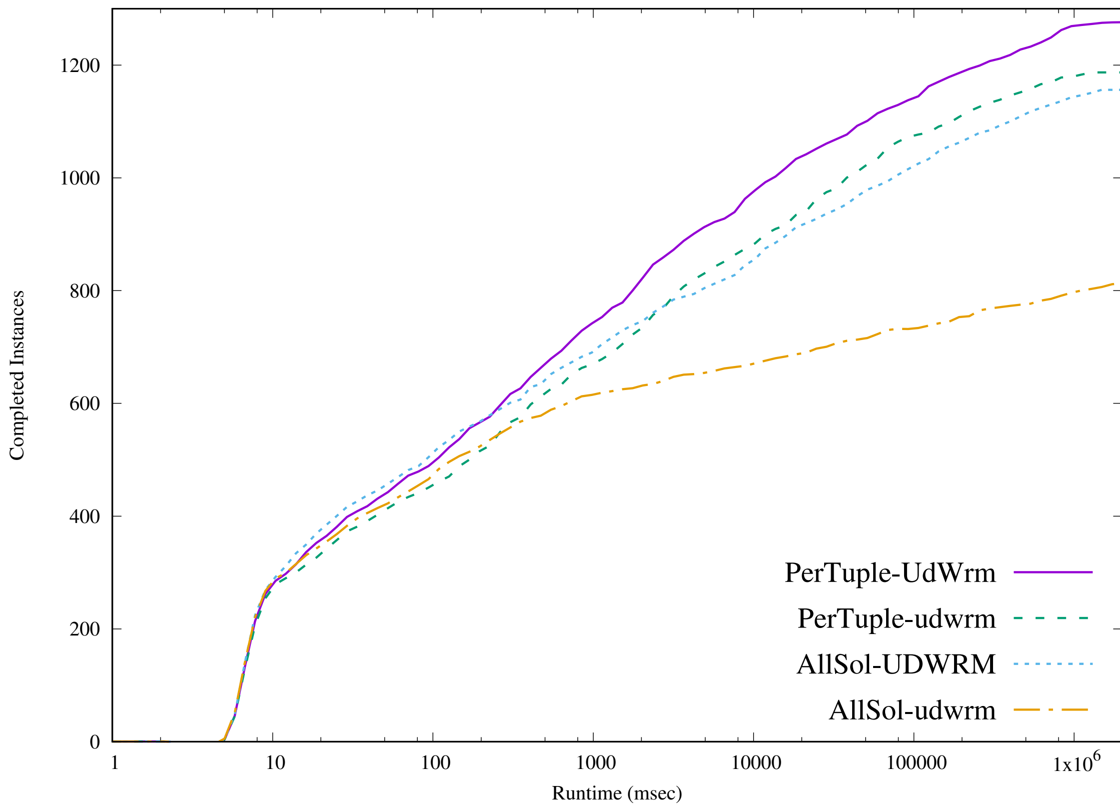


Figure 3.1: Instance completions over time

runtime on a logarithmic scale. The vertical axis reports the runtime of the original configuration of the algorithm and the horizontal axis shows the improved configuration. Consequently, a point above the diagonal indicates that the new configuration outperforms the original one. In both figures, the vast majority of the points lie above the diagonal, clearly indicating that improvement, often of several orders of magnitude. Only a handful of points are below the diagonal. Furthermore, the performance never deteriorates by more than one order of magnitude. In particular, for AllSol (Figure 3.3), a large number of points are pushed against the upper horizontal boundary, indicating that the improved configuration completes many instances that the original configuration did not complete.

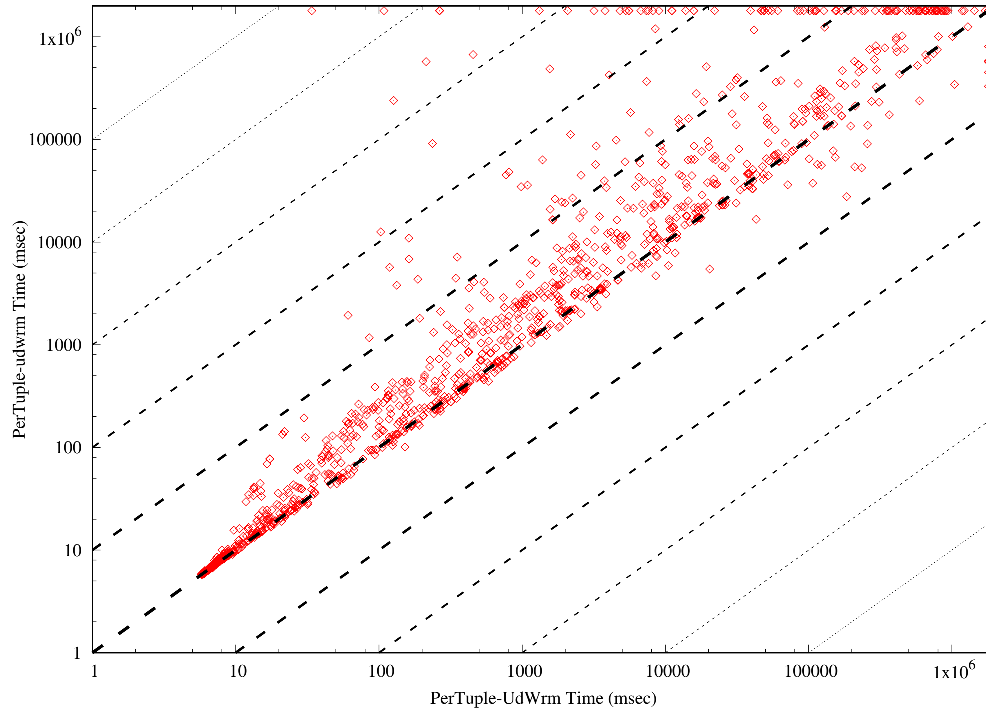


Figure 3.2: PerTuple: runtime comparison per instance

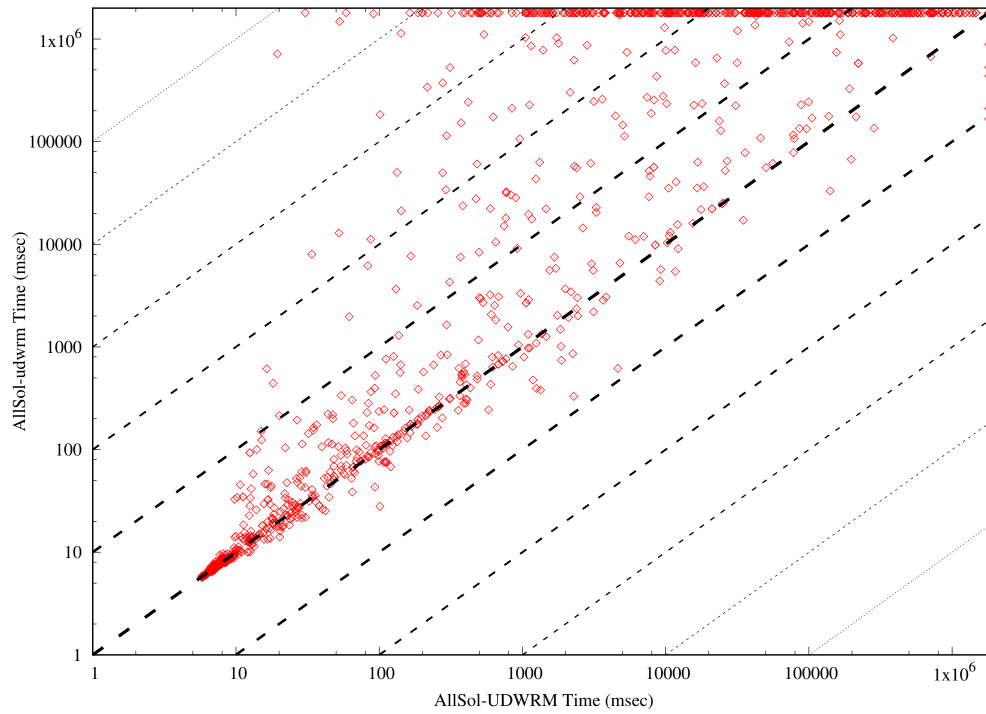


Figure 3.3: AllSol: runtime comparison per instance

3.7.2.3 ANOVA Results

Our factorial-experiment design allows us to perform a detailed analysis of the effects and interactions of all six considered factors. We run a six-way, repeated measures ANOVA test using a within-subject design. We consider runtime as the dependent variable. We apply a log transformation to the runtime data in order to achieve an almost normal distribution. Our data is right-censored because of the 30-minute time limit. For this reason, our data can be considered only 'approximately normally distributed.'

Table 3.3 reports the results of the ANOVA test. In the table, we provide the following data: The effect name (effect), the degrees of freedom of the effect (DFn), the degrees of freedom of the error (DFd), the sum of squares of the effect (SSn), the sum of squares of the error (SSd), the F measure (F), the corresponding p-value (p), an indicator of the significance of the effect (p<.05), and the generalized eta-squared measure of effect size (ges).

Each row in this table shows the effect of a given combination of factors. If the effect is a single factor, the data in the row shows the significance of the considered factor. If the effect is a combination of factors, the data indicates whether their interaction is significant. The asterisk reported in the column labeled "p<.05" indicates that the corresponding effect is statistically significant. The table is sorted by effect size (i.e., decreasing values of the "ges" column).

The ANOVA test show the following results:

1. The factors tested in this experiment are all closely related and interwoven with each other. As a result, it is not surprising to see that all the factors and most of the interactions have statistically significant effects.

2. Each of the six factors tested appear near the top of the table sorted by effect

Table 3.3: ANOVA results sorted by generalized eta squared

Effect	**DFn**	**DFd**	**SSn**	**SSd**	**F**	**p**	**p<.05**	**ges**
(Intercept)	1	1389	1.13E+06	3.67E+05	4.27E+03	0.00E+00	*	7.39E-01
RFL:MinD	1	1389	3.16E+03	4.28E+03	1.03E+03	5.36E-169	*	7.86E-03
RFL	1	1389	2.88E+03	8.22E+03	4.87E+02	1.02E-92	*	7.16E-03
Alg	1	1389	1.11E+03	5.07E+03	3.04E+02	9.19E-62	*	2.77E-03
Alg:RFL	1	1389	9.48E+02	3.09E+03	4.26E+02	9.26E-83	*	2.37E-03
Alg:RFL:MinD	1	1389	7.90E+02	2.19E+03	5.00E+02	7.95E-95	*	1.97E-03
MinD	1	1389	7.39E+02	2.36E+03	4.35E+02	3.38E-84	*	1.85E-03
Alg:MinD	1	1389	3.32E+02	1.58E+03	2.91E+02	2.31E-59	*	8.30E-04
DI	1	1389	1.96E+02	1.09E+03	2.49E+02	1.04E-51	*	4.90E-04
UF	1	1389	9.67E+01	1.40E+02	9.60E+02	1.19E-160	*	2.42E-04
Weight	1	1389	7.92E+01	4.86E+02	2.26E+02	1.71E-47	*	1.98E-04
Alg:UF	1	1389	5.29E+01	9.68E+01	7.60E+02	8.79E-134	*	1.33E-04
MinD:DI	1	1389	4.07E+01	2.82E+02	2.00E+02	1.63E-42	*	1.02E-04
Weight:RFL	1	1389	3.79E+01	3.60E+02	1.47E+02	3.78E-32	*	9.50E-05
Alg:DI	1	1389	3.21E+01	4.42E+02	1.01E+02	5.53E-23	*	8.04E-05
Alg:Weight:RFL	1	1389	1.55E+01	1.67E+02	1.29E+02	1.02E-28	*	3.89E-05

Table 3.3: ANOVA results sorted by generalized eta squared (continued)

Effect	**DFn**	**DFd**	**SSn**	**SSd**	**F**	**p**	**p<.05**	**ges**
Alg:RFL:DI	1	1389	1.19E+01	1.27E+02	1.30E+02	9.13E-29	*	2.98E-05
UF:RFL	1	1389	6.54E+00	3.23E+01	2.81E+02	1.34E-57	*	1.64E-05
Weight:MinD	1	1389	5.62E+00	2.63E+02	2.97E+01	6.06E-08	*	1.41E-05
UF:RFL:MinD	1	1389	4.60E+00	2.21E+01	2.89E+02	4.90E-59	*	1.15E-05
Alg:RFL:MinD:DI	1	1389	4.01E+00	9.53E+01	5.84E+01	3.92E-14	*	1.00E-05
Alg:UF:MinD	1	1389	3.93E+00	2.15E+01	2.53E+02	1.74E-52	*	9.83E-06
Alg:Weight	1	1389	3.82E+00	1.29E+02	4.12E+01	1.87E-10	*	9.57E-06
UF:MinD	1	1389	3.72E+00	2.31E+01	2.24E+02	4.58E-47	*	9.33E-06
Alg:Weight:RFL:MinD	1	1389	3.30E+00	1.85E+02	2.48E+01	7.12E-07	*	8.27E-06
RFL:DI	1	1389	3.04E+00	1.75E+02	2.41E+01	1.00E-06	*	7.62E-06
Alg:UF:RFL:MinD	1	1389	2.55E+00	1.68E+01	2.11E+02	1.48E-44	*	6.38E-06
UF:DI	1	1389	1.81E+00	1.41E+01	1.78E+02	2.43E-38	*	4.53E-06
Alg:UF:RFL	1	1389	1.68E+00	2.18E+01	1.07E+02	2.90E-24	*	4.21E-06
Alg:MinD:DI	1	1389	1.46E+00	1.28E+02	1.59E+01	6.94E-05	*	3.66E-06
Alg:Weight:MinD	1	1389	1.39E+00	1.83E+02	1.06E+01	1.17E-03	*	3.49E-06
Weight:RFL:MinD	1	1389	1.04E+00	3.11E+02	4.66E+00	3.11E-02	*	2.61E-06

Table 3.3: ANOVA results sorted by generalized eta squared (continued)

Effect	**DFn**	**DFd**	**SSn**	**SSd**	**F**	**p**	**p<.05**	**ges**
Alg:UF:DI	1	1389	7.98E-01	1.07E+01	1.04E+02	1.36E-23	*	2.00E-06
Weight:DI	1	1389	6.34E-01	2.43E+01	3.63E+01	2.16E-09	*	1.59E-06
Weight:RFL:DI	1	1389	4.29E-01	2.19E+01	2.72E+01	2.15E-07	*	1.07E-06
UF:RFL:DI	1	1389	3.50E-01	4.21E+00	1.16E+02	6.04E-26	*	8.77E-07
Alg:UF:RFL:DI	1	1389	2.96E-01	3.86E+00	1.06E+02	4.38E-24	*	7.40E-07
UF:MinD:DI	1	1389	2.89E-01	4.39E+00	9.15E+01	4.91E-21	*	7.25E-07
Alg:UF:MinD:DI	1	1389	1.91E-01	3.74E+00	7.09E+01	9.07E-17	*	4.78E-07
Weight:MinD:DI	1	1389	1.71E-01	1.79E+01	1.33E+01	2.70E-04	*	4.29E-07
Weight:UF:MinD	1	1389	1.70E-01	5.28E+00	4.47E+01	3.37E-11	*	4.26E-07
Weight:UF:RFL:MinD	1	1389	1.68E-01	5.36E+00	4.35E+01	6.10E-11	*	4.20E-07
Alg:Weight:UF:MinD	1	1389	1.67E-01	6.09E+00	3.81E+01	8.85E-10	*	4.19E-07
Alg:Weight:UF:RFL:MinD	1	1389	1.63E-01	5.97E+00	3.80E+01	9.06E-10	*	4.09E-07
UF:RFL:MinD:DI	1	1389	1.56E-01	3.54E+00	6.13E+01	9.73E-15	*	3.91E-07
Weight:RFL:MinD:DI	1	1389	1.37E-01	1.86E+01	1.02E+01	1.41E-03	*	3.44E-07
Alg:UF:RFL:MinD:DI	1	1389	1.06E-01	3.46E+00	4.26E+01	9.22E-11	*	2.66E-07
Weight:UF:RFL	1	1389	2.90E-02	5.28E+00	7.62E+00	5.84E-03	*	7.25E-08

Table 3.3: ANOVA results sorted by generalized eta squared (continued)

Effect	DFn	DFd	SSn	SSd	F	p	p<.05	ges
Alg:Weight:UF:RFL	1	1389	1.85E-02	5.74E+00	4.47E+00	3.46E-02	*	4.63E-08
Alg:Weight:RFL:DI	1	1389	1.66E-02	3.30E+01	7.01E-01	4.03E-01		4.17E-08
Weight:UF	1	1389	1.36E-02	5.66E+00	3.35E+00	6.73E-02		3.42E-08
Alg:Weight:DI	1	1389	6.77E-03	3.29E+01	2.85E-01	5.93E-01		1.69E-08
Weight:UF:DI	1	1389	6.03E-03	2.62E+00	3.19E+00	7.41E-02		1.51E-08
Alg:Weight:UF:RFL:DI	1	1389	4.81E-03	2.54E+00	2.63E+00	1.05E-01		1.20E-08
Alg:Weight:MinD:DI	1	1389	4.13E-03	1.49E+01	3.84E-01	5.35E-01		1.03E-08
Alg:Weight:UF	1	1389	3.68E-03	5.42E+00	9.44E-01	3.31E-01		9.23E-09
Weight:UF:RFL:MinD:DI	1	1389	3.27E-03	2.32E+00	1.96E+00	1.62E-01		8.18E-09
Alg:Weight:RFL:MinD:DI	1	1389	1.22E-03	1.43E+01	1.19E-01	7.31E-01		3.05E-09
RFL:MinD:DI	1	1389	8.24E-04	1.56E+02	7.35E-03	9.32E-01		2.06E-09
Weight:UF:MinD:DI	1	1389	8.02E-04	2.31E+00	4.83E-01	4.87E-01		2.01E-09
Alg:Weight:UF:RFL:MinD:DI	1	1389	7.95E-04	2.35E+00	4.70E-01	4.93E-01		1.99E-09
Alg:Weight:UF:DI	1	1389	5.30E-04	2.60E+00	2.83E-01	5.95E-01		1.33E-09
Alg:Weight:UF:MinD:DI	1	1389	2.72E-04	2.33E+00	1.62E-01	6.88E-01		6.81E-10
Weight:UF:RFL:DI	1	1389	1.49E-04	2.59E+00	8.00E-02	7.77E-01		3.73E-10

size.

3. The most impactful single factor is the use of Real Full-Lookahead (RFL), followed closely by the choice of algorithm (i.e., AllSol or PerTuple).

4. The next most important is the use of the minimal dual graph, followed by dangle identification, UF, and finally the use of a weighted ordering.

5. We see that both the 2-way and 3-way interactions between Alg, RFL, and MinD are highly ranked.

6. There are also highly ranked interactions between the choice of algorithm and UF as well as between the minimal dual graph and dangle identification.

7. Importantly, not a single factor is shown to be insignificant.

3.7.3 Discussion

By examining the results shown in Table 3.2, we conclude the following about the five improvements we introduced in this chapter:

UF value ordering. The unmarked-first value ordering heuristic is beneficial. It has a minor positive effect for AllSol and yields noticeable improvement for PerTuple. Without this heuristic, PerTuple has a tendency to reuse tuples across solutions, resulting in overlapping solutions and fewer marked tuples. UF 'encourages' PerTuple to find solutions with unmarked tuples, allowing more tuples to be marked more rapidly and fewer searches to be made. UF exhibits a reliable and stable positive performance and we decide to always use it.

Dangle identification. The effect of dangle identification is less straightforward. On its own, it can be useful, but when paired with a minimal dual graph, its

effectiveness is *greatly* increased. Indeed, the minimal dual graph can remove a large number of redundant edges, resulting in a much sparser graph where dangling tree structures are increasingly prevalent. When paired with real-full lookahead, maintaining arc consistency on the dangles yields insignificant overhead. Thus, AllSol, which already performs better with RFL and the minimal dual, can also benefit from the use of dangle identification. However, PerTuple performs better when using forward checking and, consequently, does not benefit from the minimal dual graph. As a results, the effectiveness of dangle identification for PerTuple is limited. However, in the next chapter, we propose a technique to make dangle identification advantageous for PerTuple as well as for AllSol, which allows to always use dangle identification for both algorithms.

dom/wdeg variable ordering. Using dom/wdeg weighted ordering heuristic during search generally provides a small but consistent improvement. The cases where it is detrimental are limited. Thus, we decide to use the weighted ordering in all cases.

Lookahead: FC versus RFL. The impact of real-full lookahead deserves a discussion. When used with PerTuple, RFL is oftentimes a costly and wasteful operation. At each node in the search tree, RFL expends effort maintaining arc consistency throughout all future nodes. As soon as a solution is found, all filtering effort is thrown away and a new search begins. Additionally, the constraint checks performed for RFL are more costly than those performed for FC (i.e., DualFC vs. DualRFL). However, AllSol significantly benefits from RFL because none of the lookahead effort is wasted. Upon finding a solution, AllSol simply backtracks, preserving the filtering that it performed by at pre-

vious levels. As a result, in the rest of this thesis, we use FC for PerTuple and RFL for AllSol.

Minimal dual graph. The benefit of the minimal dual graph is closely tied to which lookahead strategy it is combined with. When used with forward checking, using the minimal dual graph can, generally speaking, be costly. Although we never lose correctness, using a minimal dual graph may delay the detection of inconsistencies by removing edges. In the case of real-full lookahead, we do not encounter this problem. Because the entire network is made arc consistent at every step, the removal of redundant edges does not delay the detection of inconsistency. In fact, it can result in a noticeable speedup due to the reduced number of constraint checks that RFL needs to perform to reach a fixpoint. For this reason, we always use a minimal dual graph and RFL for AllSol. As we discuss in the next chapter, for PerTuple, we use a minimal dual graph only to identify dangles but we execute forward checking on the original dual graph.

Summary

In the chapter, we discussed several techniques to improve the performance of the two algorithms for enforcing relational minimality, namely PerTuple and AllSol. Two techniques, dangle identification and the unmarked-first ordering are novel and shown to be useful in this context. Three other techniques, namely, the dom/wdeg heuristic, RFL, and minimal dual graph, have previously appeared in the literature but had never been applied before in the context of constraint minimality. We empirically validate the benefits of the five proposed improvements on the performance of PerTuple and AllSol, drawing conclusions on how they should be used.

Chapter 4

Which Minimal Dual Graph

In Chapter 3, we investigated the impact of using a minimal dual graph on the performance of PerTuple and AllSol. We also argued that alternative minimal dual graphs can be generated using the efficient algorithm proposed by Janssen *et al.* [1989] and use the MinDeg heuristic to this end (see Section 3.5). In this chapter, we re-examine our initial decision and propose instead a new heuristic, MaxDeg, which combines particularly favorably with dangle identification.

4.1 A Minimal Dual Graph in PerTuple

In Chapter 3, we concluded that dangle identification is a useful strategy and using the minimal dual graph only increases its effectiveness. However, we also determined that the better lookahead strategy for PerTuple is forward checking but its benefits are greatly hindered when using a minimal dual graph. Consequently, we recommended executing FC on the original dual graph for PerTuple, sacrificing benefits to be drawn from dangle identification.

Generally speaking, the choice of using a dual graph has three effects on the search

of PerTuple and AllSol:

1. *Lookahead*: The graph determines which edges are checked during the lookahead procedure.

2. *Variable ordering*: The graph affects the computation of the variable ordering heuristics dom/deg and dom/wdeg.

3. *Dangle identification*: The graph is used to identify dangling vertices to remove from search.

In Chapter 3, we used the same graph for all three purposes. In order to remedy the above-discussed limitation related to using a minimal dual graph for PerTuple, we propose, for the case of PerTuple, to use a minimal dual graph *only* for the purpose of dangle identification but use the original dual graph for both lookahead and variable ordering. This choice allows us to promote the benefits of dangle identification in PerTuple while avoiding hindering the effectiveness of lookahead.

Theorem 5. *The search procedure maintains correctness when using a minimal dual graph for dangle identification and the original dual graph for lookahead and ordering.*

Proof. Search is correct when using a minimal dual for all three purposes. A change in the computation of the ordering may change the ordering, but cannot break correctness. Including additional, redundant, edges results in new constraint checks but cannot affect the set of solutions. By definition, the equalities enforced along the redundant edges are also enforced along an alternative existing path of edges. □

4.2 The MaxDeg Heuristic for a Minimal Dual Graph

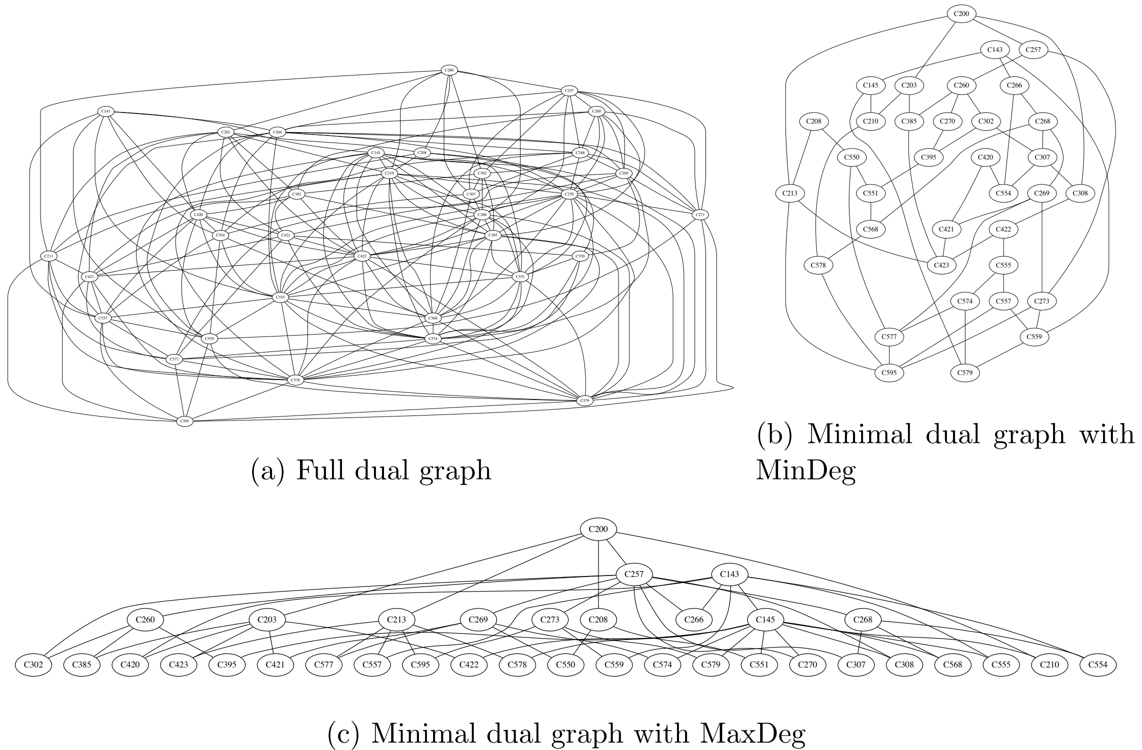
Many different minimal dual graphs may exist for any given dual graph. The heuristic used during the construction of a minimal dual in the will affect the structure of the resulting graph.

Our procedure for constructing the minimal dual graph is based on the algorithm by Janssen *et al.* [1989]. The algorithm first identifies all subscopes s_i in the original dual graph where a subscope is the set of CSP variables shared by two dual variables. For each subscope s_i, it builds the set A_{s_i} of dual variables C_x where $s_i \subseteq scope(C_x)$. As a result, the dual variables in every set A_{s_i} induce a clique on the original dual graph. Next, we build a partial order on the subscopes comparing two subscopes using the relation $\subseteq$ and induce a total ordering by topological sorting. Starting from the set A_{s_i} with the largest subscope in the total ordering, we incrementally connect the relations within A_{s_i} to form a connected component, the seed of a minimal dual graph, by adding a single edge between two disconnected components in A_{s_i}. Then, we move to the A_{s_j} such that s_j directly precede s_i in the total ordering and repeat the same operation. If the original dual graph is connected, we are guaranteed to end the process with a connected minimal dual graph. In order to connect two connected components, we add an edge between two of their dual variables. At this point, we have the freedom to choose the two variables between which the edge is added. The heuristic MaxDeg adds an edge between the two dual variables of largest degree whereas MinDeg does it for the dual variables of smallest degree.

In contrast to MinDeg, in MaxDeg, the edges in a minimal dual graph tend to be consolidated around a few vertices of high degree while the majority of vertices are of low degree. Such a structure for a minimal dual graph would yield many dangles

upon the assignment of the high degree vertices.

Figure 4.1 illustrates the difference that the choice of heuristic can make. Figure 4.1a shows a dual graph, Figure 4.1b a minimal dual graph using the MinDeg heuristic, and Figure 4.1c a minimal dual graph using the MaxDeg heuristic.[1]



(a) Full dual graph

(b) Minimal dual graph with MinDeg

(c) Minimal dual graph with MaxDeg

Figure 4.1: Dual graphs of an example instance

Table 4.1 presents statistics on the edges and vertex degrees of each graph. Clearly a minimal dual graph can remove a significant number of redundant edges (190 versus 57 edges). The difference between the two heuristics can be seen in the degree statistics as well as visually. The graph in Figure 4.1b has a maximum degree of only 5 and a median of 3. In contrast, the graph in Figure 4.1c has a maximum degree of 12 with a median of 2 because its edges are 'concentrated' over a few vertices, resulting

[1]Example is a cluster taken from the composed-75-1-2-0 instance

in a more tree-like structure.

Table 4.1: Degree statistics of the dual graphs in Figure 4.1

Graph	Edges	Degree			
		Min	Max	Mean	Median
Full Dual Graph	190	6	17	10.86	10
Minimal Dual Graph - MinDeg	57	2	5	3.26	3
Minimal Dual Graph - MaxDeg	57	2	12	3.26	2

4.3 Metrics for Dangle Identification

We propose two metrics to assess the effectiveness of the dangle-identification procedure: Normalized Average Dangle-Level (NADL) and Average Percent Dangles-Identified (APDI).

4.3.1 Normalized Average Dangle Level

The NADL metric measures at what depth level during search dangles are identified. We use a list to record the depth at which a dangle is found. At the beginning the list is empty. If we identify any dangling vertices at depth d, we add to the list as many 'd' entries as there are dangling vertices (zero for preprocessing). Upon search completion, we compute the average of all recorded entries. Then, we normalize the resulting number by the number of vertices in the dual graph. Thus, this metric ranges from 0 (entire problem identified as dangles at preprocessing) to $\frac{e-2}{e}$, where e is the number of constraints in the CSP. Note that last two vertices in the dual graph are always necessarily dangles.

4.3.2 Average Percent Dangles Identified

The APDI metric measures the percentage of future dual variables that are identified as dangles at each step (including preprocessing and instantiations of dual variables). At each step, the percentage of dangling vertices to total considered vertices at this step is recorded in a list. Upon search completion, we compute the average of the percentages in the list. The metric ranges from 0 (no dangles are ever identified) to 1 (entire problem is identified as dangles during preprocessing).

4.4 Experimental Evaluation

Below, we describe an experiment to empirically assess the effectiveness of the MaxDeg heuristic against that of MinDeg.

4.4.1 Setup

We test the effect of the MaxDeg heuristic on both the runtime and dangle identification performance of AllSol and PerTuple. We test a total of six configurations, three for AllSol and three for PerTuple. For each algorithm, we compare using the full dual graph, using the minimal dual generated from the MinDeg heuristic, and the minimal dual generated from the MaxDeg heuristic. In the case of PerTuple, we use a minimal dual only for the purpose of dangle identification. For variable ordering and lookahead, we use the original dual graph.

As stated in Section 3.7.1, our goal is to execute AllSol and PerTuple on clusters of a tree decomposition of a CSP. Thus, we run our experiments on individual clusters taken from decompositions of full instances. We use instances from benchmarks in the XCSP library.[2] We sample 10 clusters from each of 175 benchmarks in

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

the library. This sampling yields 1,684 total clusters (some benchmarks have fewer than 10 clusters).[3] We run each of the six configurations on all 1,684 clusters for a total of 10,104 runs.

We run our experiments on a computer cluster of Intel Xeon E5-2670 2.60 GHz processors. We allocate 30 minutes and 12GB of memory per run. To account for load variations on the cluster computer, we measure instruction count and convert it to runtime using a standardized measure of instructions per cycle and clock speed. For each run, we first enforce GAC prior to enforcing minimality.

4.4.2 Results

Table 4.2 summarizes the performance of the six tested configurations, indicating, for each configuration, the number of instances completed, the average runtime on instances completed by at least one configuration, the average runtime on instances completed by all configurations, the average number of nodes visited (in the minimality search) on instances completed by all configurations, the average NADL, and the average APDI. We highlight the row with best values for each algorithm. For both PerTuple and AllSol, we see that using the minimal dual graph generated from the MaxDeg heuristic yields the best performance in every measured aspect.

Figures 4.2, 4.3, 4.4, 4.5, 4.6, and 4.7 plot the runtimes of the tested configurations on individual clusters. Each scatter plot compares the runtimes of the two configurations on its axes. In each plot, we place the 'original' configuration on the vertical axis and the configuration with an improvement on the horizontal axis. Consequently, points above the diagonal indicate a positive improvement.

[3] The same clusters tested in Chapter 3

Table 4.2: Results summary for tested dual graph types

Configuration	Instances completed	Average time (ms) by one	Average time (ms) by all	Avg NV by all	Avg NADL by all	Avg APDI by all
PerTuple-full	1,268	219,550.2	7,130.7	415,505.6	0.48	35%
PerTuple-MinDeg	1,250	241,635.1	8,788.3	376,230.5	0.21	47%
PerTuple-MaxDeg	1,371	106,588.2	3,783.7	40,304.8	0.12	53%
AllSol-full	1,103	444,505.6	54,620.9	506,474.0	0.50	40%
AllSol-MinDeg	1,156	367,089.4	20,850.9	242,457.8	0.21	52%
AllSol-MaxDeg	1,184	338,317.4	20,506.7	230,128.5	0.12	62%

MinDeg does not significantly affect the performance of PerTuple (Figure 4.2) but improves that of AllSol (Figure 4.3)

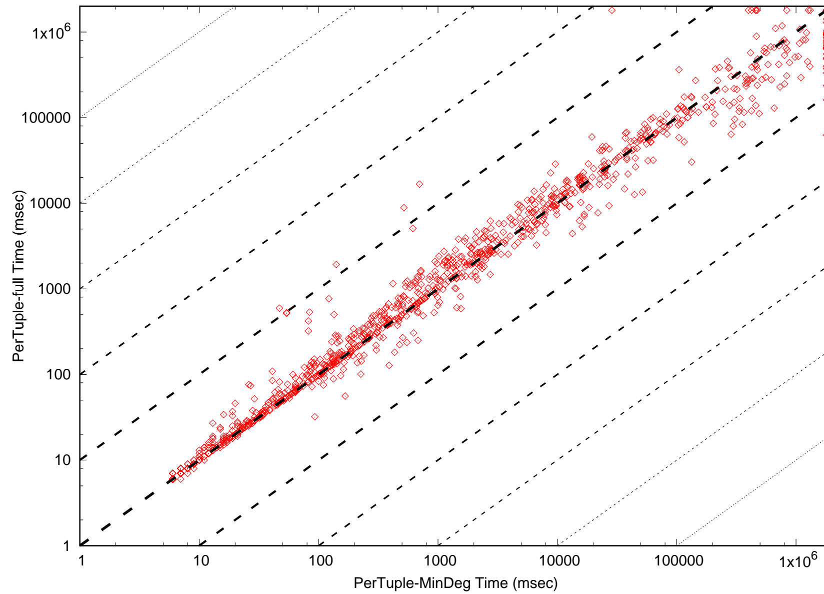


Figure 4.2: PerTuple's runtime on original vs. minimal dual graph with MinDeg

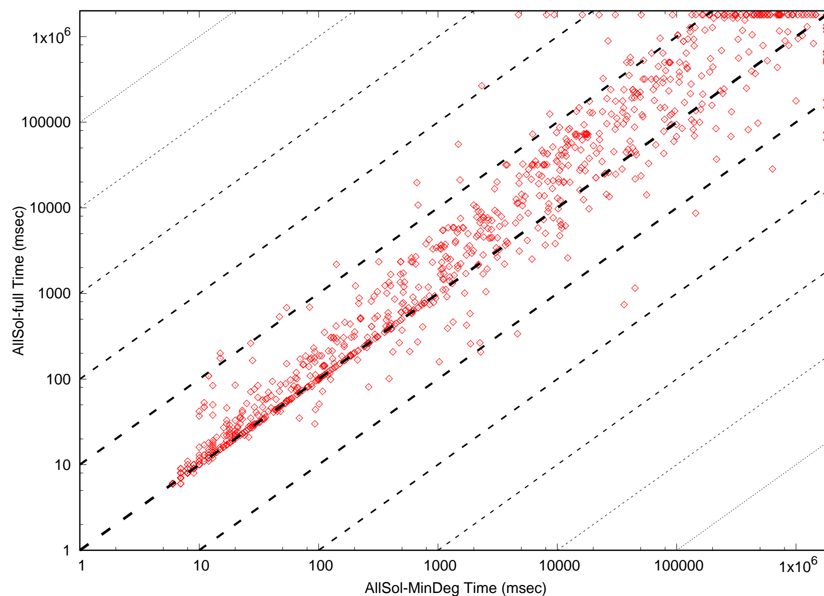


Figure 4.3: AllSol's runtime on original vs. minimal dual graph with MinDeg

MaxDeg significantly outperforms MinDeg for PerTuple (Figure 4.4). As for AllSol, the improvement is small but generally positive (Figure 4.5).

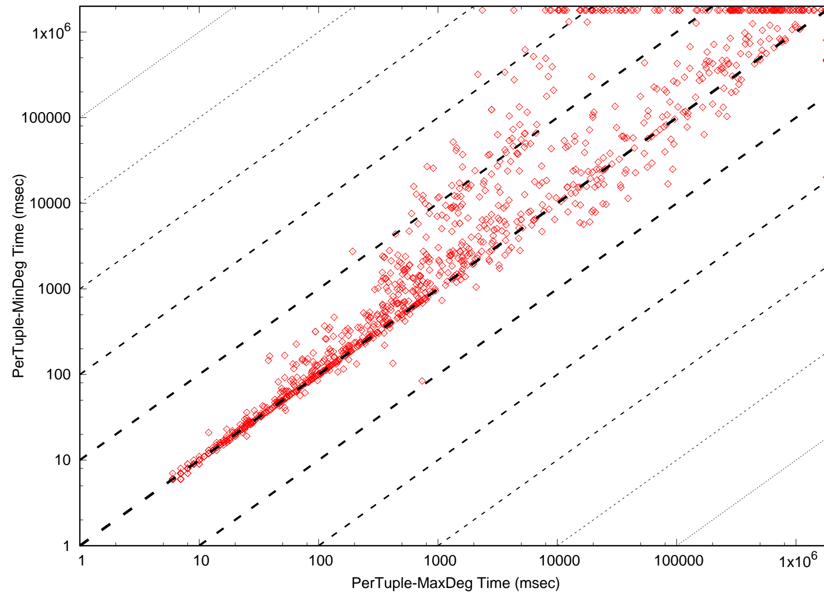


Figure 4.4: PerTuple's runtime on minimal dual graph with MinDeg versus MaxDeg

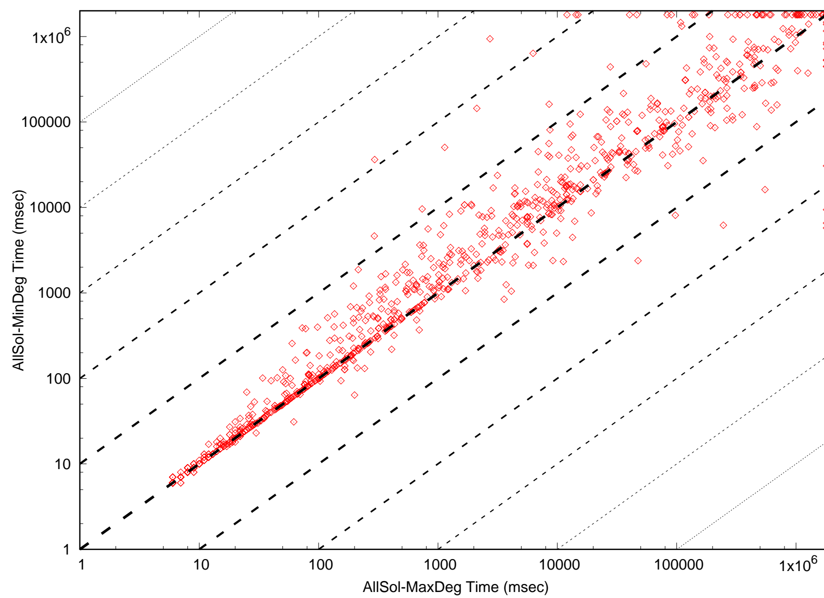


Figure 4.5: AllSol's runtime on minimal dual graph with MinDeg versus MaxDeg

MaxDeg significantly improves the performance of PerTuple (Figure 4.6) AllSol (Figure 4.7), with many instances showing an order of magnitude improvement.

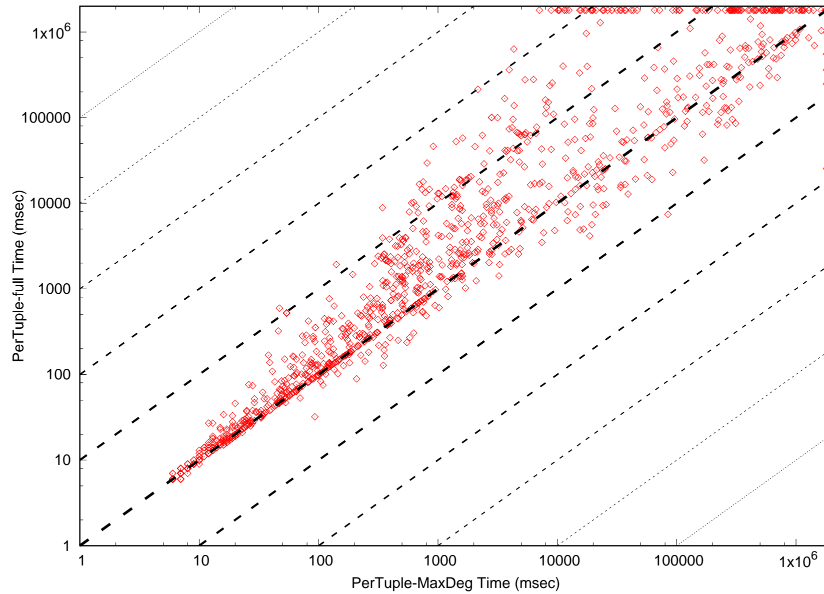


Figure 4.6: PerTuple's runtime on original vs. minimal dual graph with MaxDeg

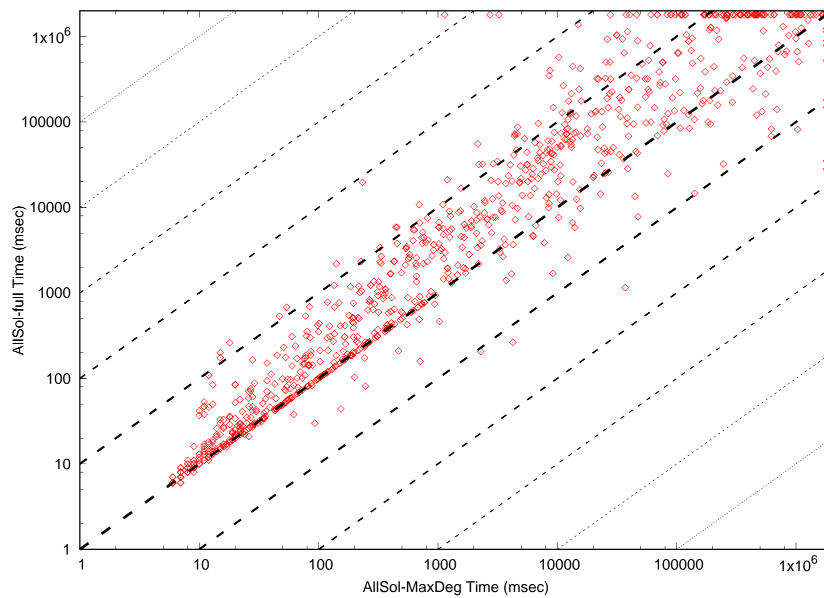


Figure 4.7: AllSol's runtime on original vs. minimal dual graph with MaxDeg

4.4.3 Discussion

Our results show that using a minimal dual graph significantly improves the effectiveness of dangle identification by allowing more dangles to be earlier in the search. Using a minimal dual graph removes edges, allowing the dual graph to become more quickly tree-like. We also see that the specific minimal dual graph used has a significant impact on dangle identification. While the minimal dual graphs computed with MinDeg and MaxDeg have the same number of edges, the minimal dual graph generated with MaxDeg is more beneficial for dangle identification. Having a few high degree vertices and many low degree vertices results in a graph with high potential for dangling tree structures. In particular, a variable ordering heuristic that favors selecting vertices of high degree (e.g., dom/deg and dom/wdeg) quickly removes a large numbers of edges.

Using the MaxDeg heuristic improves the performance of both AllSol and PerTuple. In the case of PerTuple, a minimal dual is only used for the purpose of dangle identification. In Chapter 3, we had argued that dangle identification was slightly detrimental to PerTuple performance. With the use of the MaxDeg heuristic, we are able to invert the effect. Indeed, dangle identification now benefits the performance of PerTuple.

Summary

In this chapter, we introduced MaxDeg as a heuristic for the algorithm that computes a minimal dual graph. We showed that MaxDeg results in minimal dual graphs that are well suited to dangle identification. We empirically evaluate the effect of the heuristic on the runtime of PerTuple and AllSol and find substantial improvement in the case of PerTuple and moderate improvement in the case of AllSol.

Chapter 5

Weight Update in High-Level Consistencies

This chapter discusses a potential obstacle for high-level consistencies: the lack of a coherent weight-update strategy for the dom/wdeg variable-ordering heuristic. We introduce a parameterized framework for updating constraint weights when using HLC and empirically evaluate its effectiveness.

5.1 Weight-Update Strategies: Motivation

Using high-level consistencies allows for stronger reasoning power to be used on problems that require it. However, it remains an open question whether and how HLCs prunings should affect variable ordering during search. Currently, the most popular variable ordering heuristic is dom/wdeg [Boussemart *et al.*, 2004]. It is used in the context of both binary and non-binary constraints and operates based on the principle of "the squeaky wheel gets the grease" as follows. The weights of all constraints are initially set to one. Every time enforcing a constraint yields a domain wipeout,

the weight of the constraint is incremented by one. Dom/wdeg instantiates first the variable whose ratio of current domain size to weighted degree is the smallest, where the weighted degree of a variable is the sum of the weights of the constraints that apply to it. In the case of GAC, it is straightforward to assign blame for a wipeout on the constraint currently being checked. In the case of high-level consistencies, the the wipeout is typically the result of the confluence of a number of constraints. Woodward and Choueiry [2017] explored weight-update strategies for dom/wdeg in the context of Partition-One Arc-Consistency (POAC) [Bennaceur and Affane, 2001] and Relational Neighborhood Inverse Consistency (RNIC) [Woodward *et al.*, 2011]. In this chapter, we explore strategies suitable for cluster minimality.

Cluster minimality is similar to RNIC in that both enforce minimality on a subproblem. The former does it for every constraint in the subproblem, the latter for one constraint. They also differ in how a subproblem is defined. RNIC operates on the subproblem induced by the neighborhood, in the dual graph, of a single constraint while cluster minimality is executed on clusters of a tree decomposition. For RNIC, it is reasonable to assign blame, in the event of a wipeout, on the constraint on which RNIC is enforced. However, in the case of cluster minimality, it is not clear how to assign blame. If a wipeout occurs, it is because the subproblem in the cluster is unsatisfiable and it would be so regardless of the search order over the constraints in the cluster. Below, we investigate directions for meaningful weight-updates in the context of cluster minimality.

5.2 Weight-Update Parameters

To define new weight-update strategies, we propose a framework based on three orthogonal parameters:

1. *Occurrence*: when is a weight update performed.

2. *Distribution*: how to allocate the weight amongst the constraints.

3. *Scale*: what is the magnitude of the total weight update.

In the following sections, we describe each parameter and propose two or more possible options for the parameter.

5.2.1 Occurrence

The occurrence parameter specifies when weights must be updated. When using dom/wdeg with GAC, weights are only updated in the advent of a domain wipeout as a result of a constraint check. Enforcing a high-level consistency is typically a complex and relatively time-consuming operation compared to enforcing GAC: It may encompass hundreds or thousands of constraint checks. Conceivably, a weight update may be beneficial even in the absence a domain wipeout. We consider two options for the occurrence parameter:

Always: The 'Always' setting means that we always update weights by some amount: Every time a cluster has been processed, the weights of the cluster's constraints are incremented by some value.

OnWipeout: The 'OnWipeout' setting is more conservative. It updates the constraints' weights following the detection of an inconsistency (wipeout) within the cluster. This method is similar to how weights are typically updated with GAC (i.e., only when a constraint check results in wipeout).

5.2.2 Distribution

The distribution parameter specifies how the weight update is allocated across $\psi(cl)$, the constraints of a given cluster cl. We denote $u_b(c, cl)$ the base weight-update function of a single constraint $c \in \psi(cl)$ and require that:

$$\sum_{c_i \in \psi(cl)} u_b(c_i, cl) = 1. \tag{5.1}$$

We consider two options. On one hand, the simplest strategy, which uniformly updates the weights of all constraints, may or may not provide enough discriminative power. On the other hand, we propose to distribute the unity weight-update of Expression (5.1) based on the performance of search on the dual CSP. Below, we describe these two options:

Uniform: The 'Uniform' setting is straightforward in that it uniformly updates all constraints in a cluster. The base weight update for a constraint is given by:

$$u_b(c, cl) = \frac{1}{|\psi(cl)|}.$$

DualWeight: The 'DualWeight' setting allocates the weight update in accordance with the weights of the dual edges (i.e., equality constraints). When executing search over the subproblem (i.e., AllSol or PerTuple), the weights of the equality constraints of the dual graph are computed similarly to dom/wdeg on a general CSP. At the conclusion of the search, we compute the *increase* of the weighted degree of each dual variable c and denote it $\Delta wdeg(c)$. We assign the

base weight-update proportionally to each constraint's $\Delta wdeg(\cdot)$:

$$u_b(c, cl) = \frac{\Delta wdeg(c)}{\sum\limits_{c_i \in \psi(cl)} \Delta wdeg(c_i)}.$$

5.2.3 Scale

The scale parameter specifies by how much to update the weights. Typically, the weight of a constraint is incremented by one. When enforcing high-level consistency, we may want to increment the weights of many constraints and by a larger amount to reflect the efforts it took to process a particular subproblem. To this end, we need to consider adjusting the total amount of weight updates.

For both values of the distribution parameter, the total base weight-update is one, see Expression (5.1). We denote by $s(cl)$ the scaling factor of a cluster cl. The weight update of a constraint $c \in \psi(cl)$ is now given by:

$$u(c, cl) = u_b(c, cl) \cdot s(cl).$$

where, the final total weight update is:

$$s(cl) = \sum_{c_i \in \psi(cl)} u(c_i, cl).$$

We consider four options for the scale parameter:

None: The 'None' setting uses:

$$s(cl) = 1.$$

This setting is a logical baseline to compare to, as the standard dom/wdeg weight-update increments constraint weight by 1.

ClusterSize: The 'ClusterSize' setting uses:

$$s(cl) = |\psi(cl)|.$$

By scaling by the number of constraints in the cluster, we may be able to avoid penalizing large clusters.

SumΔwdeg: The 'SumΔwdeg' setting uses:

$$s(cl) = \sum_{c_i \in \psi(cl)} \Delta wdeg(c_i).$$

This setting is an attempt to account for the effort spent on the subproblem search. The more wipeouts are encountered by the subproblem search, the more we increment the weight of constraints in the cluster.

AvgΔwdeg: The 'AvgΔwdeg' setting uses:

$$s(cl) = \frac{\sum_{c_i \in \psi(cl)} \Delta wdeg(c_i)}{|\psi(cl)|}.$$

By averaging the $\Delta wdeg(\cdot)$ values, we remove the bias against clusters with many constraints.

5.3 Experimental Evaluation

To evaluate the effect of each parameter in our weight-update framework, we perform a factorial experiment testing all 16 combinations [Box *et al.*, 1978].

5.3.1 Setup

In our experiment, we consider three factors with a varying number of levels:

1. Occurrence: Always, OnWipeout
2. Distribution: Uniform, DualWeight
3. Scale: None, ClusterSize, SumΔwdeg, AvgΔwdeg

We test all 16 configurations of these three factors as shown Table 5.1. Additionally,

Table 5.1: The three factors tested

Factor	Level	Abbrev.
Occurrence	Always	Al
	OnWipeout	OW
Distribution	Uniform	Un
	DualWeight	DW
Scale	None	No
	ClusterSize	CS
	SumΔwdeg	SΔ
	AvgΔwdeg	AΔ

we run a test where we do not update constraint weights after enforcing minimality on a cluster. All tests enforce cluster minimality using PerTuple during search for the entire CSP. For the PerTuple algorithm, we use the best configuration from Chapter 4, that is, forward checking on the original dual graph and dangle identification on a minimal dual graph found with the MaxDeg heuristic.

We perform our experiments over instances taken from benchmarks in the XCSP library.[1] We randomly sample (up to) 10 instances from 198 different benchmark

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

problems to reach 1939 instances. We test 17 different configurations for a total of 32,963 runs.[2]

We run our experiments on a computer cluster of Intel Xeon E5-2670 2.60 GHz processors. Each execution is allocated 2 hours and 12GB of memory. To account for load variations on the cluster computer, we measure instruction count and convert it to runtime using a standardized measure of instructions per cycle and clock speed.

5.3.2 Results

In Table 5.2, we summarize the performance of PerTuple on the 17 tested configurations. We report

1. The number of instances completed by each configuration
2. The CPU time (in milliseconds) averaged over instances completed by at least one configuration. In the case that a configuration does not complete an instance, the CPU time is treated as the time limit (i.e., 7,200,000 ms).
3. The CPU time (in milliseconds) averaged over instances completed by all configuration. Thus, no timeouts/memouts are included in this average
4. The average number (averaged over instances completed by all configurations) of nodes visited by the search

In each of the last four columns, we typeset the best entry in boldface.

We perform a three-way, repeated measures, ANOVA test using a within-subject design in order to determine the effects of the three considered factors and their levels. We consider runtime as the dependent variable. We apply a log transformation to the

[2]The 17 different configurations correspond to the 16 configurations of Table 5.1 and the no weight-update strategy.

Table 5.2: Results summary for all tested weight-update strategies

Configuration	Instances completed	Average time (ms) by one	Average time (ms) by all	Avg NV by all
Cl-PerTuple	**836**	1,092,818.6	373,591.5	**80,137.6**
Cl-PerTuple-Al,Un,No	822	1,194,506.5	387,414.1	143,082.2
Cl-PerTuple-Al,Un,CS	825	1,122,870.6	**356,951.1**	270,076.2
Cl-PerTuple-Al,Un,SΔ	827	1,120,249.9	372,719.6	113,825.2
Cl-PerTuple-Al,Un,AΔ	817	1,163,415.3	371,427.5	88,400.9
Cl-PerTuple-Al,DW,No	820	1,206,371.1	386,567.9	91,022.1
Cl-PerTuple-Al,DW,CS	830	**1,071,235.5**	362,763.6	148,750.0
Cl-PerTuple-Al,DW,SΔ	824	1,129,008.0	366,410.2	112,567.6
Cl-PerTuple-Al,DW,AΔ	826	1,121,936.5	374,298.9	85,853.3
Cl-PerTuple-OW,Un,No	827	1,134,225.8	382,627.8	100,495.0
Cl-PerTuple-OW,Un,CS	827	1,124,292.0	375,400.4	166,142.2
Cl-PerTuple-OW,Un,SΔ	831	1,106,725.9	377,073.4	115,470.5
Cl-PerTuple-OW,Un,AΔ	827	1,145,740.9	384,524.3	108,860.1
Cl-PerTuple-OW,DW,No	822	1,153,398.2	383,868.7	108,397.5
Cl-PerTuple-OW,DW,CS	833	1,102,083.2	379,539.0	151,978.4
Cl-PerTuple-OW,DW,SΔ	835	1,093,759.3	385,395.9	109,622.8
Cl-PerTuple-OW,DW,AΔ	822	1,161,157.4	389,909.1	109,770.3

runtime data in order to achieve an almost normal distribution. Our data is right-censored because of the 2-hour time-limit. For this reason, our data can be considered only 'approximately normally distributed.' The results of the ANOVA test are given in Table 5.3.

Table 5.3 provides the following data: The effect name (effect), the degrees of freedom of the effect (DFn), the degrees of freedom of the error (DFd), the sum of squares of the effect (SSn), the sum of squares of the error (SSd), the F measure (F), the corresponding p-value (p), an indicator of the significance of the effect ($p<.05$),

Table 5.3: Weight update ANOVA results sorted by generalized eta squared

Effect	DFn	DFd	SSn	SSd	F	p	p<.05	ges
(Intercept)	1	890	3.26E+05	29323.1357	9890.70164	0	*	9.16E-01
Scale	3	2670	3.87E-01	102.10487	3.3754685	0.01766081	*	1.30E-05
Distribution:Scale	3	2670	1.37E-01	64.81172	1.8880243	0.12942453		4.60E-06
Occurrence:Scale	3	2670	1.34E-01	84.1384	1.4134284	0.23691293		4.47E-06
Distribution:Occurrence:Scale	3	2670	8.17E-02	64.92289	1.1204603	0.33939289		2.73E-06
Occurrence	1	890	3.59E-02	156.26385	0.2045297	0.65119937		1.20E-06
Distribution:Occurrence	1	890	2.26E-02	47.46371	0.4237791	0.51522538		7.56E-07
Distribution	1	890	1.82E-02	52.28022	0.3105038	0.57751128		6.10E-07

and the generalized eta-squared measure of effect size (ges). The table is sorted by effect size (i.e., ges). We see that only the Scale factor has any statistically significant impact on the runtime.

5.3.3 Discussion

Our results show that none of the strategies considered in this chapter provides any statistically significant improvement over not using any weight-update strategy. However, these explored strategies are only a selection of the possible such strategies. Previous research has shown improvements from using weight-update strategies suited for APOAC and RNIC as HLCs [Woodward and Choueiry, 2017]. As a reminder, the techniques proposed in that research are not adaptable to our context because they are based on induced neighborhoods.

Although our current investigations do not yield a promising strategy, we argue that our framework for the weight-update parameters may still be of use. It is our hope that we may draw attention to the need for an effective weight-update strategy for high-level consistencies.

Summary

In this chapter, we argued the need for effective weight-update strategies that are compatible with HLC. We proposed a framework for expressing such strategies based on three orthogonal parameters and introduced 16 variants. We empirically evaluated these strategies on benchmark problems.

Chapter 6

Dynamic Portfolio for Cluster-Based Minimality

In this chapter, we investigate the use of an algorithm portfolio that operates at the cluster level. The portfolio approach has roots in the Algorithm Selection Problem, which involves selecting the best algorithm to apply to a particular instance to maximize some performance metric [Rice, 1976]. Early work by Gomes and Selman [2001] used a portfolio of several algorithms running in parallel to exploit their complementarity to solve various combinatorial problems. Portfolios gained in popularity for both SAT and CSPs through solver competitions, with SATzilla [Xu *et al.*, 2008] winning the SAT Challenge 2012 and cpHydra [O'Mahony *et al.*, 2008] winning the 2008 Constraint Solver Competition.

In our context, we propose to design a portfolio to operate *dynamically* and *locally* on the clusters of a tree decomposition to select an algorithm, PerTuple or AllSol, if any, to enforce minimality on the cluster. A feature of our approach is that the portfolio operates at a fine grain, atomic level, of the search process, which, to the best of our knowledge, was never attempted before.

6.1 Collecting Training Data

The first step of training any machine learning classifier is to gather training data. To this end, we execute both AllSol and PerTuple on a set of 27,226 clusters sampled from tree decompositions of instances taken from 176 benchmarks in the XCSP library.[1] We execute each algorithm on each cluster for up to 30 minutes. We record the runtime for each algorithm.

In addition to the runtimes, we also collect a set of feature values for every cluster used in our training. We collect a total of 427 features for every cluster. Below we describe each of the features collected as well as the aggregation functions used to compute some of these features.

6.1.1 Features

We distinguish three types of features: general features computed on the CSP parameters of an instance, graph features computed on a number of graph structures, tree-decomposition features are computed on two types of tree decomposition of the CSP.

Many of the features compute a list of data points that are then reported using several aggregate functions presented in Section 6.1.2. Such features are indicated by an asterisk.

6.1.1.1 General Features

Below we list the general features of the CSP instance in the cluster:

1. *Variables*: The number of variables in the CSP.

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

2. *Constraints*: The number of constraints in the CSP.

3. *Values**: A data point per variable, reporting the number of values in the domain of the variable.

4. *Tuples**: A data point per constraint, reporting the number of tuples in the relation of the constraint.

5. *Arity**: A data point per constraint, reporting the arity of the constraint.

6. *Tightness**: A data point per constraint, reporting the tightness of the constraint (i.e., the number of forbidden tuples divided by the cardinality of the Cartesian product of the domains of the variables in the scope of the constraint).

7. *Relational linkage**: A data point per subtuple over the variables shared by two constraints (i.e., subscope), as a measure of the likelihood of the subtuple to appear in a solution. For every two relations R_i, R_j, let $V_{ij} = scope(R_i) \cap scope(R_j)$. $\forall R_k, scope(R_k) \supseteq V_{ij}, \forall x \in scope(R_k) \setminus V_{ij}$, the relational linkage of every subtuple $t_s \in \pi_{V_{ij}}(R_i \bowtie R_j)$ is computed as:

$$min_{R_k} \left(\frac{|\sigma_{t_s}(R_k)|}{\prod_x |domain(x)|} \right)$$

   where $\bowtie, \sigma, \pi$ are the relational operators join, selection, and projection, respectively.

8. *Coarse blocks*: The number of coarse blocks (see Section 3.4).

9. *Coarse blocks size**: A data point per coarse block, reporting the number of tuples in the coarse block.

10. *Subscopes*: The number of unique subscopes in the CSP.

11. *Subscope constraints**: A data point per subscope, reporting the number of constraints whose scope includes the subscope.

12. *Subscope partitions**: A data point per subscope, reporting the number of coarse blocks induced by the subscope on a relation whose scope includes the subscope.

6.1.1.2 Graph Features

For each of the graph features listed below, we collect the feature on each of the following graph types, namely, primal, incidence, dual, minimal dual, triangulated primal, triangulated dual, and triangulated minimal dual (see Section 2.3):

13. *Graph density*: The density of the graph given by $\frac{2 \cdot e}{v \cdot (v-1)}$, where v is the number of vertices and e is the number of edges in the graph.

14. *Graph degree**: A data point per graph vertex, reporting the degree of the vertex.

15. *Graph eccentricity**: A data point per graph vertex, reporting the eccentricity of the vertex. The eccentricity of a vertex is the maximum length of a shortest-path to another vertex in the graph.

16. *Graph clustering-coefficient**: A data point per graph vertex, reporting the clustering coefficient of the vertex. The clustering coefficient of a vertex is the density of the graph induced by the neighborhood of the vertex (excluding the vertex itself) on the considered graph.

17. *Graph triangles*:[2] The number of triangles (i.e., cliques of size three) in the graph.

[2] Graph triangles are computed only on the primal, dual, triangulated primal, and triangulated dual graphs of a cluster instance.

18. *Graph MCB cycles*:[3] The number of cycles in a Minimum Cycle Basis (MCB) of the graph.[4]

19. *Graph MCB cycle-size**:[3] A data point per cycle in an MCB of the graph, reporting the size of the cycle.

20. *Graph MCB compute-time*:[3] The time to compute an MCB in milliseconds.

6.1.1.3 Tree-Decomposition Features

We collect each of the features listed below on a tree decomposition of the primal graph and on that of the dual graph:

21. *Tree-decomposition clusters*: The number of clusters in the decomposition.

22. *Tree-decomposition depth*: The depth of the decomposition.

23. *Tree-decomposition leaves*: The number of leaf vertices in the decomposition.

24. *Tree-decomposition cluster-constraints**: A data point per cluster, reporting the number of constraints in the cluster.

25. *Tree-decomposition cluster-variables**: A data point per cluster, reporting the number of variables in the cluster.

[3] The MCB features are computed only on the dual, incidence, and minimal dual graphs.

[4] A *cycle basis* of a graph is a maximal set of cycles that are linearly independent (i.e., cycles in the basis cannot be obtained by taking the composition of other cycles in the basis) [Horton, 1987]. In a weighted graph the weight of a cycle in the graph is the sum of the weights of the edges in the cycle. A minimum cycle basis is a cycle basis where the sum of the weights of the cycles in the cycle basis is minimum. Informally, a minimum cycle basis is a minimum set of cycles that can generate all of the cycles of the graph. In the case of an unweighted graph, the weights of each edge is one, a minimum cycle basis has a minimum total length. An MCB is not unique. Algorithms for finding a minimum cycle basis are either exact or approximate, finding the minimum within some bound [Horton, 1987; Kavitha *et al.*, 2007; Mehlhorn and Michail, 2009; Amaldi *et al.*, 2010].

26. *Tree-decomposition separator-constraints**: A data point per separator (i.e., an overlap between two clusters), reporting the number of constraints in the separator.

27. *Tree-decomposition separator-variables**: A data point per separator (i.e., an overlap between two clusters), reporting the number of variables in the separator.

6.1.2 Aggregate Functions

For the features that compute a list of data points, we report aggregate values using the following functions:

1. *Minimum*
2. *Maximum*
3. *Mean*
4. *Median*
5. *Standard deviation*
6. *Coefficient of variation*
7. *Entropy*
8. *Sum*
9. *Product*
10. *Log of the product*

Coefficient of variation is the normalized standard deviation (i.e., $\frac{standardDeviation}{mean}$). The entropy of a multiset $\mathcal{X} = \langle X, m \rangle$ (the set X is the possible values in $\mathcal{X}$ and for all $x \in X$, $m(x)$ is the multiplicity of x in $\mathcal{X}$) is calculated by $H(\mathcal{X}) = -\sum_{x \in X} \frac{m(x)}{|\mathcal{X}|} \log\left(\frac{m(x)}{|\mathcal{X}|}\right)$. We compute the log of the product by summing the natural logarithm of each of the data points.

6.2 Decision-Tree Classifier

We choose to use a simple decision-tree classifier for our portfolio for several reasons:

1. It is easy to implement and use during a search procedure for solving a CSP
2. It is easy to interpret the decisions of the tree
3. It allows us to collect the values of the features in a partial and incremental manner, on an 'as-needed basis,' as we are moving along a given branch of the tree.

6.2.1 Labels and Weights for Classification

We use the classifier to select between three decisions for each processed cluster, i.e., we classify a cluster as

1. 'AllSol' when AllSol is faster than PerTuple
2. 'PerTuple' when PerTuple is faster than AllSol
3. 'Neither' when neither algorithm is able to complete within the set time-threshold of 10,000 milliseconds.

We choose a ten-second cutoff because we use a runtime limit of one second per cluster during search. Indeed, we deliberately make the classifier more 'forgiving' in terms of the time cutoff in order to allow it to process instances that may need a processing time close to the one-second limit. We choose a timeout of one second per cluster because, based on the results of the 27,226 training instances shown in Figure 6.1, this value strikes a good balance between completing clusters and not spending excessive time on any one cluster. Indeed, 68% of our training instances are able to complete

with one of the two algorithms in under a second. Increasing the limit by an order of magnitude would allow the completion of only 8% more of our training instances.

To every instance, i, in the collected training dataset, we assign a label $l(i)$ and weight $w(i)$ according to the following scheme where $cpu_P(i)$ and $cpu_A(i)$ are the runtimes (in milliseconds) of PerTuple and AllSol, respectively:

$$l(i) = \begin{cases} \text{`PerTuple'} & cpu_P(i) \leq cpu_A(i),\ cpu_P(i) \leq 10{,}000 \\ \text{`AllSol'} & cpu_A(i) \leq cpu_P(i),\ cpu_A(i) \leq 10{,}000 \\ \text{`Neither'} & cpu_P(i) > 10{,}000,\ cpu_A(i) > 10{,}000 \end{cases}$$

and

$$w(i) = \begin{cases} \lceil \log_{10}(\frac{\min(cpu_A(i),10{,}000)}{cpu_P(i)+1} + 1) \rceil & l(i) = \text{`PerTuple'} \\ \lceil \log_{10}(\frac{\min(cpu_P(i),10{,}000)}{cpu_A(i)+1} + 1) \rceil & l(i) = \text{`AllSol'} \\ \lceil \log_{10}(\frac{\min(cpu_P(i),cpu_A(i))}{10{,}000}) \rceil & l(i) = \text{`Neither'} \end{cases}$$

The weight values range over {0,1,2,3}. We design the weighting scheme to emphasize instances for which one selection greatly outperforms the other options. Figure 6.1 shows our training dataset in a scatter plot where the horizontal axis represents CPU time of AllSol and the vertical axis the cpu time of PerTuple. This plot gives:

- The labels of the data points shown with color, namely, purple for 'PerTuple', green for 'AllSol', and black for 'Neither'
- The weights of each data point shown by the size of its point, where the largest points being of weight three.

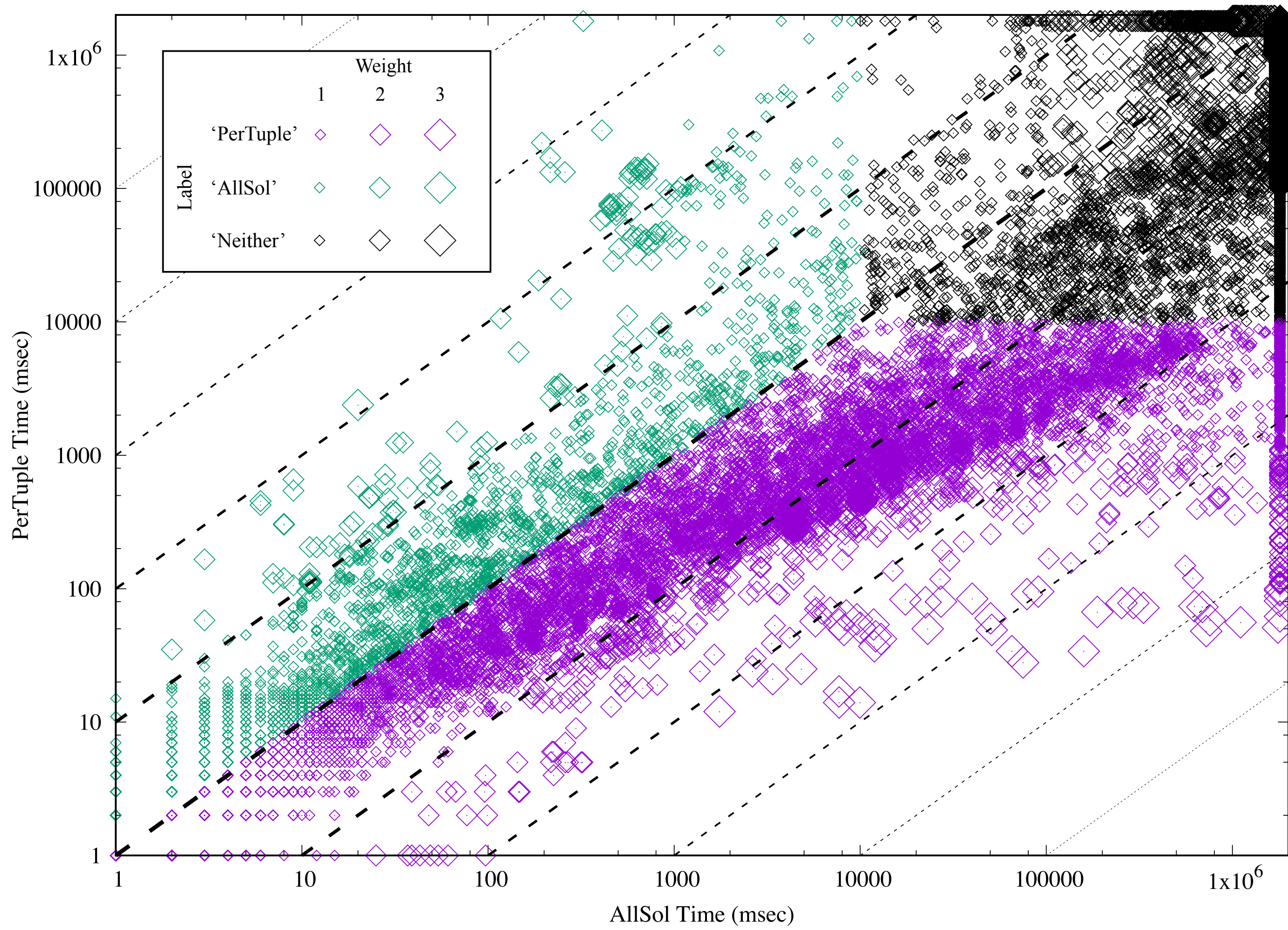


Figure 6.1: The weighted and labeled training data

6.2.2 Training

After labeling and weighting the data, we use the J48 algorithm from the WEKA data-mining software to construct the decision tree [Hall *et al.*, 2009].[5] Initially, we used the entire set of features listed in Section 6.1.1 and used the default J48 parameters. However, some of the features proved to be too costly to compute at every cluster visit and we removed them from consideration.[6] The features that we do compute are: variables, constraints, values, tuples, arity, tightness (Section 6.1.1.1).

With this restricted set of features and using the default parameters of the J48 algorithm, the generated decision trees were often quite large and with only a small number of training instances located at each leaf. To avoid overfitting the data, we adjusted the parameters of J48 requiring, for any leaf node, a minimum requirement of 1,000 training instances (out of 32,518 total instances after accounting for instance weight) .

Our decisions for using a limited feature set and enforcing a minimum number of instances per leaf are empirically motivated following a series of trial-and-error experiments. This process results in a lightweight decision tree with easily computed features and an intuitive interpretation.

6.2.3 Training Results

While our classification approach may appear to be relatively simple, it yields reasonable results with a 10-fold cross validation and serves as a useful proof-of-concept for our idea of a cluster-level portfolio. Indeed, Table 6.1 presents our results when

[5] `https://www.cs.waikato.ac.nz/ml/weka/`

[6] The removed features are: all graph features (Section 6.1.1.2), all tree-decomposition features (Section 6.1.1.3), relational linkage, all two coarse-blocks, and three subscope features (Section 6.1.1.1). Further, concerning the aggregate functions (Section 6.1.2), we removed product in favor of log product. Finally, for the general features 'values' and 'tuples,' we removed the 'sum' aggregate function.

trained using 10-fold cross validation. In this table, we report the number of in-

Table 6.1: Classifier training results

Class	Instances	TP	FP	Precision	Recall	F-Measure
‘AllSol’	2,902	0.429	0.029	0.595	0.429	0.499
‘PerTuple’	10,494	0.844	0.106	0.791	0.844	0.817
‘Neither’	19,122	0.928	0.111	0.923	0.928	0.926
Weighted average		0.856	0.102	0.851	0.856	0.852

stances in each of the three classes. For each class, we report the true-positive (TP) rate, false-positive (FP) rate, precision, recall, and F-measure. Finally, we report the weighted average of each metric, weighted by the number of instances per class. Our classifier attains an accuracy of 85.6% (which is the weighted average true-positive rate). It is able to achieve a reliable F-Measure on the ‘Neither’ instances, a reasonable F-Measure on the ‘PerTuple’ instances, and a somewhat poor F-Measure on the ‘AllSol’ instances.

Figure 6.2 shows the confusion matrix of our classifier. The vertical axis of the

Figure 6.2: Confusion matrix of the decision-tree classifier

matrix indicates the true label of an instance. The horizontal axis indicates the pre-

dicted label. Instances falling on the diagonal are correctly classified by the decision tree. The shading of each cell corresponds to the total number of instances in the given category. The number in each cell shows the percentage of instances within a given row that were predicted to have a given label (i.e., each row is normalized). The most common misclassification is misclassifying an 'AllSol' instance as a 'PerTuple' instance, followed closely by misclassifying 'Neither' as 'PerTuple' and vice versa. This poor performance for 'AllSol' is likely due to the class imbalance (i.e., the relatively few instances in the 'AllSol' class).

6.2.4 Trained Decision Tree

Figure 6.3 shows the generated decision-tree classifier, which we use as a portfolio in Section 6.3. The tree uses four features: Tuples_logProd, Tuples_max, Tuples_min, and Tightness_max. Each of these four features is updated throughout the course of

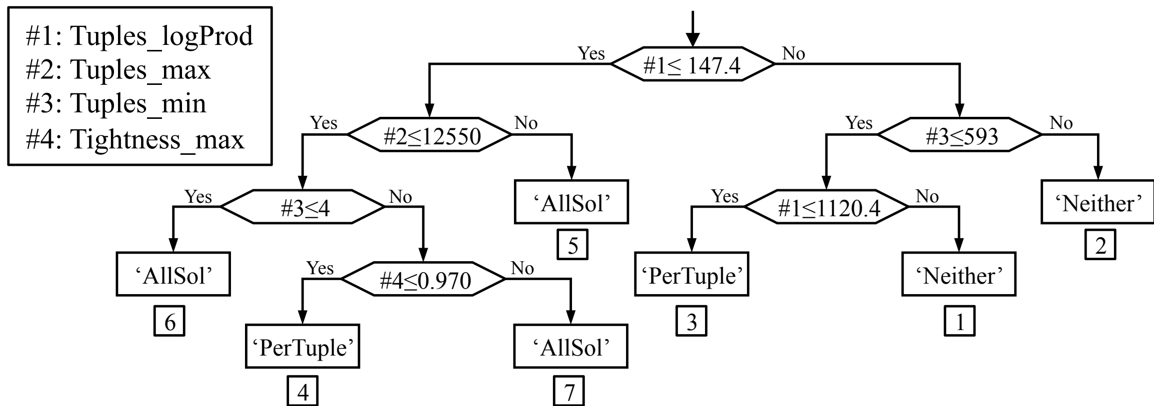


Figure 6.3: The trained decision-tree classifier

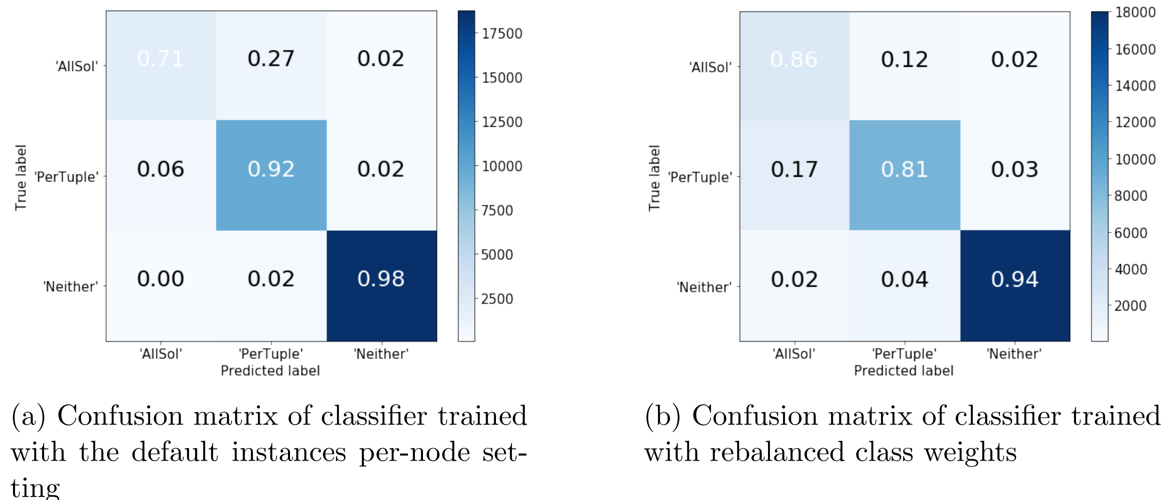
search only as the relations are filtered.

The product of the size of the domains (relations in the case of the dual CSP) is an often used estimate for the size of the search space. Thus, Tuples_logProd is a logical choice for the root of the tree. The tree selects 'Neither' if the problem is

very large (leaf 1) or somewhat large but with no small relations (leaf 2). It selects 'PerTuple' if the problem is somewhat large (leaf 3). If the problem is small, the tree selects 'PerTuple' (leaf 4) unless one of the following three conditions is met. If there is at least one particularly large relation (leaf 5), or one particularly small relation (leaf 6), or a relation that is extremely tight (leaf 7), the tree selects 'AllSol'. As stated above, the values are collected and computed one a 'as-needed' base while traversing a given branch of the tree.

6.2.5 Alternate Classifiers

We train several alternate classifiers in an attempt to improve the classification accuracy of the 'AllSol' instances. Figure 6.4 shows the confusion matrices of two of alternate classifiers:



(a) Confusion matrix of classifier trained with the default instances per-node setting

(b) Confusion matrix of classifier trained with rebalanced class weights

Figure 6.4: Confusion matrices of two alternate classifiers

- We train the classifier shown in Figure 6.4a using the J48 algorithm with a confidence factor of 0.01 (resulting in stronger pruning of the decision tree than the default value of 0.25) and with a minimum number of instances per leaf of

2, the default value. Reducing the minimum number of instances per leaf yields a much larger tree than our original decision tree.

- We train the classifier shown in Figure 6.4b using J48 with a confidence factor of 1×10^{-6}, a minimum number of instances per leaf of 100, and class rebalancing on the training dataset. Class rebalancing adjusts the instance weights such that all classes have the same total weight, increasing the representation of the 'AllSol' instances.

Both classifiers improve the classification accuracy of 'AllSol' and slightly improve that of 'Neither'. In the case of the classifier trained with class rebalancing, the misclassification of 'PerTuple' increases, likely due to the overrepresentation of 'AllSol' in the rebalanced training data. Although both alternate classifiers attain better classification accuracy than our original classifier in 10-fold cross validation (93.7% and 89.2% respectively compared to 85.6%), this improvement does not translate to better search performance on benchmark instances. The disconnect between classification and search performance may be attributed to several factors: the one-second time limit imposed when processing a cluster during search, the fact that PerTuple performs partial filtering when interrupted while AllSol cannot, and the fact that the clusters encountered during search likely have different properties than the clusters obtained from the initial tree decomposition. We hope to address these limitations in future iterations of our portfolio. In the experimental evaluations below, we consider only the original decision-tree classifier described in Section 6.2.4.

6.3 Experimental Evaluation

In this section, we evaluate the portfolio of Section 6.2.4 to maintain cluster-level minimality during search (Cl-Portfolio).

6.3.1 Setup

Our experiment compares the performance of five algorithms under two ordering heuristics. The five algorithms are:

1. STR2 maintains Generalized Arc Consistency (GAC) during search using the algorithm STR2 for GAC [Lecoutre, 2011]. We choose to use the STR2 algorithm to enforce GAC because, in addition to filtering the domains, STR2 also filters the relations, which makes it compatible with algorithms that enforce relational consistencies such as AllSol and PerTuple.

2. Cl-PerTuple enforces cluster minimality using PerTuple. It first executes STR2 on the entire problem, then PerTuple on the clusters (with a one-second time limit per cluster) along the max-clique ordering and until reaching a fixpoint. Finally, it executes STR2 on the entire problem again.

3. Cl-AllSol enforces cluster minimality using AllSol. It first executes STR2 on the entire problem, then AllSol on the clusters (with a one-second time limit per cluster), along the max-clique ordering and until reaching a fixpoint. Finally, it executes STR2 on the entire problem again.

4. Cl-Random enforces cluster minimality. It first executes STR2 on the entire problem, then it randomly decides, for each cluster along the max-clique ordering and until reaching a fixpoint, to enforce, with a one-second time limit per

cluster, PerTuple, AllSol, or neither, on each cluster. Finally, it executes STR2 on the entire problem again.

5. Cl-Portfolio enforces cluster minimality. It first executes STR2 on the entire problem, then, using our classifier, it decides, for each cluster along the max-clique ordering and until reaching a fixpoint, to enforce, with a one-second time limit per cluster, PerTuple, AllSol, or neither, on each cluster. Finally, it executes STR2 on the entire problem again.

We test each algorithm using both dom/deg and dom/wdeg [Boussemart *et al.*, 2004]. We report the results with dom/deg in order to evaluate the improved filtering power of our approach without the interference of weight updates in accordance with the common practice in the study of high-level consistency [Balafrej *et al.*, 2015; Paparrizou and Stergiou, 2016; Paparrizou and Stergiou, 2017]. However, we also report the results with dom/wdeg because it is currently the most effective ordering heuristic. Further, we show that there are instances unsolvable with STR2 (and dom/wdeg) but that are solvable with our technique.[7]

We perform our experiments over instances taken from benchmarks in the XCSP library.[8] We conduct the analysis and discussion over a set of benchmarks selected based on the following two criteria:[9]

1. High-level consistencies should be evaluated on difficult problems where GAC-level algorithms struggle or fail.

2. The goal of this thesis is to improve the performance of minimality algorithms and to execute the right algorithm at any particular point.

[7]Note that, despite our efforts to adapt the weight updates of dom/wdeg to the context of cluster minimality (see Chapter 5), none of the strategies tested proved effective and are ignored in this experiment.

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

[9]We provide the complete results in Appendix A.

For this reason, we exclude all easy benchmarks that can be solved with STR2. Based on the above, we identify:

- For dom/deg, 47 benchmarks of a total 1066 instances (Section 6.3.2).
- For dom/wdeg, 21 benchmarks of a total 299 instances (Section 6.3.3).

We execute our experiments on a computer cluster of Intel Xeon E5-2670 2.60 GHz processors. Each run is allocated 2 hours and 12GB of memory. To account for load variations on the cluster computer, we measure instruction count and convert it to runtime using a standardized measure of instructions per cycle and clock speed.

6.3.2 Cluster-Minimality Algorithms Using dom/deg

In Table 6.2, we show the benchmarks that benefit from enforcing minimality using the dom/deg ordering heuristic. The table displays, for each algorithm, the number of benchmarks the algorithm is best on, followed by a listing of the benchmarks. Cl-Portfolio performs best on significantly more benchmarks than the other three cluster-minimality algorithms.

Table 6.3 summarizes the performance of all five tested algorithms on the instances of the considered benchmarks. The table header indicates the number of instances completed by at least one of the algorithms and the total number of instances considered. The table provides, for each algorithm:

1. The number of instances completed by the algorithm.
2. The number of instances completed in a backtrack-free manner.
3. The number of instances that the algorithm solved with the fewest number of node visits than all other algorithms, considering only instances completed by all algorithms.

Table 6.2: Benchmarks where a given algorithm performs best (dom/deg)

Algorithm	**# best**	**Benchmark**
Cl-PerTuple	7	bqwh-18-141, composed-25-10-20, haystacks, pigeons, QCP-25, queenAttacking, rlfapScens11
Cl-AllSol	7	aim-100, aim-200, dubois, mug, jobShop-enddr1, modifiedRenault, super-jobShop-enddr1
Cl-Random	10	cril, golombRulerArity4, graceful, sgb-queen, QCP-10, QCP-15, QCP-20, QWH-20, QWH-25, rlfapScensMod
Cl-Portfolio	23	BH-4-4, composed-25-1-2, composed-25-1-25, composed-25-1-40, composed-25-1-80, composed-75-1-2, composed-75-1-25, composed-75-1-40, composed-75-1-80, ehi-85, ehi-90, full-insertion, leighton-15, sgb-book, sgb-games, os-taillard-5, pseudo-aim, queensKnights, rand-2-23, rand-2-24, rlfapGraphsMod, super-jobShop-enddr2, super-os-taillard-4

Table 6.3: Performance summary using dom/deg

Algorithm	**Instances** Comp. by one: 861 Total: 1066						**Avg time (ms)**
	Completed	**Timeout**	**Memout**	**BT-free**	**Min(#NV)**	**Fastest**	
STR2	623	407	36	118	42	327	2,274,935.0
Cl-PerTuple	779	222	65	601	534	103	884,289.6
Cl-AllSol	774	237	55	576	491	48	999,397.0
Cl-Random	761	211	94	561	414	76	1,061,008.4
Cl-Portfolio	791	226	49	601	523	266	765,517.2

Portfolio selection: 56%P 37%A 7%N

4. The number of instances that the algorithm solved the fastest.

5. The runtime, in milliseconds, averaged over all instances that at least one algorithm completed. If an algorithm does not complete an instance, the CPU time is considered to be the time limit (i.e., 7,200,000 ms).

Finally, the last row shows the portfolio's percentages of selecting: 'PerTuple' (%P), 'AllSol', (%A), and 'Neither' (%N), averaged over all instances. For each of the last five columns, we highlight with a box the best value in the column.

Cl-Portfolio completes the most instances and achieves the lowest average runtime. Cl-Portfolio ties with Cl-PerTuple for the most instances solved backtrack-free. It is competitive with Cl-PerTuple for the number of instances solved in the fewest node visits. Among the four minimality algorithms, Cl-Portfolio results in the fewest number of memouts because our implementation delays building the costly data structures until they are needed. Among the cluster-minimality algorithms, it has, by far, the most instances solved the fastest. Cl-PerTuple has somewhat better performance than Cl-AllSol and we see that the portfolio chooses 'PerTuple' nearly twice as often as 'AllSol'.

Figure 6.5 is a cactus plot of the five algorithms, displaying the cumulative number of instances completed over time. At nearly every runtime, Cl-Portfolio has the most instance completions. Between 1,000 and 10,000 milliseconds, STR2's number of instance completions jumps ahead of those of the cluster-minimality algorithm, but then quickly (i.e., for more difficult instances) falls behind.

Table 6.4 reports, per benchmark, the results using the dom/deg ordering heuristic. The first row summarizes the results in the table (also reported in Table 6.3). The first column indicates the benchmark, the number of instances included in the table (i.e., instances solved by at least one algorithm), and the total number of instances in

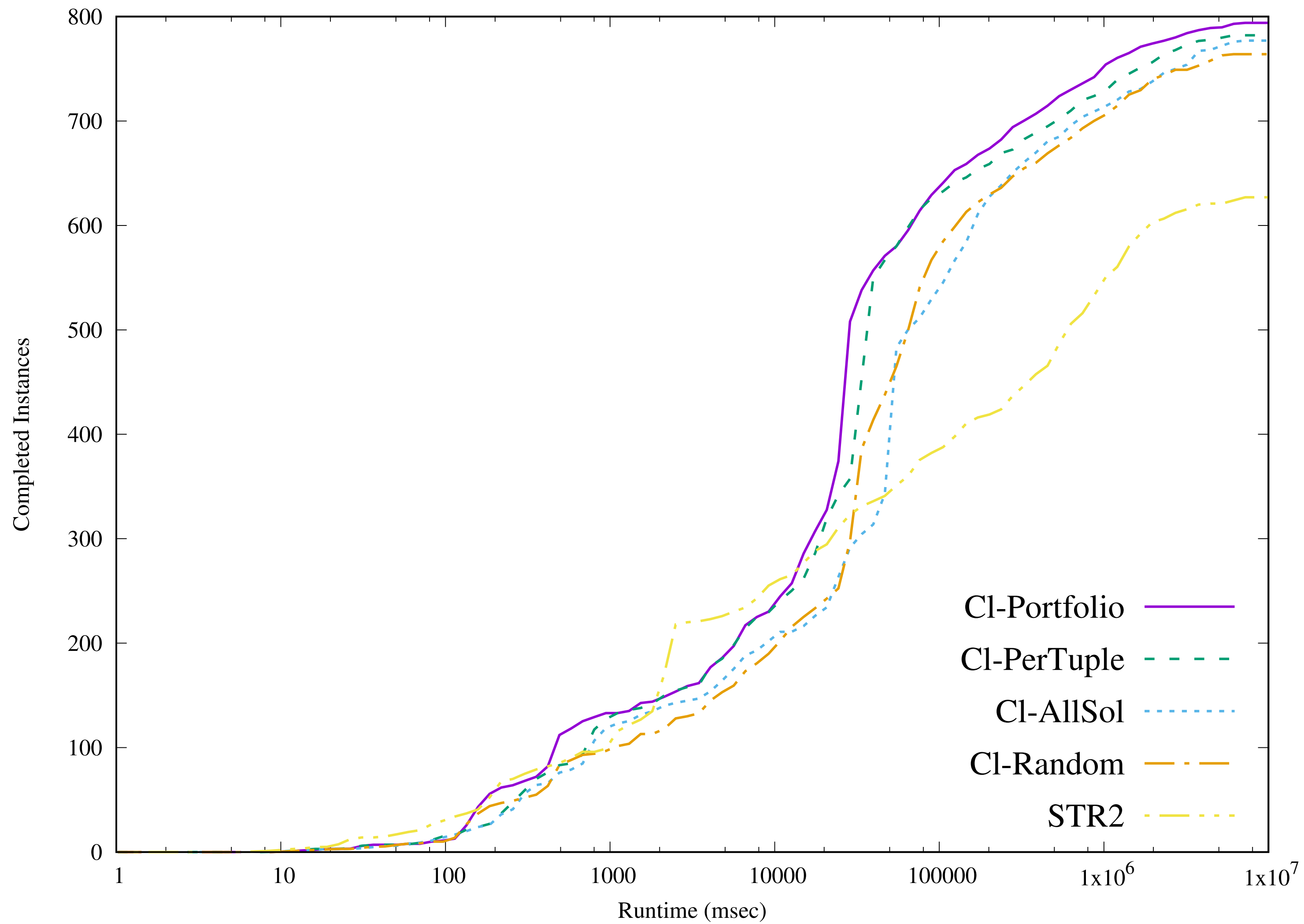


Figure 6.5: Instance completions over time with dom/deg

the benchmark. The next five columns report the results for a given algorithm: both the runtime (in milliseconds) averaged over instances completed by at least one algorithm and the number of instances completed by the given algorithm. When an an algorithm does not complete an instance, the CPU time is treated as the time limit (i.e., 7,200,000 ms). The last three columns report how often the portfolio selects 'PerTuple', 'AllSol', or 'Neither', respectively.

In many of the benchmarks, Cl-Portfolio uses some combination of PerTuple and AllSol in order to outperform both Cl-PerTuple and Cl-AllSol. On most of the benchmarks, the portfolio does not select 'Neither'. However, in the cases where it selects 'Neither', Cl-Portfolio usually outperforms all other algorithms (see ehi-85, ehi-90, leighton-15, os-taillard-5, queensKnights, rand-2-23, rand-2-24, super-jobShop-enddr1, super-os-taillard-4).

Table 6.4: Per-benchmark performance using dom/deg

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
Summary	2,274,935.0	884,289.6	999,397.0	1,061,008.4	**765,517.2**	56%	37%	7%
861/1066	623	779	774	761	791			
aim-100	2,289,850.8	1,280,598.0	**1,203,669.9**	1,992,658.4	1,258,765.3	9%	91%	0%
24/24	18	20	21	18	20			
aim-200	5,268,448.1	5,117,181.1	**4,886,498.9**	5,793,812.7	5,167,170.4	18%	74%	8%
24/24	8	7	8	5	7			
BH-4-4	7,200,000.0	3,808.2	3,500,493.7	3,869.8	**3,560.9**	28%	72%	0%
10/10	-	10	10	10	10			
bqwh-18-141	97,353.0	**50,069.5**	170,490.2	105,331.7	51,660.6	25%	75%	0%
100/100	100	100	100	100	100			
composed-25-1-2	7,200,000.0	197.0	202.0	121.3	**121.1**	90%	10%	0%
10/10	-	10	10	10	10			
composed-25-1-25	7,200,000.0	250.9	258.2	1,402.5	**143.5**	90%	10%	0%
10/10	-	10	10	10	10			
composed-25-1-40	7,200,000.0	280.7	298.5	1,332.1	**164.3**	87%	13%	0%
10/10	-	10	10	10	10			
composed-25-1-80	4,140,176.6	432.8	836.8	898.8	**289.0**	100%	0%	0%
10/10	5	10	10	10	10			
composed-25-10-20	3,244,168.3	**58,576.1**	148,372.1	79,947.7	60,692.4	34%	66%	0%
10/10	6	10	10	10	10			
composed-75-1-2	7,200,000.0	712.7	733.3	3,049.3	**415.4**	88%	12%	0%
10/10	-	10	10	10	10			

Table 6.4: Per-benchmark performance using dom/deg (continued)

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
composed-75-1-25	7,200,000.0	760.6	782.9	3,405.6	**440.8**	92%	8%	0%
10/10	-	10	10	10	10			
composed-75-1-40	7,200,000.0	812.9	838.7	1,504.0	**463.0**	94%	6%	0%
10/10	-	10	10	10	10			
composed-75-1-80	5,040,063.1	1,413.2	2,945.8	1,949.8	**1,057.8**	96%	4%	0%
10/10	3	10	10	10	10			
cril	5,252,506.9	3,108,408.4	4,115,732.2	**3,106,661.9**	3,107,536.9	68%	32%	0%
7/8	2	4	3	4	4			
dubois	1,580,785.7	1,593,133.9	**1,573,624.5**	2,462,762.2	1,965,018.6	0%	100%	0%
7/13	7	7	7	6	6			
ehi-85	446,748.4	29,337.8	41,181.1	35,871.3	**22,187.2**	81%	0%	19%
100/100	100	100	100	100	100			
ehi-90	976,702.8	31,621.1	45,320.0	40,051.3	**23,717.2**	80%	0%	20%
100/100	95	100	100	100	100			
golombRulerArity4	249,900.7	249,418.1	249,401.3	**249,376.0**	249,377.0	63%	38%	0%
2/14	2	2	2	2	2			
graceful	2,811,900.0	2,537,964.9	2,612,469.2	**2,534,769.2**	2,551,260.6	57%	43%	0%
3/4	2	2	2	2	2			
full-insertion	2,488,182.9	2,604,259.8	2,632,382.1	3,152,287.3	**2,410,508.3**	30%	70%	0%
32/41	21	21	21	19	22			
leighton-15	3,278,469.2	2,064,039.7	3,357,295.0	2,465,225.9	**1,150,907.7**	70%	7%	23%
9/26	5	7	5	6	8			

Table 6.4: Per-benchmark performance using dom/deg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
mug	3,600,012.2	19,480.1	**19,059.5**	346,696.4	24,004.2	11%	89%	0%
8/8	4	8	8	8	8			
sgb-book	1,672,291.7	436,830.9	726,704.7	1,340,712.8	**435,278.1**	83%	17%	0%
23/26	19	22	22	19	22			
sgb-games	3,604,888.0	16,446.9	1,840,626.6	21,023.2	**15,562.5**	69%	31%	0%
4/4	2	4	3	4	4			
sgb-queen	2,894,074.8	1,996,657.0	1,954,690.3	**1,798,709.2**	1,982,455.7	68%	14%	18%
15/50	9	11	12	12	11			
haystacks	3,215,129.3	**710,220.7**	2,057,686.4	1,715,194.3	750,043.9	70%	30%	0%
7/51	4	7	5	6	7			
jobShop-enddr1	808,161.5	2,025,407.3	**579,259.0**	1,634,787.8	1,091,202.1	36%	61%	3%
9/10	8	9	9	8	9			
modifiedRenault	248,697.8	39,287.4	**36,821.1**	224,612.7	37,430.9	58%	42%	0%
50/50	49	50	50	49	50			
os-taillard-5	3,715,048.7	5,003,526.2	4,043,962.7	4,819,209.8	**3,444,165.4**	38%	39%	23%
19/30	10	8	10	8	12			
pigeons	256,218.6	**193,237.3**	422,270.0	313,310.9	202,913.2	81%	19%	0%
13/24	13	13	13	13	13			
pseudo-aim	1,952,036.6	1,704,745.6	1,712,151.9	2,350,053.5	**1,703,246.6**	9%	91%	0%
48/48	39	37	37	33	37			
QCP-10	178,137.2	58,158.1	87,559.5	**56,489.4**	62,848.9	17%	83%	0%
15/15	15	15	15	15	15			

Table 6.4: Per-benchmark performance using dom/deg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
QCP-15	3,842,598.7	128,626.1	132,850.9	**98,666.9**	115,090.6	24%	71%	5%
15/15	8	15	15	15	15			
QCP-20	7,200,000.0	2,273,913.0	2,351,278.5	**1,138,250.2**	2,158,952.1	42%	52%	6%
11/15	-	9	9	11	9			
QCP-25	7,200,000.0	**838,179.4**	1,269,032.8	2,274,623.1	2,254,291.6	100%	0%	0%
4/15	-	4	4	3	3			
queenAttacking	1,829,407.8	**360,437.5**	582,309.6	501,667.5	1,803,037.9	74%	14%	12%
4/10	3	4	4	4	4			
queensKnights	1,543,653.4	1,518,278.8	1,743,128.7	1,654,649.1	**1,517,909.0**	67%	28%	5%
10/18	8	8	8	8	8			
QWH-20	6,564,135.3	492,676.5	340,291.2	**298,351.1**	447,840.4	11%	85%	4%
10/10	1	10	10	10	10			
QWH-25	7,200,000.0	7,200,000.0	5,130,443.6	**4,774,416.0**	7,200,000.0	-	-	-
2/10	-	-	1	1	-			
rand-2-23	432,574.9	446,742.5	1,713,566.1	1,492,744.4	**379,067.9**	93%	0%	7%
10/10	10	10	10	10	10			
rand-2-24	956,969.7	859,555.3	5,101,956.2	3,689,564.9	**720,362.5**	90%	0%	10%
10/10	10	10	5	9	10			
rlfapGraphsMod	4,201,029.1	462,767.4	945,847.3	524,672.7	**449,657.1**	46%	54%	0%
12/12	5	12	11	12	12			
rlfapScens11	7,200,000.0	**969,702.2**	4,517,589.5	2,726,426.3	999,830.0	94%	4%	2%
8/12	-	7	3	5	7			

Table 6.4: Per-benchmark performance using dom/deg (continued)

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
rlfapScensMod	2,772,517.2	251,109.3	770,869.3	**198,637.2**	255,753.7	54%	46%	0%
13/13	8	13	12	13	13			
super-jobShop-enddr1	4,810,078.1	5,252,276.5	**3,720,746.0**	7,200,000.0	3,988,823.9	76%	17%	7%
3/10	1	2	2	-	2			
super-jobShop-enddr2	3,619,861.2	7,200,000.0	4,704,472.8	7,200,000.0	**3,237,503.5**	74%	18%	8%
2/6	1	-	1	-	2			
super-os-taillard-4	1,623,196.1	2,907,544.3	1,286,475.3	2,028,325.5	**1,202,340.0**	36%	36%	27%
28/30	22	21	26	23	27			

6.3.3 Cluster-Minimality Algorithms Using dom/wdeg

When studying high-level consistency, results are commonly reported only for dom/deg to ensure results comparability. However, for the sake of completeness, we provide the results also for dom/wdeg. In Table 6.5, for each of the cluster-minimality algorithms, we list the benchmarks on which the algorithm performs best.

Table 6.5: Benchmarks where a given algorithm performs best (dom/wdeg)

Algorithm	# best	Benchmark
Cl-PerTuple	4	cril, pigeons, QCP-25, rlfapScens11
Cl-AllSol	1	mug
Cl-Random	3	QCP-20, QWH-20, QWH-25
Cl-Portfolio	13	BH-4-4, composed-25-1-2, composed-25-1-25, composed-25-1-40, composed-75-1-2, composed-75-1-25, composed-75-1-40, leighton-15, sgb-book, sgb-games, haystacks, rand-2-23, rand-2-24

Table 6.6 shows that Cl-PerTuple outperforms all other in terms of instance completions, solved backtrack-free, and with the fewest number of nodes visited.

However, Cl-Portfolio achieves comparable results in all these categories, and, furthermore, is the fastest on a significantly larger number of instances than any other algorithm including Cl-PerTuple. As a result, Cl-Portfolio ends up beating all five tested algorithm in terms of average time.

In terms of average CPU time, the gap between Cl-PerTuple and Cl-AllSol is significantly larger for dom/wdeg (Table 6.6) than for dom/deg (Table 6.3). Consequently, Cl-Portfolio selects 'PerTuple' nearly three times as often as 'AllSol'.

Figure 6.6 shows the cactus plot of the five algorithms using dom/wdeg. Though Cl-PerTuple ends with one more completion than Cl-Portfolio, Cl-Portfolio achieves as many as or more completions than Cl-PerTuple throughout most of the chart.

Table 6.6: Performance summary using dom/wdeg

Algorithm	Instances (Comp. by one: 196, Total: 299)						Avg time (ms)
	Completed	Timeout	Memout	BT-free	Min(#NV)	Fastest	
STR2	169	115	15	16	15	44	1,252,525.9
Cl-PerTuple	190	78	31	139	54	22	513,501.6
Cl-AllSol	175	95	29	118	37	1	1,426,762.8
Cl-Random	183	67	49	127	36	35	971,294.9
Cl-Portfolio	189	89	21	135	46	93	502,754.6

Portfolio selection: 73%P 24%A 3%N

Table 6.7 reports the results per benchmark using the dom/wdeg ordering heuristic. While the portfolio does not select 'AllSol' often, when it does, the selection generally is advantageous. On the mug and QWH-20 benchmarks, Cl-AllSol outperforms Cl-PerTuple and the portfolio chooses to use AllSol the majority of the time. On the BH-4-4, composed, sgb-book, sgb-games, and haystacks benchmarks, occasional use of AllSol allows it to outperform Cl-PerTuple.

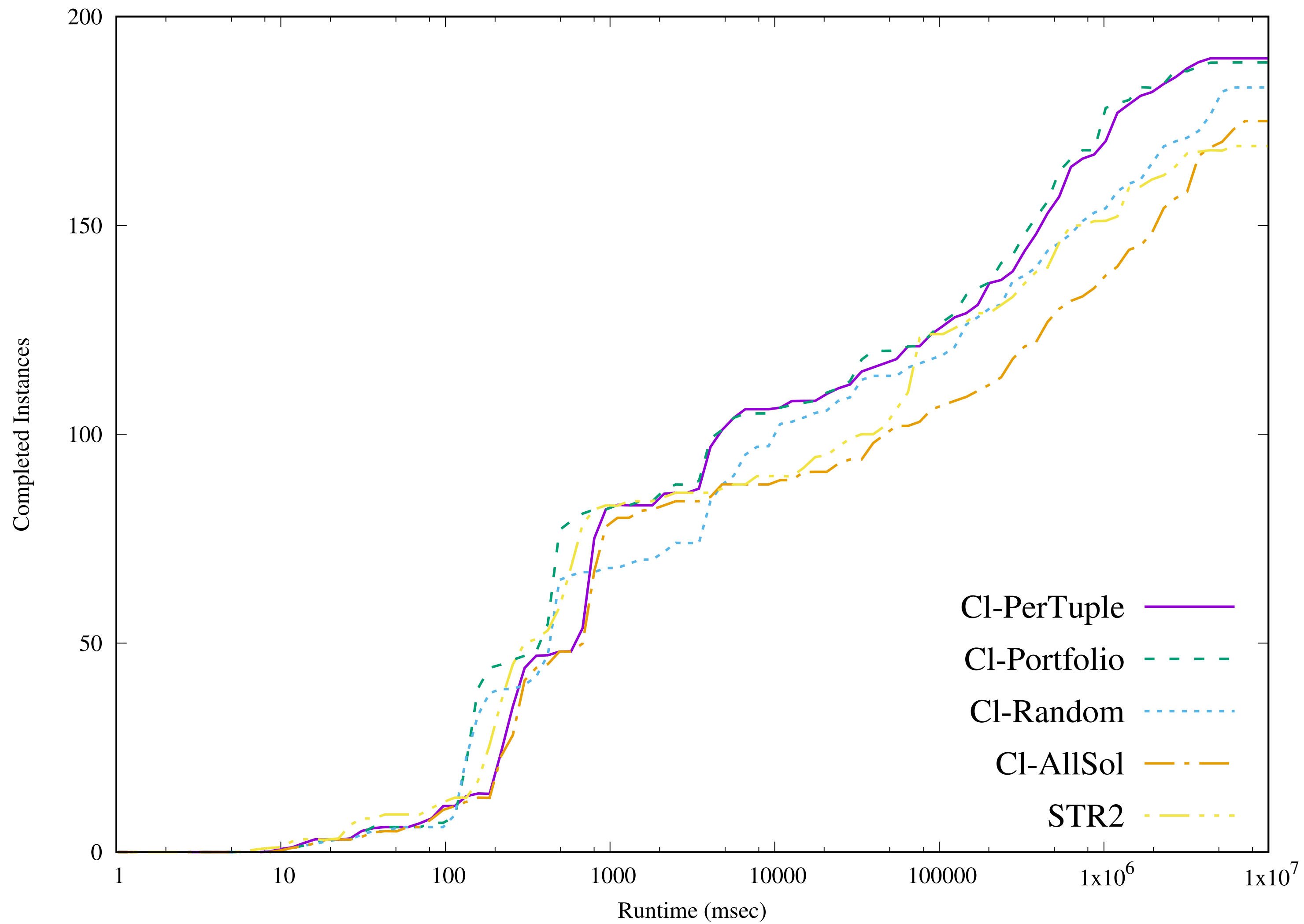


Figure 6.6: Instance completions over time with dom/wdeg

Table 6.7: Per-benchmark performance using dom/wdeg

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
Summary	1,252,525.9	513,501.6	1,426,762.8	971,294.9	**502,754.6**	73%	24%	3%
196/299	169	190	175	183	189			
BH-4-4	70,069.1	3,804.9	3,921,972.9	3,898.0	**3,537.8**	28%	72%	0%
10/10	10	10	9	10	10			
composed-25-1-2	167.1	196.9	202.2	122.0	**121.5**	90%	10%	0%
10/10	10	10	10	10	10			
composed-25-1-25	213.8	251.1	259.0	1,434.1	**144.1**	90%	10%	0%
10/10	10	10	10	10	10			
composed-25-1-40	244.2	281.3	299.3	1,392.0	**165.2**	87%	13%	0%
10/10	10	10	10	10	10			
composed-75-1-2	471.9	714.3	735.0	3,097.8	**417.1**	88%	12%	0%
10/10	10	10	10	10	10			
composed-75-1-25	586.7	761.8	784.6	3,565.3	**442.7**	92%	8%	0%
10/10	10	10	10	10	10			
composed-75-1-40	634.1	814.8	840.4	1,501.6	**464.6**	94%	6%	0%
10/10	10	10	10	10	10			
cril	2,765,996.6	**625,532.0**	1,726,567.6	630,008.0	675,353.3	44%	42%	14%
7/8	5	7	6	7	7			
leighton-15	1,587,880.6	1,960,107.9	3,356,928.3	2,464,417.4	**1,029,962.0**	69%	7%	25%
9/26	8	7	5	6	8			
mug	3,600,012.7	1,471,388.5	**1,238,042.7**	3,237,005.5	1,317,636.5	12%	88%	0%
8/8	4	7	7	5	7			

Table 6.7: Per-benchmark performance using dom/wdeg (continued)

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark # instances	**Avg time** completed	**Avg time** completed	**Avg time** completed	**Avg time** completed	**Avg time** completed	**Selection** %P	%A	%N
sgb-book 23/26	537,648.5 23	437,689.8 22	741,300.6 22	658,572.3 23	**436,777.0** 22	82%	18%	0%
sgb-games 4/4	153,403.6 4	16,336.3 4	1,841,056.3 3	20,705.7 4	**15,561.5** 4	69%	31%	0%
haystacks 7/51	5,143,146.1 2	393,391.0 7	2,009,199.5 6	2,176,003.9 5	**363,636.2** 7	73%	27%	0%
pigeons 13/24	261,160.4 13	**207,801.8** 13	424,787.9 13	322,454.2 13	209,119.5 13	90%	10%	0%
QCP-20 11/15	4,181,471.6 5	1,384,722.1 11	2,082,329.8 10	**1,016,116.7** 11	1,116,712.2 11	41%	53%	6%
QCP-25 4/15	5,403,980.1 1	**837,841.5** 4	1,259,769.9 4	2,281,713.4 3	2,252,573.3 3	100%	0%	0%
QWH-20 10/10	947,171.9 9	476,163.4 10	342,075.3 10	**295,832.8** 10	404,047.6 10	12%	84%	4%
QWH-25 2/10	7,200,000.0 -	7,200,000.0 -	4,976,835.6 1	**2,567,511.8** 2	7,200,000.0 -	-	-	-
rand-2-23 10/10	435,346.4 10	447,983.2 10	1,723,637.9 10	1,546,868.7 10	**377,903.1** 10	93%	0%	7%
rand-2-24 10/10	962,257.9 10	806,797.6 10	4,949,429.3 6	3,740,633.4 9	**711,405.7** 10	90%	0%	10%
rlfapScens11 8/12	3,060,212.7 5	**113,393.7** 8	4,517,665.8 3	2,726,426.6 5	993,615.6 7	94%	4%	2%

6.3.4 Discussion

Our results show that, on difficult benchmarks where high-level consistency is needed, cluster-based minimality is warranted and solves problems otherwise unsolvable by STR2. Our experiments show that using a portfolio to select between AllSol, PerTuple, and neither is beneficial on many of these difficult benchmarks and outperforms all other four tested algorithms. We see that Cl-PerTuple most often outperforms Cl-AllSol than otherwise. Consequently, the portfolio selects PerTuple more often than it selects AllSol. Perhaps more important than the decision to select between PerTuple or AllSol is the decision of when to run neither. Indeed, Cl-Portfolio is able to beat, on several benchmarks, each of STR2, Cl-PerTuple, and Cl-AllSol by advantageously selecting 'Neither'.

We note that the random classifier has surprisingly good results. We attribute this strong performance to the use of the one-second cluster timeout: The detrimental effect of a bad selection by the random classifier is minimized by halting execution after one second. While initially introduced to mitigate poor performance, in fact, this fixed timeout may, in fact, constitute a limitation of our approach. In future work, one may want to investigate how to adaptively adjust the value of this timeout as a function of the amount of filtering being achieved.

Summary

In this chapter, we introduced a portfolio strategy that operates dynamically, during search, and locally, at the level of the clusters. We use such a portfolio to select between executing PerTuple or AllSol or skipping the given cluster. We described our procedure for training the classifier used in our portfolio and empirically evaluated our approach on benchmark problems.

Chapter 7

Conclusions and Future Work

We conclude the thesis and highlight possible directions for future work.

7.1 Conclusions

Minimality is a powerful consistency property that, when enforced during search, can yield large amounts of filtering, reduce thrashing, and allow us to solve difficult problems. Such filtering power often comes at a substantial cost and, consequently, minimality is rarely used in practice. Previous work demonstrated the promise of minimality for lookahead when restricted to local subproblems as clusters of a tree decomposition. Our research further explored this idea. We introduced three fundamentally new improvements to the AllSol and PerTuple algorithms:

1. Unmarked-first ordering heuristic to avoid redundant marking
2. Dangle identification to dynamically identify and remove tractable tree-substructures from the problem

3. MaxDeg heuristic for minimal dual graph construction, which greatly improves dangle identification performance.

These novel improvements resulted in an order of magnitude speed improvement for both AllSol and PerTuple.

We addressed an obstacle that affects many HLC algorithm, including minimality: the lack of a coherent weight-update strategy for the dom/wdeg ordering heuristic. We proposed a framework for expressing weight-update strategies defined by three orthogonal parameters. While our proposed parameter settings did not improve performance, we hope that our framework can guide the design of alternative strategies that effectively update the weights used in an ordering heuristic, for minimality, other HLCs, and beyond.

We introduced an algorithm portfolio that operates dynamically, during search, and locally, at the cluster level. When used for lookahead, it chooses, for each cluster, whether to apply PerTuple or AllSol or to bypass the cluster. We empirically established that this portfolio outperforms other alternatives, including STR2, on difficult benchmarks.

7.2 Future Work

The research presented in this thesis opens up many avenues for further investigations:

1. The improvements made to the AllSol and PerTuple algorithms have only been evaluated in the context of constraint minimality. While the use of the minimal dual graph is specific to operating on the dual CSP, both dangle identification and the unmarked-first ordering heuristic could be applied to domain minimality (i.e., inverse consistency including global inverse consistency).

2. Dangle identification can advantageously be exploited during the standard backtrack-search procedure. Identifying dangling tree structures is a cheap operation and would save search effort by steering away search from such non-problems.

3. The study of weight-update strategies for high-level consistencies remains an open question and deserves further attention, whether by expanding the framework proposed here or exploring new directions. Algorithms for enforcing lightweight consistency properties such as GAC are successful, in part, because their efforts are guided by a flexible and responsive ordering heuristic such as dom/wdeg. HLC seems to miss on the benefits of dom/wdeg because the algorithms lack a sensible weight-update strategy.

4. The cluster-level portfolio may be improved in numerous ways. A more sophisticated classifier may yield more accurate classification (e.g., graph convolutional networks for identifying important structural features). A classifier that predicts the expected filtering would call costly consistency algorithms only when they are most beneficial. Additionally, the one-second runtime limit could be dynamically adjusted to allow active filtering to continue.

5. AllSol and PerTuple may be advantageously combined into a hybrid algorithm. AllSol could be used as the primary minimality algorithm with PerTuple used selectively to attempt to mark or remove lingering unmarked tuples.

6. Stampede, the solver on which we have implemented all algorithms discussed in this thesis, supports generating visualization of the search and lookahead procedures. It would be both useful and informative to develop visualizations of our dangle identification and cluster-minimality algorithms.

7. On some benchmarks, cluster minimality is often too costly to execute at every variable instantiation in search. We believe that a strategy, such as the one currently being developed by Woodward et al., [1] which reactively calls a high-level consistency only when it is most needed and as long as it is beneficial, would allow us to exploit the filtering power of cluster minimality and of our portfolio on every instance of any benchmark.

[1] publication forthcoming

Appendix A

Results of Experiments in Section 6.3

This appendix contains the detailed results of the experiments discussed in Section 6.3. These experiments are carried over from 195 different benchmark problems with a total of 5,038 instances.

In Sections 6.3.2 and 6.3.3, we report only benchmarks that cannot advantageously be solved by GAC but require high-level consistency. We exclude the benchmarks that can be solved with GAC. The excluded benchmarks are as follows:

The 103 benchmarks best solved by STR2 and dom/deg are: aim-50, allIntervalSeries, bddLarge, bddSmall, bqwh-15-106_glb, bqwh-15-106, bqwh-18-141_glb, chessboardColoration, coloring, dag-half, dag-rand, domino, driver, fapp01, frb30-15, frb35-17, frb40-19, frb45-21, frb50-23, geom, golombRulerArity3, hos, k-insertion, leighton-25, leighton-5, myciel, register-fpsol, register-inithx, register-mulsol, register-zeroin, school, sgb-miles, hanoi, jnhSat, jnhUnsat, jobShop-e0ddr1, jobShop-e0ddr2, jobShop-enddr2, jobShop-ewddr2, knights, langford, langford2, langford3, langford4, lexVg, marc, nengfa, ogdVg, os-taillard-4, os-

taillard-7, pret, pseudo-garden, pseudo-jnh, pseudo-par, pseudo-primesDimacs, pseudo-radar, pseudo-ssa, pseudo-uclid, queens, QWH-10, QWH-15, ramsey3, ramsey4, rand-10-20-10, rand-2-25, rand-2-26, rand-2-27, rand-2-30-15-fcd, rand-2-30-15, rand-2-40-19-fcd, rand-2-40-19, rand-2-50-23-fcd, rand-2-50-23, rand-3-20-20-fcd, rand-3-20-20, rand-3-24-24-fcd, rand-3-24-24, rand-3-28-28-fcd, rand-3-28-28, rand-8-20-5, renault, rlfapGraphs, rlfapScens, schurrLemma, ssa, subs, super-jobShop-e0ddr1, super-jobShop-ewddr2, super-os-taillard-5, super-queens, tightness0.1, tightness0.2, tightness0.35, tightness0.5, tightness0.65, tightness0.8, tightness0.9, travellingSalesman-20, travellingSalesman-25, ukPuzzle, ukVg, varDimacs, wordsVg.

The 129 benchmarks best solved by STR2 and dom/wdeg are: aim-100, aim-200, aim-50, allIntervalSeries, bddLarge, bddSmall, bqwh-15-106_glb, bqwh-15-106, bqwh-18-141_glb, bqwh-18-141, chessboardColoration, coloring, composed-25-1-80, composed-25-10-20, composed-75-1-80, dag-half, dag-rand, domino, driver, dubois, ehi-85, ehi-90, fapp01, frb30-15, frb35-17, frb40-19, frb45-21, frb50-23, geom, golombRulerArity3, golombRulerArity4, graceful, hos, full-insertion, k-insertion, leighton-25, leighton-5, myciel, register-fpsol, register-inithx, register-mulsol, register-zeroin, school, sgb-miles, sgb-queen, hanoi, jnhSat, jnhUnsat, jobShop-e0ddr1, jobShop-e0ddr2, jobShop-enddr1, jobShop-enddr2, jobShop-ewddr2, knights, langford, langford2, langford3, langford4, lexVg, marc, modifiedRenault, nengfa, ogdVg, os-taillard-4, os-taillard-5, os-taillard-7, pret, pseudo-aim, pseudo-garden, pseudo-jnh, pseudo-par, pseudo-primesDimacs, pseudo-radar, pseudo-ssa, pseudo-uclid, QCP-10, QCP-15, queenAttacking, queens, queensKnights, QWH-10, QWH-15, ramsey3, ramsey4, rand-10-20-10, rand-2-25, rand-2-26, rand-2-27, rand-2-30-15-fcd, rand-2-30-15, rand-2-40-19-fcd,

rand-2-40-19, rand-2-50-23-fcd, rand-2-50-23, rand-3-20-20-fcd, rand-3-20-20, rand-3-24-24-fcd, rand-3-24-24, rand-3-28-28-fcd, rand-3-28-28, rand-8-20-5, renault, rlfapGraphs, rlfapGraphsMod, rlfapScens, rlfapScensMod, schurrLemma, ssa, subs, super-jobShop-e0ddr1, super-jobShop-enddr1, super-jobShop-enddr2, super-jobShop-ewddr2, super-os-taillard-4, super-os-taillard-5, super-queens, tightness0.1, tightness0.2, tightness0.35, tightness0.5, tightness0.65, tightness0.8, tightness0.9, travellingSalesman-20, travellingSalesman-25, ukPuzzle, ukVg, varDimacs, wordsVg.

Table A.1 reports a summary of the results of the five algorithms and two ordering heuristics over the 195 benchmarks tested.

Table A.1: Results summary for STR2 and cluster-minimality algorithms

	Algorithm	**Complete**	**Timeout**	**Memout**	**Average time (ms)**		**Avg NV**
					by one	**by all**	**by all**
dom/deg	STR2	**3,059**	**1,379**	**600**	**892,092.1**	**123,222.6**	1,117,820.8
	Cl-PerTuple	2,623	1,523	892	2,037,417.5	487,988.3	**68,905.3**
	Cl-AllSol	2,629	1,536	873	2,087,294.5	509,396.7	68,969.4
	Cl-Portfolio	2,631	1,636	771	2,082,403.0	566,410.7	69,991.6
	Cl-Random	2,615	1,446	977	2,068,741.9	458,112.4	128,045.6
dom/wdeg	STR2	**3,311**	**1,127**	**600**	**303,418.6**	**43,379.7**	398,969.1
	Cl-PerTuple	2,676	1,467	895	1,929,379.6	485,475.7	**67,779.9**
	Cl-AllSol	2,696	1,467	875	1,964,438.7	507,647.0	67,834.6
	Cl-Portfolio	2,690	1,576	772	1,955,051.3	549,490.9	68,768.1
	Cl-Random	2,687	1,379	972	1,937,688.5	464,299.9	147,659.9

The first column indicates the ordering heuristic followed by the algorithm. For each ordering and algorithm, we report:

1. The number of instances completed
2. The number of instances terminated with a timeout
3. The number of instances terminated with a memout
4. The CPU time (in milliseconds) averaged over instances completed by at least one algorithm. In the case that an algorithm does not complete an instance, the CPU time is treated as the time limit (i.e., 7,200,000 ms)
5. The CPU time (in milliseconds) averaged over instances completed by all algorithms. Thus, no timeouts/memouts are included in this average
6. The average number (averaged over instances completed by all algorithms) of nodes visited by the search

In each of the last six columns, we typeset the best entry for each ordering heuristic in boldface.

Unsurprisingly, STR2 substantially benefits from the dom/wdeg ordering heuristic whereas the cluster-minimality algorithms see consistent but relatively small improvement. Also, all cluster-minimality approaches result in a significantly larger number of memouts. This fact is due to the high-memory requirements of the data structures needed for AllSol and PerTuple as well as for weight updates within the search executed by these two algorithms.

Among the four minimality algorithms, Cl-Portfolio results in the fewest number of memouts because our implementation delays building the costly data structures until they are needed. Under dom/deg, Cl-Portfolio solves slight more instances than the other three algorithms. Under dom/wdeg, Cl-AllSol outperforms the other three.

Table A.2 reports results using the dom/deg ordering heuristic. Table A.3 reports results using dom/wdeg. In both tables, the first column indicates the benchmark, the number of instances included, and the total number of instances. The next five columns report the results for a given algorithm: both the runtime (in milliseconds) averaged over instances completed by at least one algorithm and the number of instances completed by the given algorithm. In the case that an algorithm does not complete an instance, the CPU time is treated as the time limit (i.e., 7,200,000 ms). Finally, the last three columns report how often the portfolio selected 'PerTuple', 'AllSol', or 'Neither', respectively.

Table A.2: STR2 and cluster-minimality algorithms using dom/deg

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
aim-100	2,289,850.8	1,280,598.0	1,203,669.9	1,992,658.4	1,258,765.3	9%	91%	0%
24/24	18	20	21	18	20			
aim-200	5,268,448.1	5,117,181.1	4,886,498.9	5,793,812.7	5,167,170.4	18%	74%	8%
24/24	8	7	8	5	7			
aim-50	308.4	502.4	490.6	754.1	517.8	3%	97%	0%
24/24	24	24	24	24	24			
allIntervalSeries	193,843.1	1,664,904.8	1,335,925.8	1,654,643.4	1,513,718.4	43%	55%	2%
18/25	18	14	15	14	15			
bddLarge	159,319.5	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
35/35	35	-	-	-	-			
bddSmall	58,656.0	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
35/35	35	-	-	-	-			
BH-4-13	-	-	-	-	-	-	-	-
0/7	-	-	-	-	-			
BH-4-4	7,200,000.0	3,808.2	3,500,493.7	3,869.8	3,560.9	28%	72%	0%
10/10	-	10	10	10	10			
BH-4-7	-	-	-	-	-	-	-	-
0/20	-	-	-	-	-			
bqwh-15-106_glb	24.0	447.6	482.7	418.9	496.9	3%	97%	0%
100/100	100	100	100	100	100			
bqwh-15-106	1,613.8	6,092.7	25,180.1	15,292.3	7,300.0	13%	87%	0%
100/100	100	100	100	100	100			

Table A.2: STR2 and cluster-minimality algorithms using dom/deg (continued)

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark # instances	**Avg time** completed	**Avg time** completed	**Avg time** completed	**Avg time** completed	**Avg time** completed	**Selection** %P	%A	%N
bqwh-18-141_glb 100/100	44.4 100	2,410.5 100	3,606.3 100	2,467.0 100	3,600.8 100	2%	98%	0%
bqwh-18-141 100/100	97,353.0 100	50,069.5 100	170,490.2 100	105,331.7 100	51,660.6 100	25%	75%	0%
chessboardColoration 14/19	2,344,170.4 10	2,621,827.0 9	2,652,924.1 9	2,658,117.8 9	3,678,644.0 7	70%	6%	25%
coloring 22/22	5,997.5 22	298,088.6 22	228,477.1 22	235,394.5 22	188,981.4 22	57%	42%	1%
composed-25-1-2 10/10	7,200,000.0 -	197.0 10	202.0 10	121.3 10	121.1 10	90%	10%	0%
composed-25-1-25 10/10	7,200,000.0 -	250.9 10	258.2 10	1,402.5 10	143.5 10	90%	10%	0%
composed-25-1-40 10/10	7,200,000.0 -	280.7 10	298.5 10	1,332.1 10	164.3 10	87%	13%	0%
composed-25-1-80 10/10	4,140,176.6 5	432.8 10	836.8 10	898.8 10	289.0 10	100%	0%	0%
composed-25-10-20 10/10	3,244,168.3 6	58,576.1 10	148,372.1 10	79,947.7 10	60,692.4 10	34%	66%	0%
composed-75-1-2 10/10	7,200,000.0 -	712.7 10	733.3 10	3,049.3 10	415.4 10	88%	12%	0%
composed-75-1-25 10/10	7,200,000.0 -	760.6 10	782.9 10	3,405.6 10	440.8 10	92%	8%	0%

Table A.2: STR2 and cluster-minimality algorithms using dom/deg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
composed-75-1-40	7,200,000.0	812.9	838.7	1,504.0	463.0	94%	6%	0%
10/10	-	10	10	10	10			
composed-75-1-80	5,040,063.1	1,413.2	2,945.8	1,949.8	1,057.8	96%	4%	0%
10/10	3	10	10	10	10			
cril	5,252,506.9	3,108,408.4	4,115,732.2	3,106,661.9	3,107,536.9	68%	32%	0%
7/8	2	4	3	4	4			
dag-half	2,340,958.9	6,283,935.0	5,978,589.0	6,275,038.8	6,270,043.9	96%	4%	1%
14/25	14	2	3	2	2			
dag-rand	84,898.6	948,651.7	983,687.3	958,896.8	994,743.0	0%	67%	33%
25/25	25	22	22	22	22			
domino	289,014.7	293,294.6	293,256.4	293,557.1	293,631.2	0%	100%	0%
22/23	22	22	22	22	22			
driver	80,075.0	3,466,730.4	3,524,246.7	4,261,936.3	3,468,135.8	40%	53%	7%
7/7	7	4	4	3	4			
dubois	1,580,785.7	1,593,133.9	1,573,624.5	2,462,762.2	1,965,018.6	0%	100%	0%
7/13	7	7	7	6	6			
ehi-85	446,748.4	29,337.8	41,181.1	35,871.3	22,187.2	81%	0%	19%
100/100	100	100	100	100	100			
ehi-90	976,702.8	31,621.1	45,320.0	40,051.3	23,717.2	80%	0%	20%
100/100	95	100	100	100	100			
fapp01	5,075,865.9	6,297,561.9	6,047,183.6	5,998,916.8	6,208,830.1	32%	66%	2%
11/11	10	5	9	8	7			

Table A.2: STR2 and cluster-minimality algorithms using dom/deg (continued)

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
fapp02	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp03	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp04	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp05	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp06	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp07	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp08	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp09	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp10	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp11	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp12	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			

Table A.2: STR2 and cluster-minimality algorithms using dom/deg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
fapp13	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp14	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp15	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp16	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp17	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp18	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp19	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp20	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp21	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp23	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp24	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			

Table A.2: STR2 and cluster-minimality algorithms using dom/deg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
fapp25	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp26	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp27	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp31	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp33	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp36	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp39	-	-	-	-	-	-	-	-
0/10	-	-	-	-	-			
frb30-15	4,002.2	205,937.1	246,811.5	267,993.5	204,662.7	84%	16%	0%
10/10	10	10	10	10	10			
frb35-17	49,829.7	2,713,569.8	2,836,334.5	2,804,302.6	2,752,658.9	89%	11%	0%
10/10	10	8	10	8	8			
frb40-19	384,871.0	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
10/10	10	-	-	-	-			
frb45-21	3,983,057.6	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
10/10	8	-	-	-	-			

Table A.2: STR2 and cluster-minimality algorithms using dom/deg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
frb50-23	1,200,655.9	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
2/10	2	-	-	-	-			
frb53-24	-	-	-	-	-	-	-	-
0/10	-	-	-	-	-			
frb56-25	-	-	-	-	-	-	-	-
0/10	-	-	-	-	-			
frb59-26	-	-	-	-	-	-	-	-
0/10	-	-	-	-	-			
geom	93,553.4	809,000.8	1,054,461.5	952,894.9	805,351.7	66%	34%	0%
100/100	100	93	89	90	93			
golombRulerArity3	2,648,761.6	3,298,311.1	3,694,182.7	3,525,433.3	3,561,831.0	71%	11%	18%
9/14	6	5	5	5	5			
golombRulerArity4	249,900.7	249,418.1	249,401.3	249,376.0	249,377.0	63%	38%	0%
2/14	2	2	2	2	2			
graceful	2,811,900.0	2,537,964.9	2,612,469.2	2,534,769.2	2,551,260.6	57%	43%	0%
3/4	2	2	2	2	2			
hos	3,050,276.2	3,192,506.4	3,549,763.5	3,319,302.6	3,185,207.9	51%	48%	0%
12/14	7	7	7	7	7			
full-insertion	2,488,182.9	2,604,259.8	2,632,382.1	3,152,287.3	2,410,508.3	30%	70%	0%
32/41	21	21	21	19	22			
k-insertion	527,444.6	1,184,167.4	1,364,132.0	1,185,267.8	1,149,488.0	5%	95%	0%
17/33	16	15	15	15	15			

Table A.2: STR2 and cluster-minimality algorithms using dom/deg (continued)

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
leighton-15	3,278,469.2	2,064,039.7	3,357,295.0	2,465,225.9	1,150,907.7	70%	7%	23%
9/26	5	7	5	6	8			
leighton-25	18,185.2	2,580,024.7	2,580,463.2	2,454,850.5	2,454,817.9	96%	4%	0%
6/31	6	4	4	4	4			
leighton-5	7,913.8	1,972,143.0	720,398.9	2,062,703.4	1,892,885.5	85%	10%	5%
8/8	8	6	8	6	6			
mug	3,600,012.2	19,480.1	19,059.5	346,696.4	24,004.2	11%	89%	0%
8/8	4	8	8	8	8			
myciel	268,654.8	1,696,739.7	1,440,433.3	1,417,127.9	1,415,336.1	31%	68%	1%
13/16	13	10	11	11	11			
register-fpsol	43,380.7	3,638,391.6	3,638,802.7	3,614,225.3	3,614,119.2	67%	0%	33%
6/37	6	3	3	3	3			
register-inithx	62,083.5	7,200,000.0	7,200,000.0	3,630,203.2	3,630,224.2	100%	0%	0%
6/32	6	-	-	3	3			
register-mulsol	9,896.3	3,208,585.0	2,351,666.8	1,637,574.2	1,465,719.6	68%	5%	27%
9/49	9	5	7	8	8			
register-zeroin	9,397.9	3,609,961.7	3,610,109.0	3,602,821.1	3,602,754.9	75%	0%	25%
6/31	6	3	3	3	3			
school	28,565.9	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
3/8	3	-	-	-	-			
sgb-book	1,672,291.7	436,830.9	726,704.7	1,340,712.8	435,278.1	83%	17%	0%
23/26	19	22	22	19	22			

Table A.2: STR2 and cluster-minimality algorithms using dom/deg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
sgb-games	3,604,888.0	16,446.9	1,840,626.6	21,023.2	15,562.5	69%	31%	0%
4/4	2	4	3	4	4			
sgb-miles	666,341.5	1,820,655.1	1,886,009.0	2,123,445.9	1,199,251.6	72%	14%	14%
11/42	10	9	9	8	10			
sgb-queen	2,894,074.8	1,996,657.0	1,954,690.3	1,798,709.2	1,982,455.7	68%	14%	18%
15/50	9	11	12	12	11			
hanoi	484.5	1,320.3	1,329.0	1,331.2	1,332.2	0%	100%	0%
5/5	5	5	5	5	5			
haystacks	3,215,129.3	710,220.7	2,057,686.4	1,715,194.3	750,043.9	70%	30%	0%
7/51	4	7	5	6	7			
jnhSat	5,876.8	494,488.9	544,997.1	518,512.0	1,310,073.7	22%	6%	73%
16/16	16	16	16	16	14			
jnhUnsat	2,615.8	307,549.7	307,827.2	325,472.8	656,495.8	10%	0%	90%
34/34	34	34	34	34	33			
jobShop-e0ddr1	1,446,656.6	2,783,994.4	1,864,011.8	2,053,703.7	2,363,417.3	35%	63%	3%
5/10	4	4	4	4	4			
jobShop-e0ddr2	11,150.1	2,746,535.9	637,888.8	1,092,855.2	975,372.5	34%	62%	4%
4/10	4	4	4	4	4			
jobShop-enddr1	808,161.5	2,025,407.3	579,259.0	1,634,787.8	1,091,202.1	36%	61%	3%
9/10	8	9	9	8	9			
jobShop-enddr2	12,367.4	2,904,340.5	572,973.4	1,202,031.3	791,105.0	32%	64%	4%
3/6	3	3	3	3	3			

Table A.2: STR2 and cluster-minimality algorithms using dom/deg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
jobShop-ewddr2	13,039.5	2,982,751.7	620,478.1	1,283,249.2	1,006,527.9	30%	65%	4%
10/10	10	10	10	10	10			
js-taillard-15	-	-	-	-	-	-	-	-
0/30	-	-	-	-	-			
knights	217,974.4	345,568.3	326,801.2	344,146.0	327,687.6	20%	40%	40%
10/19	10	10	10	10	10			
langford	23,110.9	583,670.8	211,366.1	379,465.4	553,630.7	61%	25%	14%
4/4	4	4	4	4	4			
langford2	5,876.9	46,581.6	48,229.3	47,199.5	42,108.0	46%	31%	23%
16/24	16	16	16	16	16			
langford3	174,421.0	2,476,237.0	1,813,865.4	2,105,588.0	2,285,935.5	85%	0%	15%
15/23	15	10	12	12	11			
langford4	142,819.5	1,511,658.9	1,409,684.2	1,465,053.7	1,478,167.2	42%	0%	58%
13/24	13	11	11	11	11			
lexVg	25,736.4	1,149,903.2	1,167,100.9	1,070,635.8	1,153,293.0	1%	99%	0%
63/63	63	57	56	57	56			
marc	47,936.2	187,704.5	191,228.2	187,725.1	185,283.3	0%	50%	50%
10/10	10	10	10	10	10			
modifiedRenault	248,697.8	39,287.4	36,821.1	224,612.7	37,430.9	58%	42%	0%
50/50	49	50	50	49	50			
nengfa	2,975,942.6	4,445,915.8	4,460,109.6	4,441,261.6	4,445,499.5	34%	60%	5%
5/10	3	2	2	2	2			

Table A.2: STR2 and cluster-minimality algorithms using dom/deg (continued)

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
ogdVg	600,164.9	2,251,092.4	2,069,489.7	2,225,340.1	1,901,566.9	12%	88%	0%
43/65	42	32	32	31	33			
os-gp	-	-	-	-	-	-	-	-
0/19	-	-	-	-	-			
os-taillard-10	-	-	-	-	-	-	-	-
0/30	-	-	-	-	-			
os-taillard-15	-	-	-	-	-	-	-	-
0/30	-	-	-	-	-			
os-taillard-4	518,870.4	844,506.8	1,121,046.1	1,019,376.7	888,456.7	27%	70%	3%
30/30	29	29	27	28	27			
os-taillard-5	3,715,048.7	5,003,526.2	4,043,962.7	4,819,209.8	3,444,165.4	38%	39%	23%
19/30	10	8	10	8	12			
os-taillard-7	6,202,651.4	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
7/30	1	-	-	-	-			
pigeons	256,218.6	193,237.3	422,270.0	313,310.9	202,913.2	81%	19%	0%
13/24	13	13	13	13	13			
pret	95,181.5	176,557.2	175,124.3	459,921.1	234,400.3	0%	100%	0%
4/8	4	4	4	4	4			
pseudo-aim	1,952,036.6	1,704,745.6	1,712,151.9	2,350,053.5	1,703,246.6	9%	91%	0%
48/48	39	37	37	33	37			
pseudo-garden	18.3	637.6	900.9	651.0	844.4	15%	85%	0%
6/7	6	6	6	6	6			

Table A.2: STR2 and cluster-minimality algorithms using dom/deg (continued)

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark # instances	**Avg time** completed	**Avg time** completed	**Avg time** completed	**Avg time** completed	**Avg time** completed	**Selection** %P	%A	%N
pseudo-jnh 16/16	13,440.2 16	3,031,271.8 11	5,060,113.1 7	3,173,600.5 11	3,513,946.8 10	59%	7%	34%
pseudo-par 20/30	36,394.8 20	1,622,947.0 17	1,584,531.9 17	2,144,914.6 15	1,616,480.8 17	0%	100%	0%
pseudo-primesDimacs 3/11	4,933,882.5 1	7,200,000.0 -	7,200,000.0 -	7,200,000.0 -	7,200,000.0 -	-	-	-
pseudo-radar 6/12	13,600.1 6	3,777,221.9 3	3,640,546.5 3	3,699,008.1 3	3,640,381.1 3	0%	100%	0%
pseudo-ssa 7/8	3,337,234.1 4	4,310,300.7 3	4,311,406.8 3	5,148,880.0 2	4,368,284.1 3	0%	100%	0%
pseudo-uclid 9/36	4,974,473.5 3	4,976,152.8 3	4,975,768.3 3	4,976,294.4 3	4,975,803.4 3	0%	100%	0%
QCP-10 15/15	178,137.2 15	58,158.1 15	87,559.5 15	56,489.4 15	62,848.9 15	17%	83%	0%
QCP-15 15/15	3,842,598.7 8	128,626.1 15	132,850.9 15	98,666.9 15	115,090.6 15	24%	71%	5%
QCP-20 11/15	7,200,000.0 -	2,273,913.0 9	2,351,278.5 9	1,138,250.2 11	2,158,952.1 9	42%	52%	6%
QCP-25 4/15	7,200,000.0 -	838,179.4 4	1,269,032.8 4	2,274,623.1 3	2,254,291.6 3	100%	0%	0%
queenAttacking 4/10	1,829,407.8 3	360,437.5 4	582,309.6 4	501,667.5 4	1,803,037.9 4	74%	14%	12%

Table A.2: STR2 and cluster-minimality algorithms using dom/deg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
queens	25,551.1	780,536.3	774,079.8	780,098.7	790,150.2	49%	30%	22%
10/14	10	9	9	9	9			
queensKnights	1,543,653.4	1,518,278.8	1,743,128.7	1,654,649.1	1,517,909.0	67%	28%	5%
10/18	8	8	8	8	8			
QWH-10	214.7	8,115.8	9,188.2	7,803.3	7,932.2	9%	91%	0%
10/10	10	10	10	10	10			
QWH-15	16,322.7	69,833.0	46,425.8	46,843.8	60,288.5	6%	92%	1%
10/10	10	10	10	10	10			
QWH-20	6,564,135.3	492,676.5	340,291.2	298,351.1	447,840.4	11%	85%	4%
10/10	1	10	10	10	10			
QWH-25	7,200,000.0	7,200,000.0	5,130,443.6	4,774,416.0	7,200,000.0	-	-	-
2/10	-	-	1	1	-			
radar-8-30-3-0	-	-	-	-	-	-	-	-
0/50	-	-	-	-	-			
radar-9-28-4-2	-	-	-	-	-	-	-	-
0/50	-	-	-	-	-			
ramsey3	36.7	42,727.8	44,363.1	41,932.7	41,753.1	48%	52%	0%
2/8	2	2	2	2	2			
ramsey4	1,022.5	717,853.6	803,870.2	638,556.1	331,441.7	43%	7%	50%
1/8	1	1	1	1	1			
rand-10-20-10	1,048.0	2,572.9	2,559.0	2,559.1	2,558.8	0%	100%	0%
20/20	20	20	20	20	20			

Table A.2: STR2 and cluster-minimality algorithms using dom/deg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
rand-2-23	432,574.9	446,742.5	1,713,566.1	1,492,744.4	379,067.9	93%	0%	7%
10/10	10	10	10	10	10			
rand-2-24	956,969.7	859,555.3	5,101,956.2	3,689,564.9	720,362.5	90%	0%	10%
10/10	10	10	5	9	10			
rand-2-25	1,938,422.0	2,841,952.8	5,701,803.2	5,486,425.0	2,490,051.0	87%	0%	13%
10/10	10	8	3	4	10			
rand-2-26	3,400,379.9	4,014,201.3	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
4/10	4	3	-	-	-			
rand-2-27	4,707,973.5	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
3/10	3	-	-	-	-			
rand-2-30-15-fcd	5,092.2	323,763.6	319,184.2	328,990.1	329,547.4	81%	19%	0%
50/50	50	50	50	50	50			
rand-2-30-15	9,286.0	481,840.6	522,216.3	496,074.4	489,957.3	86%	14%	0%
50/50	50	50	50	50	50			
rand-2-40-19-fcd	621,999.7	6,673,366.0	6,754,612.9	6,700,324.3	6,741,012.7	87%	13%	0%
50/50	50	5	5	5	4			
rand-2-40-19	1,261,624.7	7,120,509.7	7,127,045.1	7,126,287.6	7,120,259.5	90%	10%	0%
50/50	50	1	1	1	1			
rand-2-50-23-fcd	5,023,877.8	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
8/50	5	-	-	-	-			
rand-2-50-23	5,355,773.9	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
3/50	2	-	-	-	-			

Table A.2: STR2 and cluster-minimality algorithms using dom/deg (continued)

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
rand-3-20-20-fcd	398,264.0	2,546,831.1	1,270,012.7	1,813,836.5	2,272,902.5	58%	42%	0%
50/50	50	43	49	47	44			
rand-3-20-20	771,296.8	3,832,339.0	2,251,962.2	3,113,540.5	3,532,630.5	55%	45%	0%
50/50	49	35	47	41	37			
rand-3-24-24-fcd	3,167,626.0	6,735,718.7	6,341,867.9	6,418,268.2	6,568,359.2	71%	29%	0%
22/50	20	2	4	4	4			
rand-3-24-24	4,242,602.0	7,200,000.0	6,711,726.1	7,020,341.3	7,200,000.0	-	-	-
10/50	9	-	2	1	-			
rand-3-28-28-fcd	1,623,802.1	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
2/50	2	-	-	-	-			
rand-3-28-28	1,243,947.5	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
1/50	1	-	-	-	-			
rand-8-20-5	106,144.3	460,707.5	433,656.6	534,658.7	308,758.8	40%	54%	6%
20/20	20	20	20	20	20			
renault	962.4	35,296.2	19,433.3	32,329.4	21,900.2	46%	54%	0%
2/2	2	2	2	2	2			
rlfapGraphs	522,872.3	1,791,870.3	1,612,793.4	1,715,885.0	1,788,760.0	24%	76%	0%
14/14	13	13	14	12	13			
rlfapGraphsMod	4,201,029.1	462,767.4	945,847.3	524,672.7	449,657.1	46%	54%	0%
12/12	5	12	11	12	12			
rlfapScens11	7,200,000.0	969,702.2	4,517,589.5	2,726,426.3	999,830.0	94%	4%	2%
8/12	-	7	3	5	7			

Table A.2: STR2 and cluster-minimality algorithms using dom/deg (continued)

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
rlfapScens	29,552.1	489,193.9	425,673.2	329,036.9	501,243.8	30%	69%	1%
11/11	11	11	11	11	11			
rlfapScensMod	2,772,517.2	251,109.3	770,869.3	198,637.2	255,753.7	54%	46%	0%
13/13	8	13	12	13	13			
schurrLemma	275,604.5	1,991,058.5	2,082,901.8	2,268,356.2	2,416,063.5	81%	5%	14%
9/10	9	7	7	7	7			
ssa	1,029,518.4	1,032,751.8	1,032,727.3	1,034,173.1	1,034,356.6	6%	94%	0%
7/8	6	6	6	6	6			
subs	1,419.8	3,012.6	3,234.8	2,861.4	2,877.3	52%	15%	33%
9/9	9	9	9	9	9			
super-jobShop-e0ddr1	2,418,659.3	6,117,837.1	3,679,554.2	7,200,000.0	3,685,068.2	75%	16%	9%
3/10	2	1	2	-	2			
super-jobShop-e0ddr2	-	-	-	-	-	-	-	-
0/10	-	-	-	-	-			
super-jobShop-enddr1	4,810,078.1	5,252,276.5	3,720,746.0	7,200,000.0	3,988,823.9	76%	17%	7%
3/10	1	2	2	-	2			
super-jobShop-enddr2	3,619,861.2	7,200,000.0	4,704,472.8	7,200,000.0	3,237,503.5	74%	18%	8%
2/6	1	-	1	-	2			
super-jobShop-ewddr2	43,841.9	7,200,000.0	5,484,080.6	7,200,000.0	2,361,430.1	71%	22%	8%
6/9	6	-	2	-	6			
super-js-taillard-15	-	-	-	-	-	-	-	-
0/25	-	-	-	-	-			

Table A.2: STR2 and cluster-minimality algorithms using dom/deg (continued)

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
super-os-taillard-10	-	-	-	-	-	-	-	-
0/30	-	-	-	-	-			
super-os-taillard-15	-	-	-	-	-	-	-	-
0/30	-	-	-	-	-			
super-os-taillard-4	1,623,196.1	2,907,544.3	1,286,475.3	2,028,325.5	1,202,340.0	36%	36%	27%
28/30	22	21	26	23	27			
super-os-taillard-5	3,450,568.6	7,200,000.0	5,651,985.3	6,035,238.4	3,531,152.5	48%	20%	32%
8/28	5	-	2	3	7			
super-os-taillard-7	-	-	-	-	-	-	-	-
0/30	-	-	-	-	-			
super-queens	530,363.1	1,442,133.0	1,479,454.8	1,444,328.8	1,442,571.2	100%	0%	0%
5/14	5	4	4	4	4			
tightness0.1	76,313.7	655,965.5	1,124,358.3	800,347.1	4,324,708.2	14%	0%	86%
100/100	100	100	100	100	87			
tightness0.2	84,550.6	4,893,936.8	5,737,126.2	5,125,682.9	5,003,628.5	95%	5%	0%
100/100	100	46	39	48	43			
tightness0.35	76,661.5	5,361,446.1	5,360,711.4	5,474,983.9	5,441,963.9	64%	36%	0%
100/100	100	42	51	42	39			
tightness0.5	128,389.3	5,576,751.0	5,938,348.6	5,814,471.1	5,732,698.7	42%	58%	0%
100/100	100	40	36	36	37			
tightness0.65	126,396.1	3,037,537.2	3,463,681.1	2,986,661.7	3,162,833.6	35%	65%	0%
100/100	100	84	77	84	81			

Table A.2: STR2 and cluster-minimality algorithms using dom/deg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
tightness0.8	199,408.2	1,390,618.2	1,560,327.1	1,257,908.2	1,380,876.6	40%	60%	0%
100/100	100	97	94	97	97			
tightness0.9	277,714.9	820,339.6	896,032.2	791,656.8	813,590.9	41%	59%	0%
100/100	99	97	95	98	96			
travellingSalesman-20	34,316.3	1,949,023.0	1,870,585.5	1,593,791.7	1,918,282.5	31%	69%	0%
15/15	15	12	13	13	12			
travellingSalesman-25	278,859.6	4,795,258.4	4,863,280.3	4,688,381.9	4,776,402.6	32%	68%	0%
15/15	15	6	6	6	6			
ukPuzzle	7.6	14.9	13.9	15.9	16.0	50%	50%	0%
1/22	1	1	1	1	1			
ukVg	211,285.4	1,579,971.4	1,684,006.8	1,517,735.0	1,443,391.3	13%	87%	0%
36/65	36	30	29	31	30			
varDimacs	624,308.2	2,002,799.2	1,767,887.4	1,812,056.2	1,770,251.0	4%	96%	0%
9/9	9	7	7	7	7			
wordsVg	77,854.7	1,653,529.7	1,517,753.6	1,519,073.1	1,505,476.1	2%	98%	0%
65/65	65	54	55	54	55			

Table A.3: STR2 and cluster-minimality algorithms using dom/wdeg

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
aim-100	147.7	4,114.2	10,211.6	5,879.5	3,978.9	8%	92%	0%
24/24	24	24	24	24	24			
aim-200	2,908.0	182,995.0	234,708.6	157,868.5	176,658.9	18%	79%	3%
24/24	24	24	24	24	24			
aim-50	25.9	426.5	434.0	546.9	460.1	3%	97%	0%
24/24	24	24	24	24	24			
allIntervalSeries	802,873.7	1,665,719.4	1,330,272.4	1,651,533.8	1,438,644.2	43%	55%	2%
18/25	16	14	15	14	15			
bddLarge	156,632.8	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
35/35	35	-	-	-	-			
bddSmall	59,776.7	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
35/35	35	-	-	-	-			
BH-4-13	-	-	-	-	-	-	-	-
0/7	-	-	-	-	-			
BH-4-4	70,069.1	3,804.9	3,921,972.9	3,898.0	3,537.8	28%	72%	0%
10/10	10	10	9	10	10			
BH-4-7	-	-	-	-	-	-	-	-
0/20	-	-	-	-	-			
bqwh-15-106_glb	24.4	451.6	486.3	419.0	500.6	3%	97%	0%
100/100	100	100	100	100	100			
bqwh-15-106	369.5	6,029.8	24,964.5	15,333.7	7,243.3	14%	86%	0%
100/100	100	100	100	100	100			

Table A.3: STR2 and cluster-minimality algorithms using dom/wdeg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
bqwh-18-141_glb	44.0	2,199.7	3,617.5	2,290.1	3,588.0	2%	98%	0%
100/100	100	100	100	100	100			
bqwh-18-141	7,788.7	43,271.3	162,971.5	100,633.9	45,436.8	25%	75%	0%
100/100	100	100	100	100	100			
chessboardColoration	168,289.4	2,400,982.0	2,405,273.9	2,115,043.6	2,066,942.5	47%	4%	49%
14/19	14	10	10	11	11			
coloring	33,323.7	366,322.9	394,885.2	398,181.5	350,956.5	60%	39%	1%
22/22	22	21	21	21	21			
composed-25-1-2	167.1	196.9	202.2	122.0	121.5	90%	10%	0%
10/10	10	10	10	10	10			
composed-25-1-25	213.8	251.1	259.0	1,434.1	144.1	90%	10%	0%
10/10	10	10	10	10	10			
composed-25-1-40	244.2	281.3	299.3	1,392.0	165.2	87%	13%	0%
10/10	10	10	10	10	10			
composed-25-1-80	284.6	433.3	832.2	899.1	289.8	100%	0%	0%
10/10	10	10	10	10	10			
composed-25-10-20	320.5	59,248.1	148,901.9	79,981.7	61,716.9	34%	66%	0%
10/10	10	10	10	10	10			
composed-75-1-2	471.9	714.3	735.0	3,097.8	417.1	88%	12%	0%
10/10	10	10	10	10	10			
composed-75-1-25	586.7	761.8	784.6	3,565.3	442.7	92%	8%	0%
10/10	10	10	10	10	10			

Table A.3: STR2 and cluster-minimality algorithms using dom/wdeg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
composed-75-1-40	634.1	814.8	840.4	1,501.6	464.6	94%	6%	0%
10/10	10	10	10	10	10			
composed-75-1-80	622.1	1,416.1	2,956.2	1,944.1	1,059.7	96%	4%	0%
10/10	10	10	10	10	10			
cril	2,765,996.6	625,532.0	1,726,567.6	630,008.0	675,353.3	44%	42%	14%
7/8	5	7	6	7	7			
dag-half	2,841,465.2	5,994,070.5	5,956,297.5	5,704,275.3	6,016,502.5	97%	2%	0%
14/25	13	3	3	4	3			
dag-rand	92,594.1	946,982.7	983,059.2	956,312.4	994,961.1	0%	68%	32%
25/25	25	22	22	22	22			
domino	288,996.3	293,498.9	293,327.7	375,990.5	293,595.7	0%	100%	0%
22/23	22	22	22	21	22			
driver	12,557.9	3,580,045.2	3,538,796.9	3,407,841.9	3,542,690.0	40%	53%	7%
7/7	7	4	4	4	4			
dubois	1,208,932.1	1,955,813.5	1,943,293.7	2,562,174.0	2,141,086.9	0%	100%	0%
7/13	7	6	6	6	6			
ehi-85	3,610.8	29,866.3	41,572.9	36,125.4	22,549.0	81%	0%	19%
100/100	100	100	100	100	100			
ehi-90	3,697.4	32,424.9	45,104.2	40,034.2	24,124.4	80%	0%	20%
100/100	100	100	100	100	100			
fapp01	4,865,546.5	6,323,888.0	6,042,528.0	6,041,389.4	6,223,084.6	33%	65%	2%
11/11	11	4	9	8	7			

Table A.3: STR2 and cluster-minimality algorithms using dom/wdeg (continued)

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
fapp02	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp03	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp04	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp05	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp06	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp07	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp08	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp09	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp10	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp11	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp12	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			

Table A.3: STR2 and cluster-minimality algorithms using dom/wdeg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
fapp13	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp14	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp15	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp16	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp17	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp18	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp19	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp20	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp21	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp23	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp24	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			

Table A.3: STR2 and cluster-minimality algorithms using dom/wdeg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
fapp25	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp26	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp27	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp31	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp33	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp36	-	-	-	-	-	-	-	-
0/11	-	-	-	-	-			
fapp39	-	-	-	-	-	-	-	-
0/10	-	-	-	-	-			
frb30-15	3,635.8	202,162.7	250,421.2	252,378.8	203,495.8	83%	17%	0%
10/10	10	10	10	10	10			
frb35-17	41,669.5	2,720,434.9	2,846,686.5	2,851,968.7	2,782,980.4	90%	10%	0%
10/10	10	8	10	8	8			
frb40-19	336,050.5	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
10/10	10	-	-	-	-			
frb45-21	2,937,978.2	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
10/10	10	-	-	-	-			

Table A.3: STR2 and cluster-minimality algorithms using dom/wdeg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
frb50-23	847,865.9	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
2/10	2	-	-	-	-			
frb53-24	-	-	-	-	-	-	-	-
0/10	-	-	-	-	-			
frb56-25	-	-	-	-	-	-	-	-
0/10	-	-	-	-	-			
frb59-26	-	-	-	-	-	-	-	-
0/10	-	-	-	-	-			
geom	72,522.9	812,351.1	1,053,114.7	948,983.9	829,741.6	66%	34%	0%
100/100	100	93	89	90	93			
golombRulerArity3	555,360.0	3,327,754.4	3,805,032.2	3,704,735.4	3,630,972.6	71%	10%	19%
9/14	9	5	5	5	5			
golombRulerArity4	248,948.8	249,340.3	249,361.2	249,363.8	249,334.0	63%	38%	0%
2/14	2	2	2	2	2			
graceful	1,569,034.4	2,547,922.5	2,615,515.5	2,543,652.6	2,548,601.8	57%	43%	0%
3/4	3	2	2	2	2			
hos	118,026.0	2,031,873.9	2,367,611.3	2,132,081.1	2,023,164.7	44%	56%	0%
12/14	12	9	9	9	9			
full-insertion	121,758.7	2,435,924.8	2,528,618.4	2,846,691.5	2,240,350.8	29%	70%	0%
32/41	32	22	22	20	23			
k-insertion	192,786.4	1,224,052.6	1,187,676.7	1,195,122.9	1,062,520.5	4%	95%	0%
17/33	17	15	16	15	16			

Table A.3: STR2 and cluster-minimality algorithms using dom/wdeg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
leighton-15	1,587,880.6	1,960,107.9	3,356,928.3	2,464,417.4	1,029,962.0	69%	7%	25%
9/26	8	7	5	6	8			
leighton-25	18,208.3	2,580,031.6	2,580,504.5	2,454,921.4	2,454,839.0	96%	4%	0%
6/31	6	4	4	4	4			
leighton-5	6,679.0	1,955,949.6	728,513.5	2,095,130.1	1,896,232.1	85%	10%	5%
8/8	8	6	8	6	6			
mug	3,600,012.7	1,471,388.5	1,238,042.7	3,237,005.5	1,317,636.5	12%	88%	0%
8/8	4	7	7	5	7			
myciel	243,099.8	1,701,089.3	1,717,949.6	1,685,531.4	1,686,995.4	34%	64%	2%
13/16	13	10	10	10	10			
register-fpsol	43,519.9	3,638,405.1	3,646,115.1	3,614,232.8	3,614,133.5	67%	0%	33%
6/37	6	3	3	3	3			
register-inithx	61,996.6	7,200,000.0	7,200,000.0	3,630,302.0	3,630,206.4	100%	0%	0%
6/32	6	-	-	3	3			
register-mulsol	8,519.7	3,208,591.9	2,361,435.6	1,640,681.7	1,532,017.3	68%	5%	27%
9/49	9	5	7	8	8			
register-zeroin	9,396.5	3,609,969.1	3,610,106.1	3,602,828.0	3,602,759.5	75%	0%	25%
6/31	6	3	3	3	3			
school	28,555.0	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
3/8	3	-	-	-	-			
sgb-book	537,648.5	437,689.8	741,300.6	658,572.3	436,777.0	82%	18%	0%
23/26	23	22	22	23	22			

Table A.3: STR2 and cluster-minimality algorithms using dom/wdeg (continued)

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
sgb-games	153,403.6	16,336.3	1,841,056.3	20,705.7	15,561.5	69%	31%	0%
4/4	4	4	3	4	4			
sgb-miles	666,374.8	2,178,878.0	1,887,992.1	2,127,445.4	1,226,239.2	71%	14%	15%
11/42	10	8	9	8	10			
sgb-queen	482,789.9	1,998,636.1	2,030,370.3	1,854,275.9	1,970,204.8	69%	14%	17%
15/50	15	11	12	12	11			
hanoi	484.9	1,320.7	1,329.4	1,331.3	1,332.0	0%	100%	0%
5/5	5	5	5	5	5			
haystacks	5,143,146.1	393,391.0	2,009,199.5	2,176,003.9	363,636.2	73%	27%	0%
7/51	2	7	6	5	7			
jnhSat	875.4	395,601.3	451,733.5	399,920.8	317,560.9	21%	6%	73%
16/16	16	16	16	16	16			
jnhUnsat	914.3	281,352.9	251,904.8	261,011.6	191,446.1	8%	0%	92%
34/34	34	34	34	34	34			
jobShop-e0ddr1	439,981.5	2,876,760.2	1,880,781.0	2,054,783.6	2,406,322.6	35%	63%	3%
5/10	5	4	4	4	4			
jobShop-e0ddr2	11,173.6	2,782,736.3	642,291.2	1,104,881.1	1,034,785.5	34%	62%	4%
4/10	4	4	4	4	4			
jobShop-enddr1	18,734.9	2,098,788.3	558,953.3	1,121,061.2	1,165,064.2	36%	61%	3%
9/10	9	9	9	9	9			
jobShop-enddr2	12,386.2	2,995,150.1	563,680.7	1,184,432.5	920,125.3	32%	64%	4%
3/6	3	3	3	3	3			

Table A.3: STR2 and cluster-minimality algorithms using dom/wdeg (continued)

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
jobShop-ewddr2	13,041.4	2,967,877.5	652,563.6	1,273,023.6	1,071,276.4	30%	66%	4%
10/10	10	10	10	10	10			
js-taillard-15	-	-	-	-	-	-	-	-
0/30	-	-	-	-	-			
knights	218,250.9	344,035.5	326,778.9	344,079.6	325,829.9	20%	40%	40%
10/19	10	10	10	10	10			
langford	25,072.5	557,275.3	212,287.1	401,710.4	562,027.6	59%	25%	16%
4/4	4	4	4	4	4			
langford2	6,014.7	44,159.2	46,302.2	45,942.5	40,536.3	46%	30%	24%
16/24	16	16	16	16	16			
langford3	199,954.4	2,477,131.9	1,806,972.6	2,249,839.4	2,296,071.1	82%	0%	18%
15/23	15	10	12	11	11			
langford4	145,051.3	1,525,418.5	1,402,432.2	1,479,883.8	1,495,135.6	42%	0%	58%
13/24	13	11	11	11	11			
lexVg	38,602.7	1,212,695.3	1,241,073.5	1,162,807.8	1,249,681.5	1%	99%	0%
63/63	63	56	56	56	55			
marc	47,944.0	188,050.7	191,275.8	187,885.4	185,305.0	0%	50%	50%
10/10	10	10	10	10	10			
modifiedRenault	1,420.0	19,826.9	17,410.7	28,743.2	17,842.3	58%	42%	0%
50/50	50	50	50	50	50			
nengfa	449,086.7	4,440,151.1	4,459,935.5	3,579,935.7	4,080,872.7	23%	40%	37%
5/10	5	2	2	3	3			

Table A.3: STR2 and cluster-minimality algorithms using dom/wdeg (continued)

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
ogdVg	458,909.9	2,264,148.8	1,968,976.1	2,018,979.6	1,793,415.9	11%	89%	0%
43/65	43	32	33	33	34			
os-gp	-	-	-	-	-	-	-	-
0/19	-	-	-	-	-			
os-taillard-10	-	-	-	-	-	-	-	-
0/30	-	-	-	-	-			
os-taillard-15	-	-	-	-	-	-	-	-
0/30	-	-	-	-	-			
os-taillard-4	72,518.8	718,753.1	243,910.9	443,736.7	422,644.8	27%	70%	3%
30/30	30	29	30	30	30			
os-taillard-5	2,217,837.9	4,875,877.6	3,001,446.5	4,888,191.9	3,672,278.8	38%	39%	23%
19/30	16	10	15	8	11			
os-taillard-7	1,742,861.1	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
7/30	7	-	-	-	-			
pigeons	261,160.4	207,801.8	424,787.9	322,454.2	209,119.5	90%	10%	0%
13/24	13	13	13	13	13			
pret	94,820.2	183,807.1	180,861.4	403,906.0	239,985.4	0%	100%	0%
4/8	4	4	4	4	4			
pseudo-aim	31,617.4	162,714.1	307,374.4	221,721.2	148,337.4	9%	90%	1%
48/48	48	48	47	48	48			
pseudo-garden	18.9	638.1	900.1	652.4	845.2	15%	85%	0%
6/7	6	6	6	6	6			

Table A.3: STR2 and cluster-minimality algorithms using dom/wdeg (continued)

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
pseudo-jnh	1,330.3	3,427,706.0	3,594,663.4	2,556,383.0	3,394,572.2	58%	9%	33%
16/16	16	10	11	13	10			
pseudo-par	40,346.9	2,118,631.7	1,891,449.7	2,063,282.5	1,913,312.5	0%	100%	0%
20/30	20	15	16	15	16			
pseudo-primesDimacs	15,907.7	50,153.1	48,654.3	4,814,335.1	4,813,928.1	0%	100%	0%
3/11	3	3	3	1	1			
pseudo-radar	13,605.2	3,768,486.7	3,640,323.6	3,700,945.6	3,640,578.8	0%	100%	0%
6/12	6	3	3	3	3			
pseudo-ssa	41,778.3	311,130.4	314,041.3	1,324,639.3	434,575.8	0%	100%	0%
7/8	7	7	7	6	7			
pseudo-uclid	384,581.6	2,888,425.4	2,686,612.6	2,728,544.4	2,691,541.1	0%	100%	0%
9/36	9	6	6	6	6			
QCP-10	298.0	60,312.1	89,034.8	55,149.6	62,606.4	17%	83%	0%
15/15	15	15	15	15	15			
QCP-15	78,940.8	134,226.6	132,522.4	97,142.1	102,843.9	25%	70%	5%
15/15	15	15	15	15	15			
QCP-20	4,181,471.6	1,384,722.1	2,082,329.8	1,016,116.7	1,116,712.2	41%	53%	6%
11/15	5	11	10	11	11			
QCP-25	5,403,980.1	837,841.5	1,259,769.9	2,281,713.4	2,252,573.3	100%	0%	0%
4/15	1	4	4	3	3			
queenAttacking	178,886.7	350,966.2	595,489.5	511,757.8	1,393,533.9	73%	14%	13%
4/10	4	4	4	4	4			

Table A.3: STR2 and cluster-minimality algorithms using dom/wdeg (continued)

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
queens	25,090.9	780,941.1	774,706.9	781,047.4	792,249.0	49%	30%	22%
10/14	10	9	9	9	9			
queensKnights	183,174.0	325,375.9	252,138.9	269,321.5	327,365.0	73%	22%	5%
10/18	10	10	10	10	10			
QWH-10	185.4	8,102.6	9,088.1	7,841.4	7,885.7	9%	91%	0%
10/10	10	10	10	10	10			
QWH-15	2,889.6	67,499.6	47,237.2	52,023.1	66,155.0	6%	93%	1%
10/10	10	10	10	10	10			
QWH-20	947,171.9	476,163.4	342,075.3	295,832.8	404,047.6	12%	84%	4%
10/10	9	10	10	10	10			
QWH-25	7,200,000.0	7,200,000.0	4,976,835.6	2,567,511.8	7,200,000.0	-	-	-
2/10	-	-	1	2	-			
radar-8-30-3-0	-	-	-	-	-	-	-	-
0/50	-	-	-	-	-			
radar-9-28-4-2	-	-	-	-	-	-	-	-
0/50	-	-	-	-	-			
ramsey3	35.9	28,197.2	44,420.3	34,086.5	28,236.6	49%	51%	0%
2/8	2	2	2	2	2			
ramsey4	1,031.5	790,753.0	1,049,965.7	739,042.6	339,974.5	43%	8%	49%
1/8	1	1	1	1	1			
rand-10-20-10	993.4	2,571.9	2,559.7	2,558.4	2,555.1	0%	100%	0%
20/20	20	20	20	20	20			

Table A.3: STR2 and cluster-minimality algorithms using dom/wdeg (continued)

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark # instances	**Avg time** completed	**Avg time** completed	**Avg time** completed	**Avg time** completed	**Avg time** completed	**Selection** %P	%A	%N
rand-2-23 10/10	435,346.4 10	447,983.2 10	1,723,637.9 10	1,546,868.7 10	377,903.1 10	93%	0%	7%
rand-2-24 10/10	962,257.9 10	806,797.6 10	4,949,429.3 6	3,740,633.4 9	711,405.7 10	90%	0%	10%
rand-2-25 10/10	1,928,226.8 10	1,983,856.8 10	5,700,465.6 3	5,269,924.9 5	2,486,206.4 10	87%	0%	13%
rand-2-26 4/10	3,452,167.9 4	3,999,124.8 3	7,200,000.0 -	7,200,000.0 -	7,200,000.0 -	-	-	-
rand-2-27 3/10	4,904,442.5 3	5,773,562.1 1	7,200,000.0 -	7,200,000.0 -	7,200,000.0 -	-	-	-
rand-2-30-15-fcd 50/50	4,645.5 50	325,392.7 50	319,265.3 50	330,560.8 50	329,544.4 50	81%	19%	0%
rand-2-30-15 50/50	8,320.8 50	475,385.8 50	520,874.4 50	509,584.4 50	477,459.9 50	85%	15%	0%
rand-2-40-19-fcd 50/50	502,987.5 50	6,671,432.2 5	6,785,650.2 4	6,697,807.3 5	6,677,144.1 5	87%	13%	0%
rand-2-40-19 50/50	1,024,005.2 50	7,120,635.8 1	7,127,550.9 1	7,129,885.0 1	7,104,168.8 1	89%	11%	0%
rand-2-50-23-fcd 8/50	3,921,389.5 8	7,200,000.0 -	7,200,000.0 -	7,200,000.0 -	7,200,000.0 -	-	-	-
rand-2-50-23 3/50	3,945,059.6 3	7,200,000.0 -	7,200,000.0 -	7,200,000.0 -	7,200,000.0 -	-	-	-

Table A.3: STR2 and cluster-minimality algorithms using dom/wdeg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
rand-3-20-20-fcd	290,359.2	2,536,704.0	1,271,039.4	1,987,355.7	2,269,090.8	58%	42%	0%
50/50	50	43	50	48	44			
rand-3-20-20	595,994.7	3,847,263.5	2,224,070.7	3,301,313.2	3,510,153.5	53%	47%	0%
50/50	50	34	47	40	37			
rand-3-24-24-fcd	2,723,303.9	6,633,305.8	6,103,911.6	6,354,532.7	6,648,164.6	72%	28%	0%
22/50	19	4	6	5	3			
rand-3-24-24	3,706,064.4	7,200,000.0	6,860,823.7	6,950,084.5	7,200,000.0	-	-	-
10/50	9	-	1	1	-			
rand-3-28-28-fcd	1,335,236.1	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
2/50	2	-	-	-	-			
rand-3-28-28	1,275,396.6	7,200,000.0	7,200,000.0	7,200,000.0	7,200,000.0	-	-	-
1/50	1	-	-	-	-			
rand-8-20-5	117,099.7	468,115.0	488,111.0	569,480.9	387,863.7	41%	53%	6%
20/20	20	20	20	20	20			
renault	963.8	35,439.0	19,435.6	32,323.2	21,894.7	46%	54%	0%
2/2	2	2	2	2	2			
rlfapGraphs	9,738.0	1,803,871.0	1,624,526.3	1,789,369.0	1,793,931.0	24%	76%	0%
14/14	14	13	14	12	13			
rlfapGraphsMod	32,877.1	600,269.4	946,641.2	529,850.2	539,188.1	46%	54%	0%
12/12	12	12	11	12	12			
rlfapScens11	3,060,212.7	113,393.7	4,517,665.8	2,726,426.6	993,615.6	94%	4%	2%
8/12	5	8	3	5	7			

Table A.3: STR2 and cluster-minimality algorithms using dom/wdeg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
rlfapScens	11,665.5	527,345.7	419,550.1	368,275.5	542,829.2	30%	69%	1%
11/11	11	11	11	11	11			
rlfapScensMod	12,715.7	265,567.6	802,788.4	206,607.6	259,862.9	54%	46%	0%
13/13	13	13	12	13	13			
schurrLemma	269,017.6	1,914,560.9	2,008,819.7	1,617,821.0	1,948,738.2	80%	5%	14%
9/10	9	7	7	8	7			
ssa	27,228.3	262,534.8	262,314.2	1,058,488.7	1,060,205.6	6%	94%	0%
7/8	7	7	7	6	6			
subs	1,421.4	3,029.8	3,236.0	2,862.6	2,886.7	52%	15%	33%
9/9	9	9	9	9	9			
super-jobShop-e0ddr1	1,292,156.7	5,922,714.4	3,656,765.9	7,200,000.0	3,883,255.5	75%	15%	9%
3/10	3	1	2	-	2			
super-jobShop-e0ddr2	-	-	-	-	-	-	-	-
0/10	-	-	-	-	-			
super-jobShop-enddr1	2,524,544.0	5,291,943.2	3,673,891.5	7,200,000.0	3,934,428.5	76%	17%	7%
3/10	2	2	2	-	2			
super-jobShop-enddr2	91,438.5	7,200,000.0	2,678,751.0	7,200,000.0	2,920,604.5	74%	18%	8%
2/6	2	-	2	-	2			
super-jobShop-ewddr2	43,852.4	7,200,000.0	5,447,144.4	7,200,000.0	2,732,403.6	71%	21%	8%
6/9	6	-	2	-	6			
super-js-taillard-15	-	-	-	-	-	-	-	-
0/25	-	-	-	-	-			

Table A.3: STR2 and cluster-minimality algorithms using dom/wdeg (continued)

	STR2	Cl-PerTuple	Cl-AllSol	Cl-Random	Cl-Portfolio			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
super-os-taillard-10	-	-	-	-	-	-	-	-
0/30	-	-	-	-	-			
super-os-taillard-15	-	-	-	-	-	-	-	-
0/30	-	-	-	-	-			
super-os-taillard-4	450,234.1	2,928,777.3	962,934.7	1,806,995.8	1,215,920.5	36%	36%	28%
28/30	27	22	28	26	27			
super-os-taillard-5	1,528,385.0	7,200,000.0	3,918,751.2	5,946,212.8	3,375,662.7	46%	21%	33%
8/28	8	-	6	3	7			
super-os-taillard-7	-	-	-	-	-	-	-	-
0/30	-	-	-	-	-			
super-queens	600,842.2	1,442,265.6	1,478,818.2	1,444,357.0	1,442,428.9	100%	0%	0%
5/14	5	4	4	4	4			
tightness0.1	69,167.6	641,680.5	1,129,957.2	774,925.5	4,064,362.9	14%	0%	86%
100/100	100	100	100	100	91			
tightness0.2	86,170.7	4,754,039.2	5,711,894.5	5,092,659.0	4,871,417.6	95%	5%	0%
100/100	100	51	41	50	47			
tightness0.35	75,475.9	5,428,074.1	5,386,234.2	5,511,861.9	5,500,011.6	65%	35%	0%
100/100	100	40	46	40	37			
tightness0.5	115,561.4	5,701,803.6	6,032,732.0	5,950,109.1	5,833,049.8	44%	56%	0%
100/100	100	37	33	34	33			
tightness0.65	100,633.0	3,100,915.2	3,573,962.0	3,087,074.3	3,243,634.0	37%	63%	0%
100/100	100	84	78	85	80			

Table A.3: STR2 and cluster-minimality algorithms using dom/wdeg (continued)

	STR2	**Cl-PerTuple**	**Cl-AllSol**	**Cl-Random**	**Cl-Portfolio**			
Benchmark	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Avg time**	**Selection**		
# instances	completed	completed	completed	completed	completed	**%P**	**%A**	**%N**
tightness0.8	108,416.4	1,262,723.1	1,434,802.0	1,130,900.2	1,253,463.9	41%	59%	0%
100/100	100	98	98	98	98			
tightness0.9	143,122.2	761,303.3	860,299.8	762,711.3	762,963.8	41%	59%	0%
100/100	100	97	95	98	96			
travellingSalesman-20	18,423.4	1,513,958.0	1,480,214.7	1,295,844.0	1,533,166.6	32%	68%	0%
15/15	15	13	14	14	13			
travellingSalesman-25	287,828.5	4,620,826.6	4,541,672.7	4,529,165.1	4,579,528.4	33%	67%	0%
15/15	15	6	7	6	6			
ukPuzzle	7.6	14.7	13.9	15.9	15.9	50%	50%	0%
1/22	1	1	1	1	1			
ukVg	205,115.5	1,718,691.6	1,853,669.4	1,675,066.2	1,474,709.9	13%	86%	0%
36/65	36	29	28	30	30			
varDimacs	301,255.7	1,851,967.7	1,710,796.3	1,730,780.2	1,712,151.8	4%	96%	0%
9/9	9	7	7	7	7			
wordsVg	137,979.8	1,716,064.7	1,579,475.5	1,538,835.2	1,600,062.5	2%	98%	0%
65/6	65	52	54	54	53			

Bibliography

[Amaldi *et al.*, 2010] Edoardo Amaldi, Claudio Iuliano, and Romeo Rizzi. Efficient Deterministic Algorithms for Finding a Minimum Cycle Basis in Undirected Graphs. In *Integer Programming and Combinatorial Optimization (IPCO 2010)*, volume 6080 of *LNCS*, pages 397–410, 2010.

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

[Balafrej *et al.*, 2015] Amine Balafrej, Christian Bessière, and Anastasia Paparrizou. Multi-Armed Bandits for Adaptive Constraint Propagation. In *Proceedings of the 24 th International Joint Conference on Artificial Intelligence*, pages 290–296, 2015.

[Bayer *et al.*, 2007] Kenneth M. Bayer, Martin Michalowski, Berthe Y. Choueiry, and Craig A. Knoblock. Reformulating CSPs for Scalability with Application to Geospatial Reasoning. In *Proceedings of the 13 th International Conference on Principles and Practice of Constraint Programming (CP 07)*, pages 164–179, Providence, Rhode Island, 2007. LNCS 4741, Springer.

[Bennaceur and Affane, 2001] Hachemi Bennaceur and Mohamed-Salah Affane. Partition-k-AC: An Efficient Filtering Technique Combining Domain Partition and Arc Consistency. In *Principles and Practice of Constraint Programming (CP 01)*, volume 2239 of *LNCS*, pages 560–564. Springer, 2001.

[Bessiere *et al.*, 2013] Christian Bessiere, Hélène Fargier, and Christophe Lecoutre. Global Inverse Consistency for Interactive Constraint Satisfaction. In *Proceedings of 19 th International Conference on Principles and Practice of Constraint Programming (CP 13)*, volume 8124 of *LNCS*, pages 159–174, 2013.

[Boussemart *et al.*, 2004] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting Systematic Search by Weighting Constraints. In *Proceedings of the 16 th European Conference on Artificial Intelligence (ECAI 2004)*, pages 146–150, 2004.

[Box *et al.*, 1978] George E.P. Box, William G. Hunter (Author), and J.S̃tuart Hunter. *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*. Wiley, 1978.

[Debruyne and Bessière, 2001] Romuald Debruyne and Christian Bessière. Domain Filtering Consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.

[Dechter and Pearl, 1987] Rina Dechter and Judea Pearl. The Cycle-Cutset Method for improving Search Performance in AI Applications. In *Proceedings of the Third IEEE Conference on AI Applications*, pages 224–230, Orlando, FL, 1987.

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

[Dechter and Pearl, 1989] Rina Dechter and Judea Pearl. Tree Clustering for Constraint Networks. *Artificial Intelligence*, 38(3):353–366, 1989.

[Dechter and Pearl, 1992] Rina Dechter and Judea Pearl. Structure Identification in Relational Data. *Artificial Intelligence*, 58(1-3):237–270, 1992.

[Dechter and van Beek, 1997] Rine Dechter and Peter van Beek. Local and Global Relational Consistency. *Theor. Comput. Sci.*, 173(1):283–308, 1997.

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

[Demeulenaere *et al.*, 2016] Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-Table: Efficiently Filtering Table Constraints with Reversible Sparse Bit-Sets. In *Proceedings of the 22 nd International Conference on Principles and Practice of Constraint Programming (CP 2016)*, pages 207–223. Springer International Publishing, 2016.

[Freuder and Elfe, 1996] Eugene C. Freuder and Charles D. Elfe. Neighborhood Inverse Consistency Preprocessing. In *Proceedings of AAAI-96*, pages 202–208, 1996.

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

[Freuder, 1985] Eugene C. Freuder. A Sufficient Condition for Backtrack-Bounded Search. *JACM*, 32 (4):755–761, 1985.

[Freuder, 1991] Eugene C. Freuder. Completable Representations of Constraint Satisfaction Problems. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR 91)*, pages 186–195, 1991.

[Geschwender *et al.*, 2013] Daniel J. Geschwender, Shant Karakashian, Robert J. Woodward, Berthe Y. Choueiry, and Stephen D. Scott. Selecting the Appropriate Consistency Algorithm for CSPs Using Machine Learning Classifiers. In *Proceedings of AAAI-2013*, pages 1611–1612, 2013.

[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. Technical Report TR-UNL-CSE-2016-0003, Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE, 2016.

[Golumbic, 1980] Martin C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press Inc., 1980.

[Gomes and Selman, 2001] Carla P. Gomes and Bart Selman. Algorithm Portfolios. *Artificial Intelligence*, 126(1):43–62, 2001.

[Gottlob *et al.*, 1999] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A Comparison of Structural CSP Decomposition Methods. In *Proceedings of the 16 th International Joint Conference on Artificial Intelligence*, pages 394–399, Stockholm, Sweden, 1999.

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

[Gyssens, 1986] M. Gyssens. On the Complexity of Join Dependencies. *ACM Trans. Database Systems*, 11(1):81–108, 1986.

[Hall *et al.*, 2009] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The WEKA Data Mining Software: an Update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[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.

[Horton, 1987] Joseph D. Horton. A Polynomial-Time Algorithm to Find the Shortest Cycle Basis of a Graph. *SIAM Journal on Computing*, 16(2):358–366, 1987.

[Janssen *et al.*, 1989] Philippe Janssen, Philippe Jégou, Bernard Nougier, and Marie-Catherine Vilarem. A Filtering Process for General Constraint-Satisfaction Problems: Achieving Pairwise-Consistency Using an Associated Binary Representation. In *IEEE Workshop on Tools for AI*, pages 420–427, 1989.

[Jeavons *et al.*, 1994] Peter G. Jeavons, David A. Cohen, and Marc Gyssens. A Structural Decomposition for Hypergraphs. *Contemporary Mathematics*, 178:161–177, 1994.

[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 AAAI-2010*, pages 101–107, 2010.

[Karakashian *et al.*, 2012] Shant Karakashian, Robert J. Woodward, Berthe Y. Choueiry, and Stephen D. Scott. Algorithms for the Minimal Network of a CSP and a Classifier for Choosing Between Them. TR-UNL-CSE-2012-0007, 2012.

[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 AAAI-2013*, pages 466–473, 2013.

[Karakashian, 2013] Shant Karakashian. *Practical Tractability of CSPs by Higher Level Consistency and Tree Decomposition*. PhD thesis, CSE, UNL, Lincoln, NE, May 2013.

[Kavitha *et al.*, 2007] Telikepalli Kavitha, Kurt Mehlhorn, and Dimitrios Michail. New Approximation Algorithms for Minimum Cycle Bases of Graphs. In *Symposium on Theoretical Aspects of Computer Science (STACS 2007)*, volume 4393 of *LNCS*, pages 512–523, 2007.

[Kjærulff, 1990] Uffe Kjærulff. Triangulation of Graphs – Algorithms Giving Small Total State Space. Technical Report R-90-09, Aalborg University, 1990.

[le Clément de Saint-Marcq *et al.*, 2013] Vianney le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. Sparse-Sets for Domain Implementation. In *Proceedings of the CP 2013 Workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, 2013.

[Lecoutre, 2011] Christophe Lecoutre. STR2: Optimized Simple Tabular Reduction for Table Constraints. *Constraints*, 16(4):341–371, 2011.

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

[Maier, 1983] David Maier. *Theory of Relational Databases*. Computer Science Press, 1983.

[Mairy *et al.*, 2014] Jean-Baptiste Mairy, Yves Deville, and Christophe Lecoutre. Domain k-Wise Consistency Made as Simple as Generalized Arc Consistency. In *Proceedings of the 11 th Integration of AI and OR Techniques in Constraint Pro-*

gramming for Combinatorial Optimization Problems (CPAIOR 2014), volume 8451 of *LNCS*, pages 235–250. Springer, 2014.

[Mehlhorn and Michail, 2009] Kurt Mehlhorn and Dimitrios Michail. Minimum Cycle Bases: Faster and Simpler. *ACM Trans. Algorithms*, 6(1):1–13, December 2009.

[Montanari, 1974] Ugo Montanari. Networks of Constraints: Fundamental Properties and Application to Picture Processing. *Information Sciences*, 7:95–132, 1974.

[O'Mahony *et al.*, 2008] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using Case-based Reasoning in an Algorithm Portfolio for Constraint Solving. In *Proceedings of the Irish Conference on AI and Cognitive Science*, pages 210–216, 2008.

[Paparrizou and Stergiou, 2016] Anastasia Paparrizou and Kostas Stergiou. Strong Local Consistency Algorithms for Table Constraints. *Constraints*, 21(2):163–197, Apr 2016.

[Paparrizou and Stergiou, 2017] Anastasia Paparrizou and Kostas Stergiou. On Neighborhood Singleton Consistencies. In *Proceedings of the 26 th International Joint Conference on Artificial Intelligence*, pages 736–742, 2017.

[Prosser, 1993] Patrick Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9 (3):268–299, 1993.

[Rice, 1976] John R. Rice. The Algorithm Selection Problem. *Advances in Computers*, 15:65–118, 1976.

[Sabin and Freuder, 1997] Daniel Sabin and Eugene C. Freuder. Understanding and Improving the MAC Algorithm. In *Proceedings of the Third International Confer-*

ence on Principles and Practice of Constraint Programming (CP 97), volume 1330 of *LNCS*, pages 167–181. Springer, 1997.

[Samaras and Stergiou, 2005] Nikolaos Samaras and Kostas Stergiou. Binary Encodings of Non-Binary Constraint Satisfaction Problems: Algorithms and Experimental Results. *Journal of Artificial Intelligence Research*, 24:641–684, 2005.

[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 20 th International Conference on Principles and Practice of Constraint Programming (CP 2014)*, volume 8656 of *Lecture Notes in Computer Science*, pages 688–704. Springer, 2014.

[Ullmann, 2007] Julian R. Ullmann. Partition Search for Non-binary Constraint Satisfaction. *Information Sciences*, 177(18):3639–3678, September 2007.

[Waltz, 1975] David Waltz. Understanding Line Drawings of Scenes with Shadows. In P.H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, Inc., 1975.

[Woodward and Choueiry, 2017] Robert J. Woodward and Berthe Y. Choueiry. Weight-Based Variable Ordering in the Context of High-Level Consistencies. *ArXiv e-prints*, November 2017.

[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 AAAI-2011*, pages 112–119, 2011.

[Xu *et al.*, 2008] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *JAIR*, 32(1):565–606, 2008.