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

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

12-2011

Relational Neighborhood Inverse Consistency for Constraint Satisfaction: A Structure-Based Approach For Adjusting Consistency & Managing Propagation

Robert J. Woodward
University of Nebraska-Lincoln, robertwoodward@huskers.unl.edu

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

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Woodward, Robert J., "Relational Neighborhood Inverse Consistency for Constraint Satisfaction: A Structure-Based Approach For Adjusting Consistency & Managing Propagation" (2011). *Computer Science and Engineering: Theses, Dissertations, and Student Research*. 35.
http://digitalcommons.unl.edu/computerscidiss/35

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.

RELATIONAL NEIGHBORHOOD INVERSE CONSISTENCY FOR CONSTRAINT SATISFACTION: A STRUCTURE-BASED APPROACH FOR ADJUSTING CONSISTENCY & MANAGING PROPAGATION

by

Robert J. Woodward

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Berthe Y. Choueiry

Lincoln, Nebraska

December, 2011

RELATIONAL NEIGHBORHOOD INVERSE CONSISTENCY FOR CONSTRAINT SATISFACTION: A STRUCTURE-BASED APPROACH FOR ADJUSTING CONSISTENCY & MANAGING PROPAGATION

Robert J. Woodward, M. S.

University of Nebraska, 2011

Adviser: Berthe Y. Choueiry

Freuder and Elfe [1996] introduced Neighborhood Inverse Consistency (NIC) as a local consistency property defined on the *values* in the variables' domains of a Constraint Satisfaction Problem (CSP). Debruyne and Bessière [2001] showed that enforcing NIC on binary CSPs is ineffective on sparse graph and too costly on dense graphs. In this thesis, we propose Relational Neighborhood Inverse Consistency (RNIC), an extension of NIC defined as a local consistency property on the *tuples* of the relations of a CSP. We characterize RNIC for both binary and non-binary CSPs, and propose an algorithm for enforcing it whose complexity is bounded by the degree of the dual graph on which the algorithm is applied. We propose to reduce the computational cost of our algorithm by reformulating the dual graph of the CSP. We present two reformulation techniques and their combinations, and discuss their effects on the consistency property enforced by the algorithm. We also describe a selection policy for choosing an appropriate reformulation technique, tying together the various components of our approach, which we show to outperforms, in a statistically significant manner, other common approaches for solving benchmark problems. Finally, we study the effect of the structure of the dual graph on the ordering of the propagation queue of our algorithm when applied as a preprocessing step to backtrack search and also as a lookahead strategy during search. We conclude, empirically, that the most effective ordering is the one that follows the tree decomposition of the dual graph.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Berthe Y. Choueiry, for her continued support and encouragement. I am also thankful to the members of the committee, Dr. Christian Bessiere, Dr. Sebastian Elbaum, and Dr. Peter Revesz for their invaluable feedback that allowed me to improve the analysis and presentation of my work. I am especially indebted to Dr. Bessiere for many stimulating discussions about the fundamental ideas in this thesis, for his support and enthusiasm, and for his substantial input to the theoretical proofs and counterexamples.

I am especially grateful to Shant Karakashian for creating an extensive, flexible, and reliable solver framework that I exploited to build, test, and evaluate my algorithms. Shant and I first started collaborating on the R($*$,m)C property, which I had originally designed in a constraint-based system for solving the Minesweeper puzzle. Shant generalized the concept to arbitrary CSPs, and designed and implemented advanced data structures that made the algorithm competitive and the concept a reality. I learned a lot from working with Shant, both for the design and implementation of advanced software as well as for conducting research. He has contributed to the research presented in Chapter 4 of this thesis. Further, Shant has been a real friend to me since I started doing research at the University of Nebraska-Lincoln (UNL) by supporting my ideas and always being available to listen and help. Christopher Reeson implemented the algorithm for computing the minimal dual graphs of a Constraint Satisfaction Problem (CSP), which is a key component of the various methods proposed in this thesis. I would like to thank Elizabeth Claassen and Dr. David B. Marx of the Department of Statistics at UNL for their help with designing the statistical analysis.

My first steps in research in Computer Science as an undergraduate student under

the supervision of Dr. Choueiry would not have been easy or perhaps even possible without the two awards that I received from the Undergraduate Creative Activities and Research Experiences (UCARE) program. The careful mentoring of Dr. Laura Damuth, Director of Undergraduate Research and Fellowship Advisor at UNL, and her generosity have allowed me to realize my aspirations and enabled me to apply to and receive the nationally competitive Barry M. Goldwater Scholarship.

Finally, I am grateful to my loving family, who encouraged me to pursue my passion of Computer Science. I am especially grateful to my wife, Allison, who lovingly and patiently puts up with my late nights spent on research.

This research was partially supported by a National Science Foundation (NSF) Graduate Research Fellowship grant number 1041000 and NSF Grant No. RI-111795. Experiments were conducted on the equipment of the Holland Computing Center at UNL.

Contents

Contents **v**

List of Figures **x**

List of Tables **xii**

1 Introduction **1**

1.1 Motivation 2

1.2 Contributions 3

1.2.1 Relational Neighborhood Inverse Consistency 4

1.2.2 Enforcing RNIC 4

1.2.3 RNIC variations 4

1.2.4 Dual graph selection 5

1.2.5 Evaluation of RNIC 5

1.2.6 Queue management 5

1.3 Outline of Thesis 5

2 Background **7**

2.1 Constraint Satisfaction Problem (CSP) 8

2.2 Graphical Representations 10

2.2.1 Constraint graph, hypergraph, and primal graph 11
2.2.2 Dual encoding 12
2.3 Reformulation Strategies 13
2.3.1 Triangulation 13
2.3.2 Redundancy removal 15
2.4 Consistency: Properties & Algorithms 15
2.4.1 Domain consistency properties 16
2.4.2 Relational consistency properties 17
2.4.3 Comparing consistency properties 18
2.5 Solving CSPs 18
2.6 Related Work 19
2.6.1 For binary CSPs 20
2.6.2 For non-binary CSPs 20

3 Relational Neighborhood Inverse Consistency & Dual Graphs 22
3.1 Defining RNIC 23
3.2 Comparing RNIC and R($*$,m)C 24
3.3 Comparing RNIC and Domain Filtering 26
3.4 Structure of the Dual Graph of a Binary CSP 28
3.4.1 Binary CSP with a complete constraint graph 28
3.4.2 Binary CSP with a non-complete constraint graph 33
3.5 RNIC on a Binary CSPs 34
3.5.1 Comparing RNIC and NIC 34
3.5.2 Effects of the dual-graph's structure on RNIC 35

4 Enforcing RNIC 38
4.1 An Algorithm for RNIC 38

4.1.1 SearchSupport . . . 40
4.1.2 Complexity analysis . . . 41
4.2 Enforcing RNIC versus R($*$,m)C . . . 42
4.3 Reformulating the Dual Graph . . . 43
4.3.1 Removing redundant edges: wRNIC . . . 44
4.3.2 Triangulating the dual graph: triRNIC . . . 46
4.3.3 Triangulate a minimal dual graph: wtriRNIC . . . 47
4.3.4 Select the appropriate RNIC: selRNIC . . . 47

5 Evaluating RNIC **50**
5.1 Experimental Setup . . . 50
5.1.1 Measured parameters . . . 51
5.1.2 Method for statistical analysis . . . 52
5.2 Global Rankings . . . 53
5.3 Detailed Analysis . . . 59
5.3.1 The aim-100 & aim-200 benchmarks . . . 59
5.3.2 The dag-rand benchmark . . . 60
5.3.3 The modifiedRenault benchmark . . . 61
5.3.4 The lexVg benchmark . . . 61
5.3.5 All other results . . . 63

6 Propagation-Queue Management **64**
6.1 Queue-Management Strategies . . . 65
6.1.1 Exact strategies . . . 66
6.1.2 Lazy strategies . . . 67
6.2 Experimental Setup . . . 68
6.3 Pre-Processing: Empirical Evaluations . . . 70

6.3.1 Enforcing triRNIC 70

6.3.2 Enforcing wtriRNIC 71

6.4 Lookahead: Empirical Evaluations 72

6.4.1 Enforcing triRNIC 72

6.4.2 Enforcing wtriRNIC 74

7 Conclusions and Future Work **76**

7.1 Summary of Contributions 76

7.2 Directions for Future Research 78

7.3 Final Note 80

A Data Sets **81**

A.1 Binary CSPs 81

A.2 Non-Binary CSPs 93

B Code Documentation **98**

B.1 Data Structures 98

B.1.1 constraint_graph Struct Reference 99

B.1.2 constraint_graph_edge Struct Reference 99

B.1.3 constraint_graph_node Struct Reference 100

B.1.4 htable Struct Reference 101

B.1.5 light_stack Struct Reference 104

B.1.6 light_stack_node Struct Reference 104

B.1.7 llist Struct Reference 104

B.1.8 llist_node Struct Reference 105

B.1.9 s_node Struct Reference 106

B.1.10 set Struct Reference 106

B.1.11 tree_map Struct Reference 107
B.1.12 tree_map_node Struct Reference 107
B.1.13 tuple_tag Struct Reference 108
B.2 File Documentation 110
B.2.1 nic2.c File Reference 110
B.2.2 nicprocedures.c File Reference 137

Bibliography **141**

List of Figures

2.1 A graph coloring problem [Dechter, 2003]. 9

2.2 The CSP of the crypto-arithmetic puzzle SEND + MORE = MONEY [Dechter, 2003]. 9

2.3 The hypergraph of a small CSP. 11

2.4 The primal graph of a small CSP. 11

2.5 The dual graph of a small CSP. 12

2.6 Triangulating a dual graph. 14

2.7 A minimal dual graph. 15

3.1 The dual graph of a small CSP. 23

3.2 Comparing RNIC with R($*$,m)C. 25

3.3 Configurations illustrating Theorem 4. 26

3.4 The CSP is RNIC+DF but not SGAC. 27

3.5 The CSP is SGAC but not RNIC+DF. 27

3.6 Some domain filtering properties. 28

3.7 A complete constraint graph of n vertices. 28

3.8 The dual graph corresponding to the CSP in Figure 3.7. 28

3.9 The path for the constraints over variables $V_{i\geq 2}$ of the grid of Figure 3.8. . . . 29

3.10 A complete constraint graph with 3 variables. 30

3.11 The dual graph of a complete constraint graph with 3 variables. 30
3.12 A complete constraint graph with k variables. 31
3.13 The dual graph of a complete constraint graph with k variables. 31
3.14 A complete constraint graph with $k+1$ variables. 31
3.15 The dual graph of a complete constraint graph with $k+1$ variables. 31
3.16 A complete constraint graph with 5 variables. 33
3.17 A redundancy-free dual graph of a complete constraint graph with five variables, which does not form a grid structure. 33
3.18 A complete constraint graph with 5 variables. 34
3.19 The redundancy-free dual graph of a the constraint graph with 5 variables. . . 34
3.20 The binary CSP is NIC but not RNIC+DF. 34
3.21 Binary CSP is RNIC+DF but not NIC. 34
3.22 A redundancy-free configuration of four binary constraints. 36
3.23 One possible labeling of the edges incident to C_1. 36
3.24 The other possible labeling of the edges incident to C_1. 36

4.1 Variations of RNIC. 44
4.2 A minimal dual graph. 45
4.3 Relating RNIC, wRNIC, R($*$,m)C, and wR($*$,m)C. 46
4.4 Triangulating a dual graph. 46
4.5 Triangulating a minimal dual graph. 47
4.6 Relating RNIC, R($*$,m)C, and their studied variations. 48
4.7 Selecting a dual graph for selRNIC. 48

6.1 A triangulated dual graph (left) along with a perfect elimination ordering (center) and a maximal cliques ordering (right) where the orderings proceed from bottom to top. 65

List of Tables

5.1 Overview of the binary benchmarks tested (Part A). 55
5.2 Overview of the binary benchmarks tested (Part B). 56
5.3 Overview of the non-binary benchmarks tested. 57
5.4 RNIC/selRNIC completes the largest number of instances, and solves, backtrack free, the largest number of instances. 60
5.5 Despite the high density, RNIC is able to perform well. 61
5.6 RNIC is hindered by the high density of the dual graph, but its weakened versions outperform all others. 62
5.7 GAC is best on CPU, triRNIC/selRNIC is best on #BF. 62

6.1 Proposed queue management strategies. 68
6.2 Pre-processing: QMSs for enforcing triRNIC. 71
6.3 Pre-processing: QMSs for enforcing triRNIC on solvable instances. 71
6.4 Pre-processing: QMSs for enforcing triRNIC on unsolvable instances. 71
6.5 Pre-processing: QMSs for enforcing wtriRNIC. 72
6.6 Pre-processing: QMSs for enforcing wtriRNIC on solvable instances. 72
6.7 Pre-processing: QMSs for enforcing wtriRNIC on unsolvable instances. 72
6.8 Lookahead: QMSs for enforcing triRNIC. 73
6.9 Lookahead: QMSs for enforcing triRNIC on solvable instances. 73

6.10 Lookahead: QMSs for enforcing triRNIC on unsolvable instances. 73
6.11 Lookahead: QMSs for enforcing wtriRNIC. 74
6.12 Lookahead: QMSs for enforcing wtriRNIC on solvable instances. 74
6.13 Lookahead: QMSs for enforcing wtriRNIC on unsolvable instances. 74

A.1 Statistical analysis of the composed-25-1-25 benchmark. 82
A.2 Statistical analysis of the composed-25-1-2 benchmark. 82
A.3 Statistical analysis of the composed-25-1-40 benchmark. 83
A.4 Statistical analysis of the composed-25-1-80 benchmark. 83
A.5 Statistical analysis of the composed-25-10-20 benchmark. 83
A.6 Statistical analysis of the composed-75-1-25 benchmark. 83
A.7 Statistical analysis of the composed-75-1-2 benchmark. 84
A.8 Statistical analysis of the composed-75-1-40 benchmark. 84
A.9 Statistical analysis of the composed-75-1-80 benchmark. 84
A.10 Statistical analysis of the ehi-85 benchmark. 85
A.11 Statistical analysis of the ehi-90 benchmark. 85
A.12 Statistical analysis of the QCP-10 benchmark. 85
A.13 Statistical analysis of the driver benchmark. 85
A.14 Statistical analysis of the frb35-17 benchmark. 86
A.15 Statistical analysis of the frb40-19 benchmark. 86
A.16 Statistical analysis of the frb45-21 benchmark. 86
A.17 Statistical analysis of the geom benchmark. 86
A.18 Statistical analysis of the langford benchmark. 87
A.19 Statistical analysis of the marc benchmark. 87
A.20 Statistical analysis of the QCP-15 benchmark. 87
A.21 Statistical analysis of the rand-2-23 benchmark. 87

A.22 Statistical analysis of the rand-2-24 benchmark. 88
A.23 Statistical analysis of the rand-2-30-15-fcd benchmark. 88
A.24 Statistical analysis of the rand-2-30-15 benchmark. 88
A.25 Statistical analysis of the rand-2-40-19-fcd benchmark. 89
A.26 Statistical analysis of the rand-2-40-19 benchmark. 89
A.27 Statistical analysis of the tightness0.1 benchmark. 89
A.28 Statistical analysis of the tightness0.2 benchmark. 89
A.29 Statistical analysis of the tightness0.35 benchmark. 90
A.30 Statistical analysis of the tightness0.5 benchmark. 90
A.31 Statistical analysis of the tightness0.65 benchmark. 90
A.32 Statistical analysis of the tightness0.8 benchmark. 90
A.33 Statistical analysis of the tightness0.9 benchmark. 91
A.34 Statistical analysis of the coloring benchmark. 91
A.35 Statistical analysis of the frb30-15 benchmark. 91
A.36 Statistical analysis of the hanoi benchmark. 92
A.37 Statistical analysis of the QWH-10 benchmark. 92
A.38 Statistical analysis of the QWH-15 benchmark. 92
A.39 Statistical analysis of the aim-50 benchmark. 93
A.40 Statistical analysis of the dubois benchmark. 94
A.41 Statistical analysis of the ssa benchmark. 94
A.42 Statistical analysis of the travellingSalesman-20 benchmark. 94
A.43 Statistical analysis of the travellingSalesman-25 benchmark. 94
A.44 Statistical analysis of the jnhSat benchmark. 95
A.45 Statistical analysis of the jnhUnsat benchmark. 95
A.46 Statistical analysis of the rand-3-20-20-fcd benchmark. 95
A.47 Statistical analysis of the rand-3-20-20 benchmark. 95

A.48 Statistical analysis of the rand-3-24-24-fcd benchmark. 96
A.49 Statistical analysis of the ogdVg benchmark. 96
A.50 Statistical analysis of the ukVg benchmark. 96
A.51 Statistical analysis of the wordsVg benchmark. 96
A.52 Statistical analysis of the pret benchmark. 97
A.53 Statistical analysis of the rand-10-20-10 benchmark. 97
A.54 Statistical analysis of the rand-8-20-5 benchmark. 97
A.55 Statistical analysis of the varDimacs benchmark. 97

Chapter 1

Introduction

An important result in Constraint Processing (CP) ties the tractability[1] of a Constraint Satisfaction Problem to the level of consistency that it satisfies. Solving difficult problems often requires enforcing higher order consistency, which typically requires the use of more costly algorithms in time and/or in space. Freuder and Elfe [1996] introduced Neighborhood Inverse Consistency (NIC) for Constraint Satisfaction problems (CSPs) as a particularly promising consistency property because:

1. Enforcing it is light in terms of space requirements (inverse consistency is enforced by filtering the variables domains); and

2. It focuses the attention on where a variable's value most tightly interacts with the problem, namely its neighborhood.

Despite its promise and filtering effectiveness, NIC remains relatively unexploited because the algorithm for enforcing it is too costly in terms of processing time, which prevented its use on dense networks or in a lookahead scheme during backtrack search.

[1]The tractability of a problem is the ability to solve it in time polynomial in the size of the input, which, in the case of the CSP, is the number of variables.

In [Woodward *et al.*, 2011b], we generalized NIC to Relational Neighborhood Consistency (RNIC) for filtering relations. Although, Bacchus *et al.* [2002] had already identified the same property as RNIC to hold when the arc-consistency property holds in on the dual graph[2] of the CSP, they do not provide a practical algorithm for enforcing it, study its usefulness in practice, or compare to any consistency properties other than arc consistency, all of which we examine in this thesis.

1.1 Motivation

Scalability is one of the biggest challenges in computing from the theoretical point of view and also, importantly, in practice. CSPs are in general $\mathcal{NP}$-complete, and backtrack search is the only known sound and complete algorithm for solving them. The goal of consistency algorithms is to remove, from the problem or from the search tree, those components (i.e., variables values, constraint tuples, and subtrees) that are inconsistent with the constraints, and thus cannot participate in a solution. The higher the consistency level enforced by a consistency algorithm, the stronger the pruning and the lesser the search effort. However, higher consistency levels also take more time/space to enforce.

Having a CSP solver dynamically determine the amount of consistency to enforce, either over the whole CSP or determining parts of the CSP to enforce a consistency on, is the ideal goal. Dynamically determining the consistency level allows for enforcing the higher, more expensive, consistency levels in areas of the problem where the efforts are required, but in the other parts of the problem, where the higher consistency levels is not required, to be filtered by the lower, cheaper, levels of consistency. To date, most of the research is conducted on improving algorithms for enforcing low-levels of

[2] The dual graph is defined in Section 2.2.

consistency, little research is conducted on higher-levels of consistency or algorithms that adopt the consistency level to the problem.

1.2 Contributions

In this thesis, we focus on the relation-filtering property RNIC, introduce variations of the property based on the dual graph RNIC enforces on, and study its performance to characterize and improve their behavior on benchmark problems. Preliminary work on the RNIC property and the algorithm for enforcing it was published in [Woodward *et al.*, 2011b; 2011a], a detailed description of the reformulations was given in [Woodward *et al.*, 2011c], and these works were combined in [Woodward *et al.*, 2011d]. We present six main contributions:

1. Introduce RNIC, a new consistency property that operates by on the relations of the CSP and characterize it on both binary and non-binary CSPs, see Chapter 3.
2. Give an algorithm for enforcing RNIC and compare its filtering power to other consistency methods, see Chapter 4.
3. Describe three variations of RNIC and classify the resulting filtering power of each variation.
4. We propose a strategy for automatically choosing the appropriate property to enforce, which is empirically evaluated to outperform other techniques compared in a statistically significant manner.
5. We compare the performance of RNIC and its variations to other common local consistency techniques on difficult benchmark problems, see Chapter 5.

6. Propose and evaluate four new queue-management strategies for RNIC, see Chapter 6.

Below, we briefly discuss each of these contributions and summarize our results.

1.2.1 Relational Neighborhood Inverse Consistency

We introduce a new consistency property, Relational Neighborhood Inverse Consistency (RNIC). RNIC is an extension of NIC on the tuples of the relations of a CSP, rather than on the values in the variables' domains. The benefit of RNIC is that it adapts to the topology of its neighborhood. Further, it does not require introducing new relations to the CSP, but filters existing relations. We also characterize RNIC filtering power compared to other common consistency methods, and on binary and non-binary CSPs.

1.2.2 Enforcing RNIC

RNIC can be enforced by ensuring that every tuple in every relation has a valid support in its neighborhood, and removing those that do not. The complexity is polynomial in the number of relations for a fixed degree dual graph.

1.2.3 RNIC variations

We propose to reduce the computation cost and/or strength propagation by reformulating the dual graph of the CSP. The reformulations are using a minimal dual graph (removing redundant edges in the dual graph), and/or triangulating the considered dual graph.

1.2.4 Dual graph selection

We introduce a selection criteria to select which of the four dual graphs (the original, minimal, triangulated, and triangulated minimal) to enforce RNIC on, which is shown to be statistically advantageous.

1.2.5 Evaluation of RNIC

We evaluate RNIC and its variations when compared to other commonly used local consistency techniques on the CSP Solver Competition benchmark problems. Further, we identify situations where RNIC performs well, in structured problems, and situations where RNIC does not perform well, in random problems.

1.2.6 Queue management

We propose four new propagation-queue strategies (two exact and two approximate) for RNIC. We empirically compare the different heuristics of the queue for both pre-processing and lookahead. We conclude that the best strategy is full lookahead following the structure of the tree decomposition.

1.3 Outline of Thesis

This thesis is structured as follows. Chapter 2 reviews background information about CSPs. Chapter 3 introduces RNIC, characterizes it with other consistency techniques and on binary and non-binary CSPs. Chapter 4 describes an algorithm for enforcing RNIC on the dual encoding of the CSP. It also discusses three variations of dual graphs for RNIC to use, and a strategy for deciding which of the four properties to enforce. Chapter 5 discusses our experimental results, where we compare the performance of

the resulting mechanisms on difficult benchmark problems. Chapter 6 discusses queue management strategies for RNIC, and evaluates a strategy to use. Chapter 7 discusses the extension of our approach to relations specified as conflicts or in intension and concludes this document with directions for future research. Finally, Appendix A gives the complete data sets discussed in Chapter 5, and Appendix B documents the C code used to empirically evaluate RNIC.

Chapter 2

Background

Constraint Satisfaction Problems (CSPs) can be used to model a wide range of problems. One type of problems are scheduling problems, such as the assignment of teaching assistants to courses [Lim, 2006]. Another use is in constraint database [Revesz, 2001]. Puzzle games, such as Sudoku[1], Minesweeper[2], and the Game of Set[3] can all be modeled and solved as CSPs. These are just a few examples of the uses of CSPs, but almost any problem can be modeled as a CSP.

In this chapter, we review some background information about Constraint Satisfaction problems (CSPs) and their different representations. We present two graph reformulations. We review common domain and relational consistency properties, and introduce how to compare different properties. And finally, we give related work.

[1] http://sudoku.unl.edu

[2] http://minesweeper.unl.edu

[3] http://gameofset.unl.edu

2.1 Constraint Satisfaction Problem (CSP)

Definition 1 ***(Constraint Satisfaction Problem)*** *A Constraint Satisfaction Problem (CSP) is given by a tuple $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, defined as follows:*

- $\mathcal{V} = \{V_1, V_2, \ldots, V_n\}$ *is a set of variables.*
- $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$ *is a set of domains. Each variable $V_i \in \mathcal{V}$ has a finite domain $D_i \in \mathcal{D}$.*
- $\mathcal{C} = \{C_1, C_2, \ldots, C_e\}$ *is a set of constraints, which constrains the possible assignment of values to variables.*

Each constraint $C_i \in \mathcal{C}$ is specified by a relation R_i defined on a subset of the variables, called the scope of the relation and denoted $scope(R_i)$, and the *arity* of the constraint is the size of the scope. Given a relation R_i, a tuple $\tau_i \in R_i$ is a vector of allowed values for the variables in the scope of R_i. Solving a CSP corresponds to finding an assignment of a value to each variable such that all the constraints are satisfied. The task can be to determine if a solution exists, find one solution, or find all solutions.

A constraint that has arity equal to two is called a *binary* constraint and a constraint that has arity equal to one is called a *unary* constraint. A binary CSP is a CSP where all of the constraints are binary or unary. In a non-binary CSP, the constraints can be of any arity. The constraint density of a binary CSP is equal to $\frac{2e}{n(n-1)}$, where e is the number of constraints and n the number of variables.

An example of a binary CSP is the graph coloring problem, such as the one illustrated in Figure 2.1 [Dechter, 2003]. The formulation of this CSP:

- $V = \{X_1, X_2, X_3, X_4, X_5, X_6\}$
- $D_{X_1} = D_{X_2} = D_{X_3} = D_{X_4} = D_{X_5} = D_{X_6} = \{red, blue, green\}$

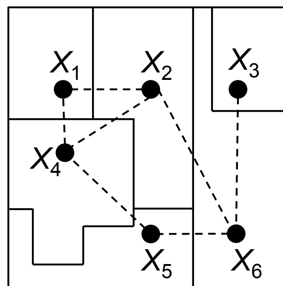


Figure 2.1: A graph coloring problem [Dechter, 2003].

- A constraint for each dotted line, which represents that two variables cannot share the same value.

One possible solution is when $X_1 = red, X_2 = blue, X_3 = red, X_4 = green, X_5 = red, X_6 = green$.

An example of a non-binary CSP is an crypto-arithmetic puzzles, such as SEND + MORE = MONEY [Dechter, 2003]. In the crypto-arithmetic puzzle, each letter represents a different digit such that the arithmetic equation is satisfied. Figure 2.2 shows an CSP for SEND + MORE = MONEY. The formulation of this CSP:

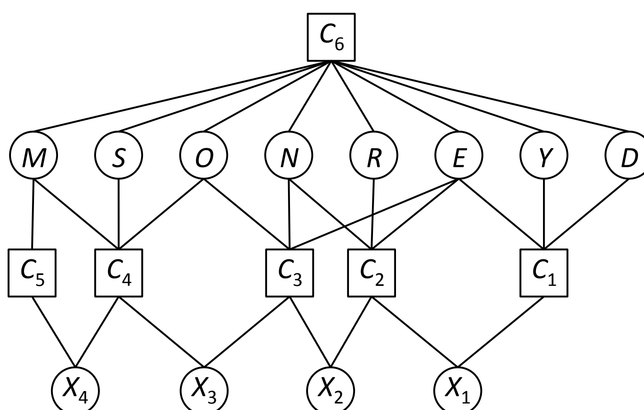


Figure 2.2: The CSP of the crypto-arithmetic puzzle SEND + MORE = MONEY [Dechter, 2003].

- $V = \{D, E, M, N, O, R, S, Y, X_1, X_2, X_3, X_4\}$, where X_1 is the carry from adding D and E, X_2 is the carry from adding N and R, etc.

- The domains of the carries are: $D_{X_1} = D_{X_2} = D_{X_3} = D_{X_4} = \{0, 1\}$. The domains for the letters are: $D_D = D_E = D_M = D_N = D_O = D_R = D_S = D_Y = \{0, 1, \ldots, 9\}$.

- $C = \{C_1, C_2, C_3, C_4, C_5, C_6\}$:

$$
\begin{aligned}
C_1 &: \quad Y + 10X_1 = D + E \\
C_2 &: \quad E + 10X_2 = X_1 + N + R \\
C_3 &: \quad N + 10X_3 = X_2 + O + E \\
C_4 &: \quad O + 10X_4 = X_3 + S + M \\
C_5 &: \quad M = X_4 \\
C_6 &: \quad D \neq E \neq M \neq N \neq O \neq R \neq S \neq Y
\end{aligned}
$$

The constraint $C_{i \in [1,5]}$, ensures the addition at the ith place in the equation. The constraint C_6, ensures that none of the digits are repeated.

One possible solution is when $D = 7, E = 5, M = 1, N = 6, O = 0, R = 8S = 9, Y = 2$, then SEND=9567, MORE=1085, and MONEY=10652.

2.2 Graphical Representations

There are multiple ways to graphically represent a CSP. The first way is representing the variables of the CSP as vertices, and the constraints as edges. An alternative way of representing a CSP is by the dual encoding.

2.2.1 Constraint graph, hypergraph, and primal graph

A binary CSP is represented by its *constraint graph* where the vertices are the variables of the CSP and the edges represent the constraints. A non-binary CSP is similarly represented by its *hypergraph* where the hyperedges represent the non-binary constraints. Figure 2.3 illustrates the hypergraph of a small non-binary CSP where $mathcalV = \{A, \ldots, F\}$ and the relations are $R_1, \ldots, R_6$.

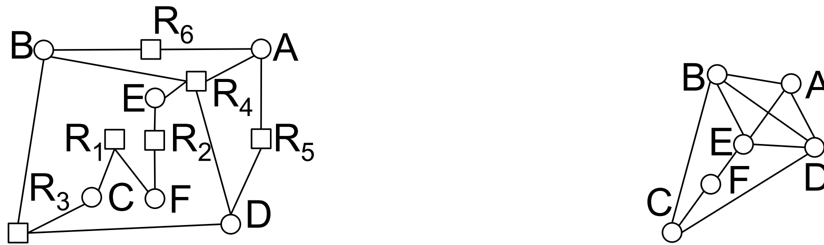


Figure 2.3: The hypergraph of a small CSP. Figure 2.4: The primal graph of a small CSP.

Another graphical representation of a non-binary CSP is the *primal graph* where the vertices are the CSP variables and edges connect every two vertices corresponding to variables in the scope of a relation [Dechter, 2003]. Figure 2.4 illustrates the primal graph of the same small non-binary CSP. Neigh(V_i) denotes the set of variables that are adjacent to V_i in the constraint graph of a binary CSP and the primal graph of a non-binary CSP. For the example of Figures 2.3 and 2.4, the neighborhoods of each variable is listed below:

1. Neigh$(A) = \{B, D, E\}$.
2. Neigh$(B) = \{A, C, E, D\}$.
3. Neigh$(C) = \{B, D, F\}$.
4. Neigh$(D) = \{A, B, C, E\}$.
5. Neigh$(E) = \{A, B, D, F\}$.

6. Neigh(F) = $\{C, E\}$.

2.2.2 Dual encoding

The dual encoding of a CSP $\mathcal{P}$, denoted $\mathcal{P}^{\mathcal{D}}$, is a binary CSP whose variables are the relations of $\mathcal{P}$, their domains are the tuples of those relations, and the constraints enforce *equalities* over the shared variables. The representation as a graph of this encoding is the *dual graph* of the CSP. Figure 2.5 illustrates the dual graph of the same small non-binary CSP. Neigh(R_i) denotes the set of relations adjacent to a

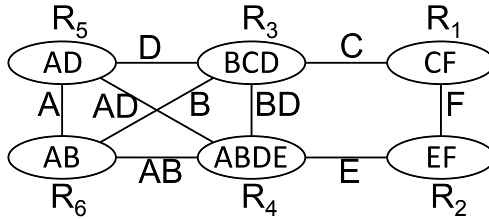


Figure 2.5: The dual graph of a small CSP.

relation R_i in the dual graph. For the example of Figure 2.5, the neighborhood of each relation is listed below:

1. Neigh(R_1) = $\{R_2, R_3\}$.

2. Neigh(R_2) = $\{R_1, R_4\}$.

3. Neigh(R_3) = $\{R_1, R_4, R_5, R_6\}$.

4. Neigh(R_4) = $\{R_2, R_3, R_5, R_6\}$.

5. Neigh(R_5) = $\{R_3, R_4, R_6\}$.

6. Neigh(R_6) = $\{R_3, R_4, R_5\}$.

2.3 Reformulation Strategies

Reformulation techniques are designed to automatically change the problem encoding into an 'easier' problem. 'Easier' can be in one of two meanings:

1. The reformulated problem asymptotic running time is lower than the original problem, or
2. The reformulated problem has advantageous properties to exploit to improve running time, but the worst-case complexity could remain the same.

Further, these 'easier' problems may change the set of solutions to the original problem, either by removing or adding solutions. The two reformulations discussed below, triangulation and redundancy removal, are used in our work to exploit properties they introduce and do not affect the set of solutions.

2.3.1 Triangulation

Graph triangulation adds an edge (a chord) between two non-adjacent vertices in every cycle of length four or more [Golumbic, 2004]. While minimizing the number of edges added by the triangulation process is NP-hard, MinFill is an efficient heuristic commonly used for this purpose [Kjærulff, 1990; Dechter, 2003]. Roughly, MinFill operates by determining, for each vertex, the number of edges needed to fully connect its parents (e.g., number of fill edges), and selects the vertex with the minimum number of fill edges, and connects all of its parents. It then repeats until all the vertices have been selected.

Notice, the dual graph of Figure 2.5 has a cycle of length four (R_1, R_2, R_3, and R_4). One possible triangulation of the dual graph would be to add an edge from R_1 to R_4, as illustrated in Figure 2.6. This triangulation is not unique, and the edges

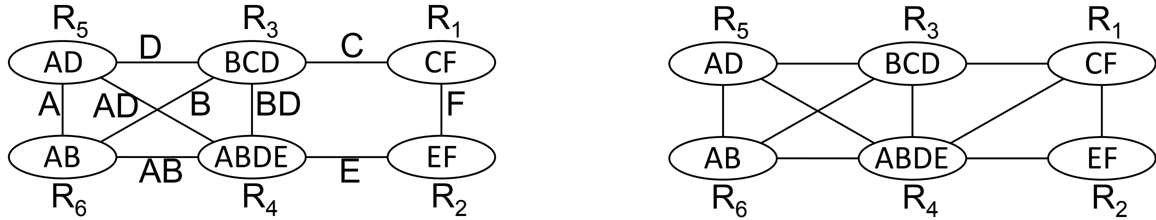


Figure 2.6: Triangulating a dual graph.

added depends on the heuristic used.

A perfect elimination ordering on a graph, is an ordering of the vertices such that, for each vertex v, v and the neighbors of v that occur after v in the ordering form a clique. If a graph is triangulated, then it is guaranteed to have a perfect elimination ordering [Fulkerson and Gross, 1965].

A tree decomposition of a CSP is a encoding of the constraint network [Dechter, 2003]. The tree decomposition is defined by a triple $\langle T, \chi, \psi \rangle$ of a CSP $P = (X, D, C)$, where $T = (V, E)$ is a tree, and for each vertex $v \in V$, χ is the variable labeling function, $\chi(v) \subseteq X$, and ψ is the relation labeling function, $\psi(v) \subseteq C$. These labeling functions determine which CSP variables and constraints appear in which nodes of the tree. The tree nodes are thus *clusters* of variables and constraints. A tree decomposition must satisfy two conditions:

1. Each constraints $c \in C$ appears in at least one node $v \in V$ in the tree where all of its variables are in that vertex, $scope(c) \subseteq \chi(v)$.

2. (Connectedness property) All the vertices where a variable $x \in X$ appears, $\{v \in V | x \in chi(v)\}$, induces a subtree of T.

2.3.2 Redundancy removal

An edge between two vertices in the dual graph is *redundant* if there exists an alternate path between the two vertices such that the shared variables appear in every vertex in the path [Janssen *et al.*, 1989; Dechter, 2003]. Janssen et al. [1989] introduced an efficient algorithm for computing the *minimal dual graph* by removing redundant edges. Many minimal graphs may exist, but all are guaranteed to have the same number of edges. Notice again, that Figure 2.5 has redundant edges. Figure 2.7 shows the dual graph and a minimal dual graph. Notice that in the original dual

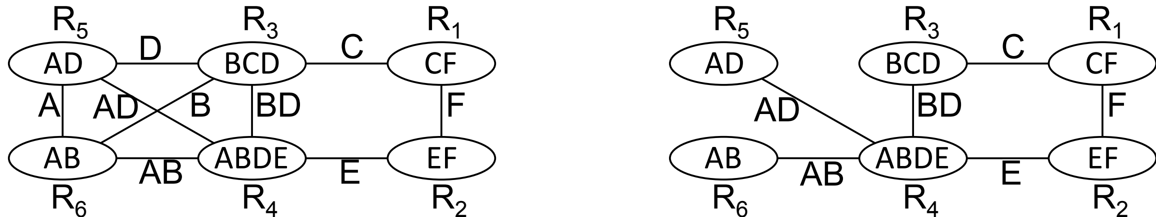


Figure 2.7: A minimal dual graph.

graph, there is a path between R_4, R_5, and R_6 where A is shared between all three. Therefore, the edge between R_5 and R_6 is redundant. The same can be discovered with B between R_3, R_4, and R_6, and D between R_3, R_4, and R_5.

2.4 Consistency: Properties & Algorithms

CSPs are in general $\mathcal{NP}$-complete and solved by search. Because the task is to determine if a solution exists, find one solution, or find all solutions, a sound and complete search must be used. Backtrack search (BT) is a sound and complete algorithm to solve CSPs.

To reduce the severity of the combinatorial explosion of search, CSPs are usually 'filtered' by enforcing a given local consistency property [Bessiere, 2006]. There are

two types of local consistency properties that we will focus on, domain consistency properties and relational consistency properties. The reader must be careful when reading not to confuse the consistency property and the algorithm to enforce the consistency property. Many times, the algorithms to enforce are named the same as the consistency property. A consistency property is a property that the CSP either has or does not have. While, the algorithm to enforce the consistency property will manipulate the CSP to have the property after running the algorithm. There can also be several algorithms to enforce the same consistency property.

2.4.1 Domain consistency properties

There are many local consistency properties that operates on the constraint graph or hypergraph. The algorithms for enforcing them typically operate by filtering the domains of variables. One common such property is Generalized Arc Consistency (GAC).

Definition 2 ***Generalized Arc Consistency*** [Mohr and Masini, 1988]*: A CSP is GAC iff, for every relation, any value in the domain of any variable in the scope of the relation can be extended to a tuple satisfying the relation.*

Another such local consistency property is Neighborhood Inverse Consistency (NIC), introduced in [Freuder and Elfe, 1996].

Definition 3 ***Neighborhood Inverse Consistency*** [Freuder and Elfe, 1996]*: A CSP is NIC iff, for every variable, any value in the domain of the variable can be extended to a partial solution in the subproblem induced by the variable and the variables in its neighborhood.*

Our work extends the local consistency property known as Neighborhood Inverse Consistency (NIC) introduced by Freuder and Elfe in [Freuder and Elfe, 1996] to relational filtering.

2.4.2 Relational consistency properties

Other local consistency properties operate on the dual graph of the CSP. The algorithms for enforcing them typically operate by filtering the constraint definition. These consistency properties have not been investigated as much.

In [1997], Dechter and van Beek introduced relational m-consistency.

Definition 4 ***Relational*** m***-Consistency*** [Dechter and van Beek, 1997]*: A CSP is relational* m*-consistency iff, for every set of* m *relations, whose scope is* $s = \cup_{i \in m} scope(R_i)$*, then every consistent partial solution of length* $|s| - 1$ *can be extended to a consistent partial solution of length* $|s|$*.*

Relational m-consistency requires adding new constraints to the CSP.

In [2010], Karakashian *et al.* introduced the property R($*$,m)C with $m \geq 2$. Unlike relational m-consistency, R($*$,m)C does not require adding new constraints to the CSP and is the first work on relational consistency that filters existing constraints.

Definition 5 ***R(*$*$*,*m*)C*** [Karakashian *et al.*, 2010]*: A CSP is R(*$*$*,*m*)C iff, for every relation, every tuple in the relation can be extended in a consistent assignment to every combination of* $m - 1$ *relations in the problem.*

Pairwise consistency [Janssen *et al.*, 1989] is equivalent to R($*$,2)C [Karakashian *et al.*, 2010].

2.4.3 Comparing consistency properties

In order to compare the various consistency properties discussed in this document we use the terminology introduced by Debruyne and Bessière in [1997]. Given two consistency properties p and p',

- p is *stronger* than p' if, in any CSP where p holds, p' also holds.
- p is *strictly stronger* than p' if p is stronger than p' and there exists at least one CSP in which p' holds but p does not.
- p and p' are *equivalent* when p is stronger than p' and vice versa.
- Finally, p and p' are *incomparable* when there exists at least one CSP in which p holds but p' does not, and vice versa.

In practice, when a consistency property is stronger (respectively, weaker) than another, enforcing the former never yields less (respectively, more) pruning than enforcing the latter on the same problem.

2.5 Solving CSPs

A CSP can be solved by search. A simple search procedure is *backtrack search*, which is a systematic, exhaustive exploration of the search space, which is made of all possible combinations of assignments of values to variables. Backtrack search explores the search space in a depth-first manner so that the space requirement remains linear in the number of variables in the CSP.

The ordering of variables during search is known to drastically affect the performance of the search process. The most constrained variables are commonly instantiated first in order to reduce the branching factor of the search tree. Many different

heuristics for variable ordering exit as an implementation of this general principle. Other variable-ordering heuristics exploit the structure of the CSP, such as the width of the constraint graph [Freuder, 1982], its induced width [Dechter and Pearl, 1987a], or its bandwidth [Zabih, 1990].

To reduce the size of the search space, we enforce a consistency property on the instance in a *pre-processing step* before running search. Alternatively, we typically interleave backtrack search with constraint propagation, in what is called a *lookahead schema*. More specifically, whenever a variable is instantiated during search, the effects of this assignment are propagated over the uninstantiated variables by removing from their domains values that do not agree with this new assignment. Lookahead schemas can either be *partial* (e.g., forward checking updates only the variables adjacent to the instantiated variable) or *full*, which enforces a given consistency property on uninstantiated variables.

2.6 Related Work

Consistency properties and their algorithms are central to CP, and perhaps best distinguish this discipline from other fields that study the same problems. Research has focused on:

- Defining new properties,
- Proposing new algorithms,
- Improving the performance of known ones, and
- Theoretically characterizing the relationship between the consistency level and the tractability of the CSP.

We first discuss the work done on binary CSPs, then on non-binary CSPs.

2.6.1 For binary CSPs

NIC was proposed by Freuder and Elfe in [1996] and evaluated by them and others on binary CSPs. Debruyne and Bessière [2001] showed that NIC is ineffective on sparse graph and too costly on dense graphs.

2.6.2 For non-binary CSPs

Bacchus *et al.* [2002] denotes *nic*(*dual*) for applying NIC to the dual encoding of a CSP. As stated in the introduction, it is identical to RNIC. However, the paper does not go beyond stating that *nic*(*dual*) is strictly stronger than *ac*(*dual*) (i.e., RNIC is strictly stronger than R($*$,2)C). More generally, relational consistency properties were formalized by Dechter and van Beek in [1997] as *relational m-consistency* and *relational (i, m)-consistency*. Enforcing those properties may require adding constraints to the problem, modifying its topology.

Most of the research on consistency for non-binary CSPs has focused on filtering the variables' domains, such as the study of 'variable-based' NIC [Gent *et al.*, 2000; Stergiou, 2007]. In contrast, our study focuses on the filtering of the relations (i.e., the constraints' definitions). As for relation-filtering properties, m-wise consistency was proposed in relational databases [Gyssens, 1986]. Janssen *et al.* [1989] showed that arc consistency on the dual encoding of a CSP enforces pairwise consistency. Algorithms for R($*$,m)C, which is equivalent to m-wise consistency, were proposed for arbitrary $m \geq 2$ and evaluated by Karakashian *et al.* in [2010]. One limitation of the algorithm for R($*$,m)C is the need to manually select m and generate all combinations of m relations that form a connected graph. The number of combinations grows exponentially with m, causing space limitations. In comparison, RNIC requires storing for each relation R a unique combination of constraints $\{R\} \cup \text{Neigh}(R)$ and

the size of this combination varies with the connectivity of R in the dual graph. Given the space requirement for storing all combinations of m relations, Karakashian *et al.* [2010] proposed to enforce R($*$,m)C on minimal dual graphs only, namely wR($*$,2)C, wR($*$,3)C, and wR($*$,4)C. The support structures used in ProcessQ (Algorithm 1 in Section 4.1) are similar to those proposed in by Bessière *et al.* in [2005].

Finally, the insight that breaking cycles yields trees in a search space (i.e., tree, or dangle, identification in SearchSupport, Section 4.1) can be related to the Cycle-Cutset method [Dechter and Pearl, 1987b].

Summary

In this chapter, we gave background information on CSPs. We also described two reformulation strategies of the dual graph of a CSP: by triangulation and redundancy removal. We introduced some common consistency properties and reviewed how they can be compared. Finally, we stated connections to prior work.

Chapter 3

Relational Neighborhood Inverse Consistency & Dual Graphs

The algorithm for enforcing NIC on CSPs of [Freuder and Elfe, 1996] was tested on binary CSPs in a preprocessing step to backtrack search on instances whose constraint density did not exceed 4.25%. Despite its pruning power and light space overhead, NIC received relatively little attention in the literature, likely because of the prohibitive cost of the algorithm for enforcing it. Below, we introduce RNIC as a generalization of NIC and characterize this new property in terms of other known consistency properties. Indeed, the former is a property that applies to the tuples of the relations of the CSP, while the latter applies to the values in the variables' domains (which are, in fact, unary relations). We then compare RNIC with R($*$,m)C and domain filter properties. We also investigate the structure of binary CSPs, and the effect of enforcing RNIC on binary CSPs.

3.1 Defining RNIC

Definition 6 *A relation R_i is said to be* RNIC *iff every tuple in R_i can be extended to the variables in $\bigcup_{R_j \in \text{Neigh}(R_i)} scope(R_j) \setminus scope(R_i)$ in an assignment that simultaneously satisfies all the relations in* Neigh(R_i). *A network is* RNIC *iff every relation is* RNIC.

Informally, every tuple τ_i in every relation R_i can be extended to a tuple τ_j in each $R_j \in \text{Neigh}(R)$ such that together all those tuples are consistent with all the relations in Neigh(R_i). Like R($*$,m)C, RNIC can be enforced by filtering the existing relations and without introducing any new relations to the CSP. A straightforward algorithm for enforcing RNIC applies the following operation to every relation R_i in the problem until quiescence:

$$R_i \leftarrow \pi_{scope(R_i)}(\bowtie_{R_j \in \{R_i\} \cup \text{Neigh}(R_i)} R_j) \tag{3.1}$$

where π and $\bowtie$ are the relational operators project and join, respectively. The space requirement of this algorithm is prohibitive in practice because it requires storing the join of $R_i \cup \text{Neigh}(R_i)$, which is not necessary as we argue in Chapter 4. For the example of Figure 3.1, RNIC examines the six subproblems induced on the dual

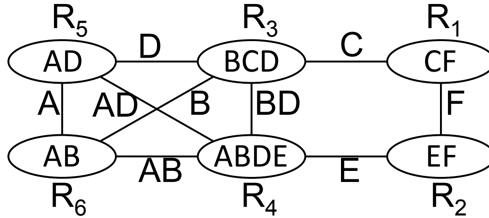


Figure 3.1: The dual graph of a small CSP.

graph by each relation and its neighborhood as listed below:

1. For R_1, Neigh$(R_1) = \{R_2, R_3\}$.

2. For R_2, Neigh$(R_2) = \{R_1, R_4\}$.

3. For R_3, Neigh(R_3) = $\{R_1, R_4, R_5, R_6\}$.

4. For R_4, Neigh(R_4) = $\{R_2, R_3, R_5, R_6\}$.

5. For R_5, Neigh(R_5) = $\{R_3, R_4, R_6\}$.

6. For R_6, Neigh(R_6) = $\{R_3, R_4, R_5\}$.

Generally speaking, the number of induced subproblems to be considered is equal to e, where e is the number of relations in the CSP; and the size of the largest subproblem is equal to $\delta + 1$, where δ is the degree of the dual graph.

3.2 Comparing RNIC and R($*$,m)C

Karakashian *et al.* in [2010] introduced the property R($*$,m)C with $m \geq 2$, which ensures that every tuple in every relation can be extended in a consistent assignment to every combination of $m-1$ relations in the problem. For the example shown in Figure 3.1, R($*$,3)C must verified on 12 combinations of two relations:

1. $\{R_1, R_2, R_3\}$
2. $\{R_1, R_2, R_4\}$
3. $\{R_1, R_3, R_4\}$
4. $\{R_1, R_3, R_5\}$
5. $\{R_1, R_3, R_6\}$
6. $\{R_2, R_3, R_4\}$
7. $\{R_2, R_4, R_5\}$
8. $\{R_2, R_4, R_6\}$
9. $\{R_3, R_4, R_5\}$
10. $\{R_3, R_4, R_6\}$
11. $\{R_3, R_5, R_6\}$
12. $\{R_4, R_5, R_6\}$

Generally speaking, the number of induced subproblems to be considered is $\mathcal{O}(e^m)$, and the size of the largest subproblem is equal to m. We compare RNIC with R($*$,m)C, which is defined for $m \geq 2$.

Theorem 1 *RNIC is strictly stronger than R($*$,m)C, $m \leq 3$.*

Sketch of proof: For a relation R_i, RNIC requires that each tuple of R_i and at least one tuple from each of the relations in Neigh(R_i) be consistent, all together. R($*$,2)C

requires that the tuple of R_i be consistent with some tuple in each of the relations in Neigh(R_i), taken in separation. Thus, RNIC is strictly stronger than R($*$,2)C. For R($*$,3)C, at least one relation in each combination of three relations is such that its neighborhood encompasses at least the other two relations. Thus, RNIC is strictly stronger than R($*$,3)C. □

Theorem 2 *R($*$,m)C with $m \geq \delta + 1$, where δ is the degree of the dual graph, is strictly stronger than RNIC.*

Sketch of proof: When $m > \delta$, every set of relations considered by RNIC is a subset of at least one set of relations on which R($*$,m)C is enforced. □

Theorem 3 *For $4 \leq m \leq \delta$, R($*$,m)C and RNIC are not comparable.*

Sketch of proof: If a dual graph has a chain of relations of length between four and $\delta - 1$, R($*$,m)C for $4 \leq m \leq \delta$ can be stronger than RNIC. Conversely, if the dual graph is a star graph of five or more vertices, $S_{i>4}$, RNIC can be stronger than R($*$,m)C for $4 \leq m \leq \delta$. □

Figure 3.2 illustrates the above first three assertions. Two interesting structures

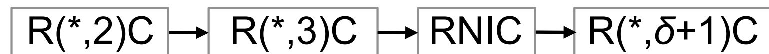


Figure 3.2: Comparing RNIC with R($*$,m)C.

of the dual graphs, trees and cycles, are such that several relational consistency properties collapse to R($*$,2)C, which is the weakest of them all:

Theorem 4 *RNIC, R($*$,2)C, and R($*$,m)C are equivalent on any dual graph that is tree structured or is a cycle of length $\geq maximum(4, m + 1)$.*

Proof: By straightforward generalization of Theorem 3. □

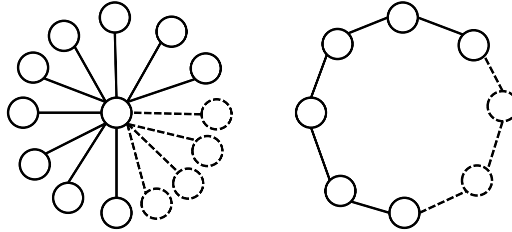


Figure 3.3: Configurations illustrating Theorem 4.

The theorem applies for a tree of any degree. As for the cycle, it must be length at least $m+1$ for $m \geq 3$. Figure 3.3 shows two such configurations. This last theorem is important because it identifies structural configurations where the relational consistency properties RNIC and R($*$,m)C collapse to their weakest version, that is R($*$,2)C. In Section 4.3 we propose reformulating the dual graph of the CSP to allow RNIC to overcome this obstacle.

3.3 Comparing RNIC and Domain Filtering

In practice, after enforcing RNIC on a CSP (by filtering the relations), the domains of the variables are updated accordingly in order to reduce the search effort. It is important to note that variable domains can be updated by simply projecting the filtered relations on the variables. Interestingly, these domain reductions do not break the RNIC property.

Theorem 5 *If a network is RNIC, domain filtering by GAC cannot enable further constraint filtering by RNIC.*

Proof: Similar to proof of Theorem 1 in [Karakashian *et al.*, 2010]. □

Following the terminology of [Bessière *et al.*, 2008], the property of a CSP where RNIC holds and where the domains agree with the constraints is denoted RNIC+GAC. Although formally correct, we find this notation confusing because it may incorrectly

suggest the need to enforce GAC, which is in general more expensive than (simply and without looping) projecting the relations on the domains. For that reason, we choose to denote this property instead RNIC+DF (i.e., RNIC followed by domain filtering).

The *singleton* variation of a given consistency property guarantees that the assignment of every value in the domain of a variable yields a CSP where the consistency property holds [Debruyne and Bessière, 2001]. Singleton consistencies have been studied mainly for arc consistency (SAC) and generalized arc consistency (SGAC).

Theorem 6 *SGAC on a non-binary CSP and RNIC+DF on the corresponding dual graph are not comparable.*

Proof: In Figure 3.4, the CSP is RNIC+DF but not SGAC. SGAC empties all

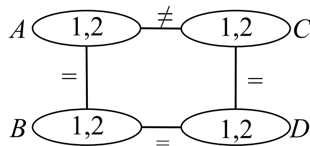
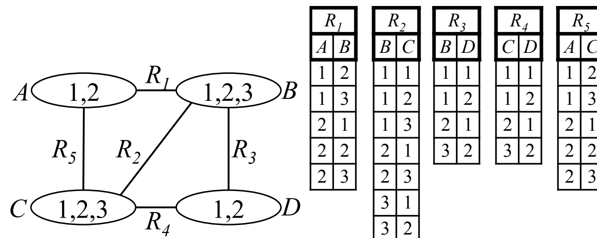


Figure 3.4: The CSP is RNIC+DF but not SGAC.



R1	
A	B
1	2
1	3
2	1
2	2
2	3

R2	
B	C
1	1
1	2
1	3
2	1
2	3
3	1
3	2

R3	
B	D
1	1
1	2
2	1
3	2

R4	
C	D
1	1
1	2
2	1
3	2

R5	
A	C
1	2
1	3
2	1
2	2
2	3

Figure 3.5: The CSP is SGAC but not RNIC+DF.

variables domains. In Figure 3.5, taken from [Debruyne and Bessière, 2001], the CSP is SGAC but not RNIC+DF. RNIC removes $\{(2,3),(3,2)\}$ from R_2, $\{(1,2),(1,3)\}$ from R_1, and $\{(1,2),(1,3)\}$ from R_5. Therefore, RNIC+DF removes the value 1 from A. □

Figure 3.6 shows the relationships between the domain-filtering properties discussed above.

Figure 3.6: Some domain filtering properties.

3.4 Structure of the Dual Graph of a Binary CSP

We first discuss the case of a binary CSP whose constraint graph is complete, then the case of a binary CSP whose constraint graph is not complete.

3.4.1 Binary CSP with a complete constraint graph

Theorem 7 *The $\frac{n(n-1)}{2}$ vertices of the dual graph of a binary CSP of n variables whose constraint graph is complete such as the one shown in Figure 3.7 (i.e., forms a clique of n vertices, K_n), can be arranged in an $(n-1) \times (n-1)$ triangle-shaped grid where:*

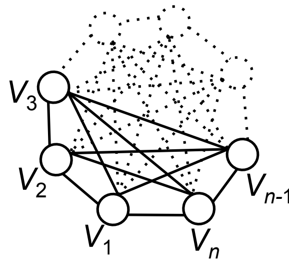


Figure 3.7: A complete constraint graph of n vertices.

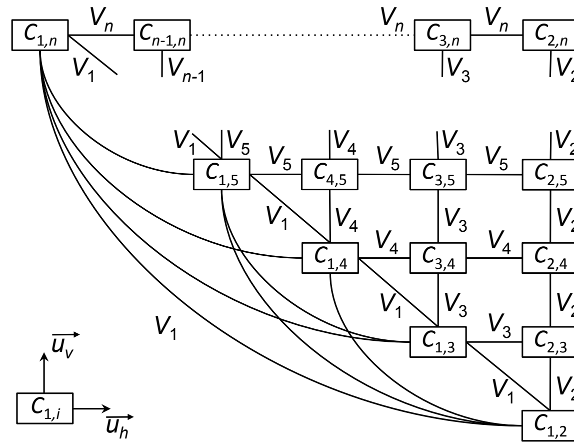


Figure 3.8: The dual graph corresponding to the CSP in Figure 3.7.

1. *The $n-1$ vertices on the diagonal of the triangle correspond to the constraints over the variable V_1. They are denoted $C_{1,i}$ where $i \in [2, n]$ and completely*

connected. The connecting edges are labeled with V_1.

2. *The $n-1$ vertices corresponding to the constraints over variable $V_{i\geq 2}$ are located along the path in the grid shown in Figure 3.9 and specified as follows:*

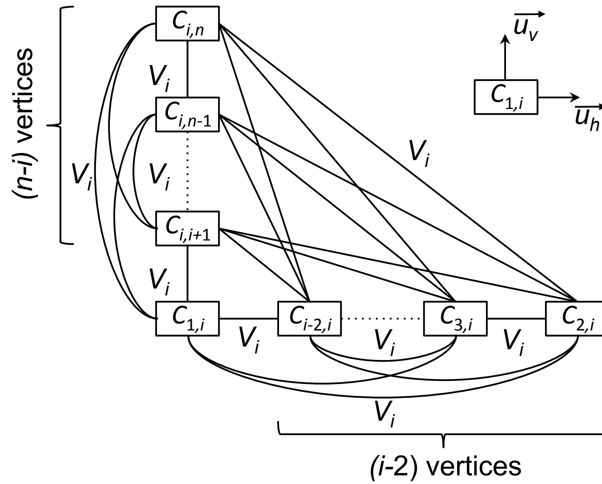


Figure 3.9: The path for the constraints over variables $V_{i\geq 2}$ of the grid of Figure 3.8.

- *Considering the coordinate system defined by the horizontal and vertical unit vectors $\vec{u_h}$, $\vec{u_v}$ and centered on $C_{1,i}$,*
- *$i-2$ vertices are lined up along the horizontal axis $\vec{u_h}$, and*
- *$n-i$ vertices are lined up along the vertical axis $\vec{v_h}$.*
- *Those $n-1$ vertices are completely connected, and the connecting edges are labeled with V_i. (For the sake of clarity, Figure 3.8 does not show all the edges of the dual graph: only all the edges labeled V_1 are shown on the diagonal of the grid.)*

Proof: (By induction of number of variables.)

Base Step: Stated for $n = 3$.

For $n = 3$, the constraint graph is shown in Figure 3.10 and the corresponding dual graph in Figure 3.11. The dual graph is obviously a triangle.

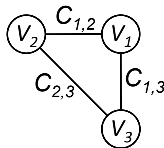


Figure 3.10: A complete constraint graph with 3 variables.

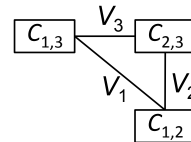


Figure 3.11: The dual graph of a complete constraint graph with 3 variables.

- The two vertices corresponding to the constraints over the variable V_1 form the diagonal.
- The two vertices corresponding to the constraints over V_2 start at $C_{1,2}$ and have 0 vertices along the horizontal axis, and one vertex along the vertical axis. Also, the two vertices corresponding to the constraints over V_3 start at $C_{1,3}$ have 0 vertices along the horizontal axis, and one vertex along the vertical axis.

Inductive Step: Assume that the theorem holds for a CSP with k variables (inductive hypothesis). We want to show the theorem holds for a CSP with $k+1$ variables (inductive step).

Consider the complete constraint graph of a CSP with k variables, which is the clique K_k, show in Figure 3.12. By the inductive hypothesis, the dual graph can be arranged in the triangle-shaped grid shown in Figure 3.13. Now, add the variable V_{k+1} to the CSP. In order to connect V_{k+1} to all k variables, k constraints are added to the constraint graph of the CSP, as shown in Figure 3.14. Namely, these k constraints are $C_{i,k+1}, \forall i \leq k$. Place the dual variables as follows, going from right to left in Figure 3.15:

- $C_{i,k+1}, i \in [2, k-1]$ is placed above $C_{i,k}$,

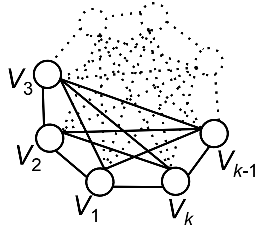


Figure 3.12: A complete constraint graph with k variables.

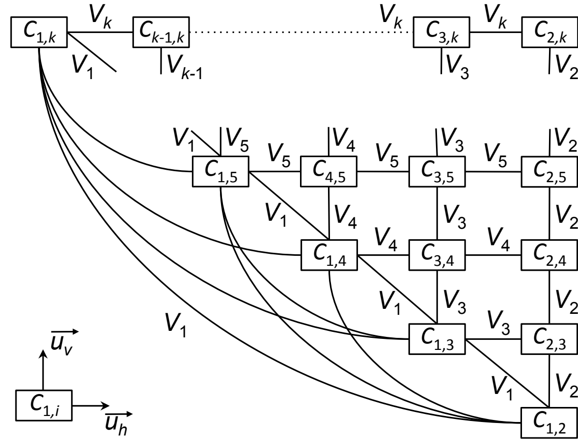


Figure 3.13: The dual graph of a complete constraint graph with k variables.

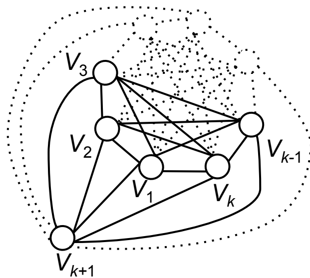


Figure 3.14: A complete constraint graph with $k+1$ variables.

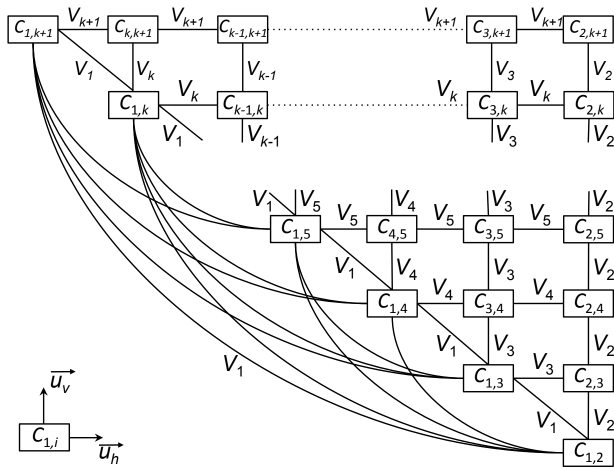


Figure 3.15: The dual graph of a complete constraint graph with $k+1$ variables.

- $C_{k,k+1}$ is placed above $C_{1,k}$, and
- $C_{k+1,1}$ is placed to the left of $C_{k,k+1}$.

This arrangement yields a dual graph that is a triangle-shaped grid because:

- The vertices corresponding to the constraints over the variable V_1 are located on the diagonal of the triangle because $C_{k+1,1}$ is to the left of $C_{k+1,k}$,

- The coordinate system for centered on $C_{1,i\in[2,k]}$ increases by one vertical unit for vertex $C_{k+1,i}$ and labeled with variable V_i.

- The coordinate system for centered on $C_{1,k+1}$ has $(k+1)-2=k-1$ vertices on the horizontal axis and 0 vertices in the vertical axis. The k vertices on the top row of the triangle form a clique whose edges are labeled with V_{k+1} (shown partially, for readability).

Consequently, this new dual graph of a complete constraint graph of $k+1$ variables has the topology of a triangle-shaped grid. □

Corollary 1 *After the removal of redundant edges, the dual graph of a binary CSP of n variables whose constraint graph is complete can be arranged in a $(n-1)\times(n-1)$ triangle-shaped grid, where every CSP variable annotates the edges of a chain of length $n-2$.*

Proof: Let's consider the $n-1$ vertices corresponding to the constraints that apply on variable V_i and the coordinate system defined by the horizontal and vertical unit vectors $\vec{u_h}$, $\vec{u_v}$ and centered on $C_{1,i}$. All edges between the $i-2$ horizontal vertices and the $n-i$ vertical vertices that link two non-consecutive vertices are redundant and can be removed, leaving a path linking the $n-1$ vertices along the horizontal and vertical axis. As for V_1, a similar operation can be applied to the vertices along the diagonal of the triangle. □

Because redundancy removal is not unique, not all redundancy-free dual graphs necessarily yield a triangle-shaped grid as we show using a counter-example. One possible redundancy-free dual graph for the complete constraint graph of five vertices of Figure 3.16 is shown in Figure 3.17. In this example, there is a cycle of size six in

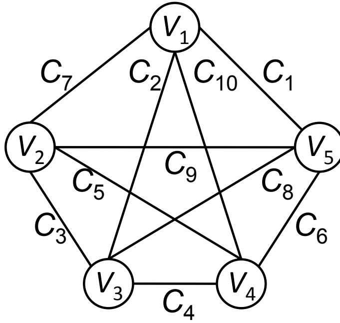


Figure 3.16: A complete constraint graph with 5 variables.

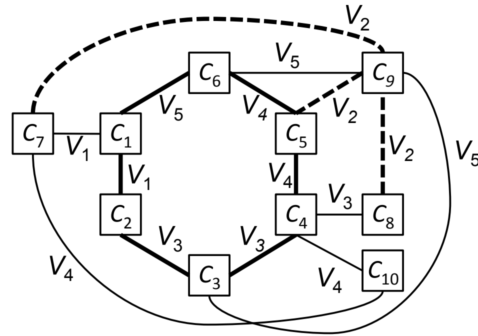


Figure 3.17: A redundancy-free dual graph of a complete constraint graph with five variables, which does not form a grid structure.

the dual graph, indicated by the bold lines in Figure 3.17. Thus, the dual graph is not a grid. Further, the variable V_2 does not annotate a chain, but a star, as indicated by the dotted lines in the dual graph.

3.4.2 Binary CSP with a non-complete constraint graph

In a binary CSP with a non-complete constraint graph, the dual graph can be thought of as the complete binary constraint graph with some missing vertices and edges. Because, in the dual graph of a complete constraint graph, all the vertices corresponding to the constraints that apply to a given CSP variable are completely connected, it is always possible to form a chain connecting those vertices that effectively appear in the dual graph. For example, consider the binary CSP with $n = 5$ variables given in Figure 3.18. A redundancy-free dual graph for that binary CSP is given in Figure 3.19, which was constructed from the dual graph for the complete CSP by removing the vertices corresponding to the constraints that are not in the CSP.

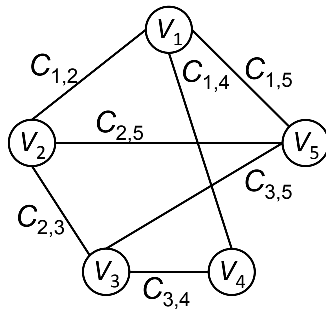


Figure 3.18: A complete constraint graph with 5 variables.

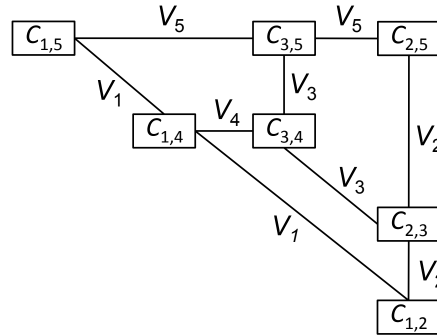


Figure 3.19: The redundancy-free dual graph of a the constraint graph with 5 variables.

3.5 RNIC on a Binary CSPs

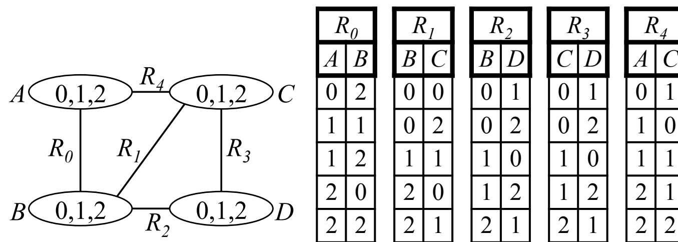
We first compare RNIC to NIC on binary CSPs, then discuss how the structure of the dual graph affects RNIC.

3.5.1 Comparing RNIC and NIC

The filter power of NIC and RNIC+DF are not comparable.

Theorem 8 *NIC (on a binary CSP) and RNIC+DF (on the dual graph of the same binary CSP) are not comparable.*

Proof: In Figure 3.20, the CSP is NIC but not RNIC+DF. RNIC removes the tuples



R_0		R_1		R_2		R_3		R_4	
A	B	B	C	B	D	C	D	A	C
0	2	0	0	0	1	0	1	0	1
1	1	0	2	0	2	0	2	1	0
1	2	1	1	1	0	1	0	1	1
2	0	2	0	1	2	1	2	2	1
2	2	2	1	2	1	2	1	2	2

Figure 3.20: The binary CSP is NIC but not RNIC+DF.

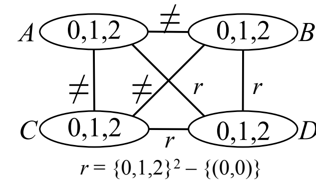


Figure 3.21: Binary CSP is RNIC+DF but not NIC.

in $\{(0,2),(2,2)\}$ from R_0, $\{(0,0),(1,2)\}$ from R_1, $\{(0,2)\}$ from R_2, $\{(0,2)\}$ from R_3,

and $\{(0,1),(2,1)\}$ from R_4. Therefore, RNIC+DF removes the value 0 from A. In Figure 3.21, the CSP is RNIC+DF but not NIC. NIC removes the value 0 from D. $\square$

Empirically, it was shown that enforcing RNIC+DF on random binary CSPs yeilds stronger filtering in almost all cases than NIC [Luchtel, 2011].

3.5.2 Effects of the dual-graph's structure on RNIC

The redundancy-free dual graph an arbitrary binary CSP can contain the following configurations:

1. A cycle of length four, on a grid-shaped dual graph
2. A cycle of length larger than four as shown in Figure 3.17.
3. A triangle along the diagonal.

On the first two cases above, RNIC is equivalent to 2-wise consistency by Theorem 4. On the third case, RNIC is equivalent to R(∗,3)C.

Theorem 9 *After the removal of redundant edges in the dual graph of a binary CSP, RNIC is never stronger than R(∗,3)C.*

Proof: (By Contradiction) Assume that, after redundancy removal, RNIC is stronger than R(∗,3)C, assume it to be R(∗,4)C. Therefore, there must be a configuration of the dual graph where a given constraint, C_1, has three adjacent constraints C_2, C_3, and C_4, and where C_1 is not an articulation point (otherwise, the argument must be applied recursively on the biconnected components). The only redundancy-free configuration is the one shown in Figure 3.22. We show that this configuration is not possible.

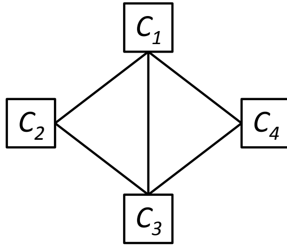


Figure 3.22: A redundancy-free configuration of four binary constraints.

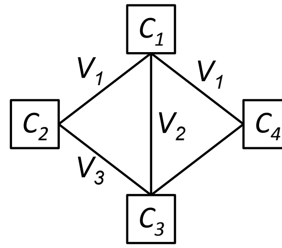


Figure 3.23: One possible labeling of the edges incident to C_1.

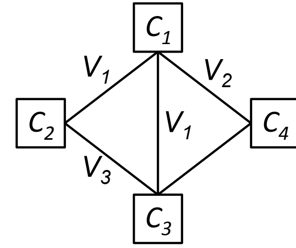


Figure 3.24: The other possible labeling of the edges incident to C_1.

1. Given the topology of the graph shown in Figure 3.22, the three edges incident to C_1 cannot have the same labeling, for example variable V_1, because C_1 becomes a unary constraint. They cannot three different labeling, for example variables V_1, V_2, and V_3, otherwise C_1 becomes a ternary constraint. Thus, they must be labeled with two variables, V_1 and V_2, as shown in Figures 3.23 and 3.24.

2. In Figure 3.23, the edge between C_2 and C_3 cannot be labeled V_1 (otherwise, C_2 becomes a unary constraint); cannot be labeled V_2 (otherwise, the scopes of C_2 and C_1 become equal, and we assume that the CSP is normalized); therefore, it must be labeled V_3. The edge between C_3 and C_4 cannot be labeled V_1 or V_4 (otherwise, C_3 becomes a ternary constraint); cannot be labeled V_2 (otherwise, the scopes of C_1 and C_3 become equal); cannot be labeled V_3 (otherwise, the scopes of C_2 and C_4 become equal). Therefore, no possible labeling for the edge between C_3 and C_4 exists, and this configuration is impossible.

3. In Figure 3.24, the edge between C_2 and C_3 cannot be labeled V_1 (otherwise, C_2 would be a unary constraint); cannot be labeled V_2 (otherwise, the scopes of C_1 and C_2 become equal); cannot be labeled V_3 (otherwise, the scopes of C_2 and C_3 become equal). Therefore, no possible labeling for the edge between C_2

and C_3 exist, and this configuration is impossible.

Consequently, no redundancy-free dual graph of a binary CSP can have a configuration of its vertices for enforcing R(∗,4)C. □

Using an algorithm for enforcing RNIC to enforce R(∗,2)C is wasteful of resources. Indeed, the former executes more consistency-checking operations than needed to enforce R(∗,2)C given that the neighborhoods considered by the former are supersets of those considered by the latter.

Summary

In this chapter, we introduced RNIC and theoretically compared RNIC to the some previously known local consistency techniques. We also discussed the structure of binary CSPs on the dual graph and RNIC's filtering power on binary CSPs.

Chapter 4

Enforcing RNIC

Below, we describe an algorithm for enforcing RNIC on a finite CSP, and analyze its complexity. The algorithm has two main components: ProcessQ (Algorithm 1) and SearchSupport. We also give three reformulating the dual graph, on which the algorithm works:

1. Removing redundant edges of the dual graph
2. Triangulating the dual graph
3. Triangulating the redundancy-free dual graph.

Further, we give a selection strategy for picking the dual graph to enforce RNIC on.

4.1 An Algorithm for RNIC

We define S_τ, the *support* of a tuple $\tau \in R$, to be the set of tuples that verify the condition: $\forall R' \in \text{Neigh}(R), \exists(\tau' \in R'), (\tau' \in S_\tau)$, and the tuples in $S_\tau \cup \{\tau\}$ agree on all shared variables. ProcessQ (Algorithm 1) enforces RNIC on a CSP $\mathcal{P}$ ensuring

that every tuple in every relation has a valid support. Note that the Neigh(R) is determined by the topology of the dual graph, which we will alter in Section 4.3.

ProcessQ operates on a queue of relations $\mathcal{Q}_R$ initialized with all the relations of $\mathcal{P}$. For each relation R of $\mathcal{P}$, we maintain a queue of tuples $\mathcal{Q}_t(R)$ initialized with all the tuples in R. The function SearchSupport(τ, R) computes S_τ as discussed below. The function Rel(τ) returns the relation to which τ belongs. The data structure *SupportedBy*(τ) maintains the list of tuples supported by τ.

ProcessQ removes from $\mathcal{Q}_R$ one relation R at a time. It iterates over the tuples of R stored in $\mathcal{Q}_t(R)$. For each tuple $\tau \in \mathcal{Q}_t(R)$, SearchSupport seeks a support for τ. When a support is not found, τ is removed from R, and all tuples τ_i supported by τ are added to the queue of their respective relations, and the corresponding relations added to $\mathcal{Q}_t$. Finally, τ is removed from $\mathcal{Q}_t(R)$. Whenever a relation is empty, ProcessQ halts and returns false indicating that $\mathcal{P}$ is not consistent. When $\mathcal{Q}_R$ is empty ProcessQ terminates successfully indicating that $\mathcal{P}$ is RNIC.

Algorithm 1: ProcessQ enforces RNIC

```
   Input: Q_R a queue of relations, {Q_t(R)} a set of queues of tuples, one for
          each relation
   Output: true if the problem is RNIC, false otherwise
1  while (Q_R ≠ ∅) do
2      R ← Pop(Q_R)
3      foreach τ ∈ Q_t(R) do
4          support ← SearchSupport(τ, R)
5          if support = false then
6              Delete(τ, R)
7              if R = ∅ then return false
8              forall τ_i ∈ SupportedBy(τ) do
9                  R_i ← Rel(τ_i)
10                 Q_t(R_i) ← Q_t(R_i) ∪ {τ_i}
11                 Q_R ← Q_R ∪ {R_i}
12         Q_t(R) ← Q_t(R) \ {τ}
13 return true
```

4.1.1 SearchSupport

SearchSupport(τ, R) operates by conducting a backtrack search on $\mathcal{P}_R^D$ the sub-problem induced by $\{R\} \cup \text{Neigh}(R)$ on the dual encoding of $\mathcal{P}$. The variables of $\mathcal{P}_R^D$ are the relations $\{R\} \cup \text{Neigh}(R)$. Their domains are the tuples of the relations except for the variable corresponding to R, which is assigned the tuple τ. A solution to $\mathcal{P}_R^D$ is $\{\tau\} \cup S_\tau$. The search stops at the first solution, or returns false if no solution is found. The process uses forward checking and dynamic variable ordering (domain/degree). Two major mechanisms significantly contributed to the success of this search process by improving its running time:

1. *The use of the index-tree data structure* to determine whether or not two tuples of two relations adjacent in the dual graph are consistent. This data structure was proposed in [Karakashian *et al.*, 2010].

2. *The dynamic identification, after each variable instantiation, of trees in the graph of uninstantiated variables.* The instantiation of a variable eliminates, from the problem, the variable and the constraints that link it to the uninstantiated variables, potentially breaking cycles in the graph and yielding trees. We call those trees *dangles*, and apply directional arc consistency on them to ensure that they are solvable. If they are, we isolate them from the search process. Otherwise, we force the search to backtrack. Dangle identification is linear in the number of vertices and edges. Its overhead, if any, was largely compensated by its benefits.

Note that dangle identification is a general mechanism for improving the performance of *any* backtrack search. Obviously, it cannot be used in the algorithm for enforcing GAC or R($*$,2)C (where there is no search). Further, it is not particularly

useful in the algorithm for enforcing R$(*, m)$C because the values of m are small in practice.

4.1.2 Complexity analysis

For our analysis, let d be the maximum domain size, k the maximum constraint arity, e the number of relations, and δ the degree of the dual graph. The maximum number of tuples t in a relation is bounded by $\mathcal{O}(d^k)$.

To find the support of a tuple, SearchSupport first verifies the validity of an existing support, then, if needed, it looks for a support by running a backtrack search on the subproblem induced by the relation and its neighbors.

Proposition 10 *The time complexity of* SearchSupport *is* $\mathcal{O}(t^\delta)$*.*

Proof: Verifying the validity of an existing support costs $\mathcal{O}(\delta)$. To build a support for a tuple, SearchSupport executes a backtrack search on a problem with $\delta + 1$ variables of maximum domain size t where the first variable is instantiated. The complexity of this search is $\mathcal{O}(t^\delta)$. □

Proposition 11 *The time complexity of* ProcessQ *is* $\mathcal{O}(t^{\delta+1}e\delta)$*.*

Proof: The outer loop (Line 1) iterates over the relations in $\mathcal{Q}_R$. This loop runs e times, the initial size of $\mathcal{Q}_R$, plus the number of times a relation is added to $\mathcal{Q}_R$ (Line 11). Given that a relation is adjacent to at most δ other relations, whenever a tuple is deleted, at most δ relations are added to $\mathcal{Q}_R$. There are $\mathcal{O}(te)$ tuples in $\mathcal{P}$ and each tuple is deleted at most once. Thus, Line 6 is executed $\mathcal{O}(te)$ times, each time enqueuing $\mathcal{O}(\delta)$ relations. Consequently, the outer loop (Line 1) runs $\mathcal{O}(te\delta)$ times.

The loop over the queued tuples (Line 3) executes $\mathcal{O}(t)$ times per relation. By Theorem 10, the complexity to find the support of a tuple is $\mathcal{O}(t^\delta)$. Thus, the time complexity of ProcessQ is $\mathcal{O}(t^{\delta+1}e\delta)$. □

The space complexity of ProcessQ is dominated by that of the data structures.

Theorem 12 *The space complexity of* ProcessQ *is* $\mathcal{O}(ket\delta)$*.*

Proof: Supports require $\mathcal{O}(et\delta)$ space. The index-trees require $\mathcal{O}(ket\delta)$ [Karakashian *et al.*, 2010]. □

The complexity of RNIC is dominated by ProcessQ, therefore, the time and space complexities of RNIC are $\mathcal{O}(t^{\delta+1}e\delta)$ and $\mathcal{O}(ket\delta)$, respectively. The time complexity of the obvious algorithm based on Expression (3.1) is $\mathcal{O}(t^{\delta+2}e\delta)$. When intermediate joins are not stored, its space complexity is $\mathcal{O}(t^{\delta+1})$, a major bottleneck for its practical implementation. Thus, ProcessQ saves on both time and space.

4.2 Enforcing RNIC versus R($*$,m)C

The above summarized algorithm and that for enforcing R($*$,m)C [Karakashian *et al.*, 2010] are similar in that they both try to 'complete' [Freuder, 1991] each tuple in each relation over one (or more) sets of relations.

The algorithm for R($*$,m)C considers *every* combination of m connected relations. The number of those combinations is $O(e^m)$. Further, each relation needs to be 'checked' against $m-1$ relations in *each* combination where it appears.

The algorithm for enforcing RNIC does not suffer from the above drawbacks. First, the number of combinations considered is equal to the number of relations (e), and each relation is 'checked' against a unique set of relations, which is determined by its neighborhood. Further, the size of the neighborhood is determined *locally* by

the connectivity of the relation in the dual graph. Thus, the 'level' of consistency enforced is not necessarily the same on all relations of the dual graph: Lower levels are enforced on sparser portions of the dual graph, and higher levels on the denser portions. In particular, on a cycle of length four or more, RNIC 'naturally' reduces to R(∗,2)C, see Theorem 4.

4.3 Reformulating the Dual Graph

In this section, we introduce two reformulations, and their combination, of the dual graph and their effects on the consistency property enforced by RNIC. An adaptive selection technique for selecting the dual graph to use is also introduced.

Two topological conditions of the dual graph can seriously hinder the performance of ProcessQ (Algorithm 1):

1. *High density of the dual graph.* As the density of the dual graph increases, the neighborhood of a given relation R_i grows, which increases the cost of enforcing RNIC. To address this issue, we reformulate the dual graph by removing redundant edges.

2. *The existence of cycles of length four or more.* On a cycle of length four or more, the two adjacent relations of a given relation R_i in the cycle are prevented from 'communicating,' thus reducing RNIC to R(∗,2)C (see Theorem 4). To address this issue, we propose to reformulate the dual graph by triangulation, which eliminates cycles of length four or more.

The above two reformulations have the following effects:

- Removing redundant edges cannot strengthen the consistency property enforced by the algorithm and cannot decrease the number of nodes visited by search.

- Adding edges by graph triangulation cannot weaken the consistency property enforced and cannot increase number of nodes visited by search.

Applying PROCESSQ on the dual graph reformulated by one or both of the above reformulations enforces three variations of RNIC, namely wRNIC, triRNIC, and wtriRNIC, where the prefixes 'w' and 'tri' denote the consistency properties resulting from removing redundant edges and triangulating the dual graph, respectively. Figure 4.1 illustrates those relationships in a partial order. Naturally, the property enforced

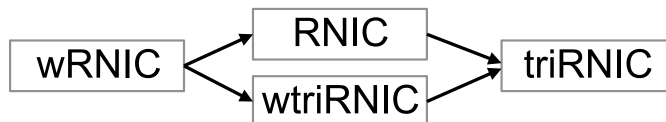


Figure 4.1: Variations of RNIC.

depends on the particular minimal and triangulated dual graph used.

While the *set of solutions to the CSP is not affected by either reformation*, it is not straightforward to predict the effect of the above reformulations on CPU time. To lay it out, we would like to remove enough edges from the dual graph to reduce the running time of PROCESSQ, which is $\mathcal{O}(t^{\delta+1}e\delta)$. However, we would also like to add enough edges to the dual graph in order to boost propagation. Furthermore, we need a strategy to automatically select the appropriate reformulation. Below, we discuss the two reformulations (Sections 4.3.1 and 4.3.2) and their combination (Section 4.3.3). In Section 4.3.4, we propose a procedure to automatically select a reformulation in a preprocessing step.

4.3.1 Removing redundant edges: wRNIC

An edge between two vertices in the dual graph is *redundant* if there exists an alternate path between the two vertices such that the shared variables appear in every vertex

in the path [Janssen *et al.*, 1989; Dechter, 2003]. Redundant edges can be removed without affecting the set of solutions of the CSP. Janssen et al. [1989] introduced an efficient algorithm for computing the *minimal dual graph* by removing redundant edges. Many minimal graphs may exist, but all are guaranteed to have the same number of edges. Figure 4.2 shows the dual graph (density 60%) and a minimal dual graph (density 40%) of the example of Figure 2.5. Note that R($*$,2)C $\equiv$ wR($*$,2)C

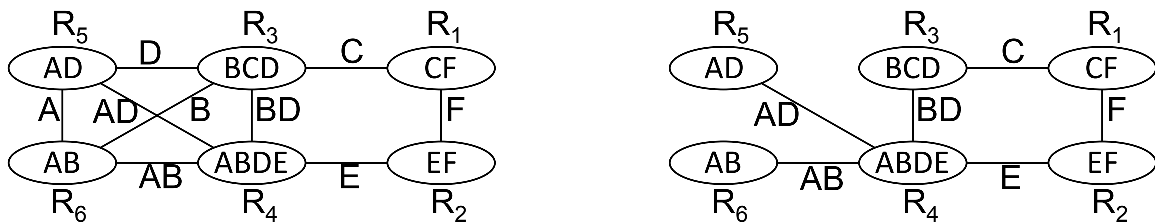


Figure 4.2: A minimal dual graph.

[Janssen *et al.*, 1989]. Also, computing and storing the combinations of relations necessary for enforcing R($*$,m)C is not possible in practice unless the redundant edges are first removed from the dual graph [Karakashian *et al.*, 2010].

Our experiments showed that RNIC is advantageous on dual graphs of density up to around 15%.[1] For higher density values, we propose to remove the redundant edges in the dual graph before running ProcessQ. This operation reduces the density of the original dual graph and the size of the induced subproblems on which Search-Support is executed. It also results in a weakened consistency, denoted wRNIC, that depends of the particular minimal graph used. Because wRNIC is enforced on a minimal dual graph (i.e., a graph with no more edges than the original dual graph), wRNIC is strictly weaker than wRNIC.

[1]In a related research, we studied the density of 1689 dual graphs of (binary and non-binary) CSPs from the Solver Competition Benchmarks. We identified a sharp threshold at 15% density. Indeed, 56.6% of the dual graphs (79.9% after redundancy removal) considered had a density less than or equal to 15%. It is not yet clear to us how to interpret the value of this threshold.

Figure 4.3 integrates the above discussion in the partial order of Figure 3.2. Note

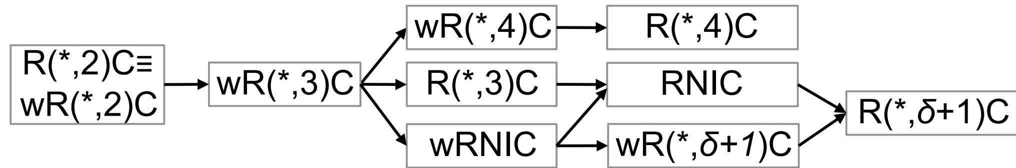


Figure 4.3: Relating RNIC, wRNIC, R($*$,m)C, and wR($*$,m)C.

that these results hold between the weakened properties provided they are enforced on the *same* minimal dual graph.

4.3.2 Triangulating the dual graph: triRNIC

When the dual graph has only cycles of size four or more, RNIC reduces to R($*$,2)C (see Theorem 4), which significantly hampers filtering and propagation. To remedy this situation, we propose to triangulate the dual graph. This process creates loops in the dual graph and increases the size of the induced subproblems on which SearchSupport is executed, boosting the propagation process, but also raising the consistency level enforced on the CSP. For example, in the dual graph of the example of Figure 2.5, Neigh(R_1) $=\{R_2, R_3\}$. However, Neigh(R_1)$=\{R_2, R_3, R_4\}$ in the triangulated graph (density 67%) of Figure 4.4. We denote the resulting consistency

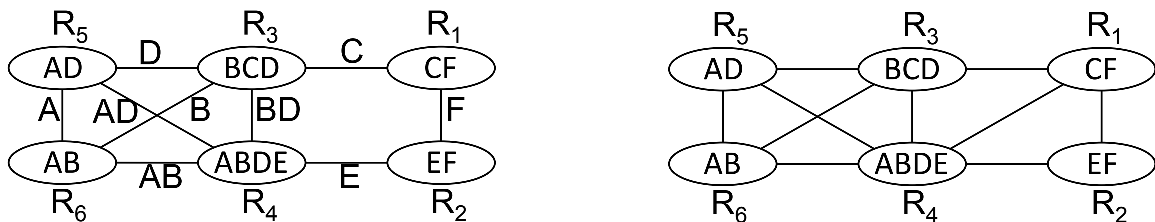


Figure 4.4: Triangulating a dual graph.

property triRNIC. Similarly to wRNIC, triRNIC depends on the particular triangulation of the dual graph.

An important feature of the triangulation process is that it operates *locally*, adding edges only where cycles of length four or more need to be shortened, irrespective of the degree of the vertices in the graph.

4.3.3 Triangulate a minimal dual graph: wtriRNIC

While using a minimal dual graph allows us to cope with the high density of difficult benchmark instances, triangulating the minimal dual graph allows us to boost propagation. We denote wtriRNIC the consistency resulting from applying ProcessQ on the triangulated minimal dual graph. Figure 4.5 shows the dual graph (density 47%) resulting from applying both reformulations in sequence for the example of Figure 2.5. As shown in Figure 4.1, wtriRNIC is strictly stronger than wRNIC applied

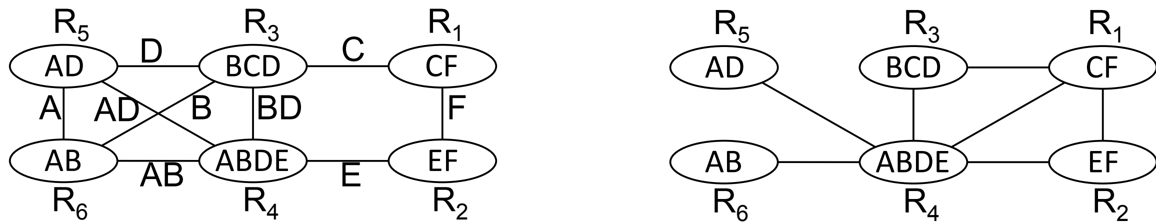


Figure 4.5: Triangulating a minimal dual graph.

on the same minimal dual graph, but strictly weaker than triRNIC. Further, it is not comparable with RNIC, which is enforced on the original dual graph. Figure 4.6 summarizes the relationships between RNIC, its reformulations, and R($*$,m)C based properties.

4.3.4 Select the appropriate RNIC: selRNIC

The algorithm summarized in Section 4.1, ProcessQ, enforces any of the four properties RNIC, triRNIC, wRNIC, and wtriRNIC on a CSP by operating on the original dual graph or some modification of it.

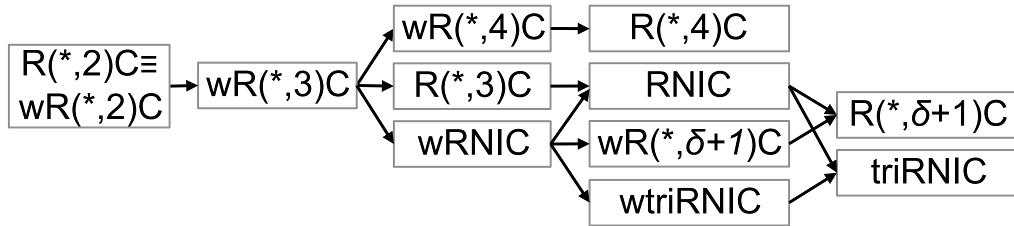


Figure 4.6: Relating RNIC, R($*$,m)C, and their studied variations.

- For RNIC, it uses the original dual graph (G_o).
- For wRNIC, it uses a minimal dual graph (G_w).
- For triRNIC, it uses a triangulated dual graph (G_{tri}).
- Finally, for wtriRNIC, it uses a triangulated minimal dual graph (G_{wtri}).

The selection policy shown in Figure 4.7 automatically chooses the dual graph on which to enforce RNIC by comparing the density d^G of a given dual graph G. The goal of this deliberation is to adjust the strength of propagation to the topology of the dual graph. Paraphrasing the content of Figure 4.7, we consider the dual graph of

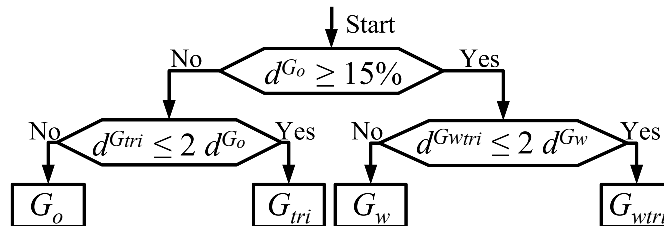


Figure 4.7: Selecting a dual graph for selRNIC.

density greater than or equal 15% to be too dense to be effectively processed by ProcessQ. For this reason, we choose to reformulate it by removing redundant edges. Whenever triangulation does not increase the density of a dual graph more than two fold, then the advantage of boosting propagation by creating loops and increasing neighborhood sizes outweighs the drawback of increasing the cost of operating on

larger neighborhoods. For the example of Figure 2.5, this policy correctly chooses the triangulated minimal dual graph (density 47%). While both operations of triangulating a dual graph and computing a minimal dual graph can be done efficiently and do not add *any* perceptible overhead in our experiments, the policy of Figure 4.7 applies each operation at most once. The resulting mechanism, which we denote selRNIC, nicely ties together our techniques in a consistent and adaptive framework.

Summary

In this chapter, we introduced an algorithm for enforcing RNIC, gave improvements on the algorithm, and studied the algorithm's complexity. We formally compared RNIC to R($*$,m)C. We introduced three reformulations to the dual graph that RNIC can be enforced on: (1) redundancy-free, (2) triangulated, and (3) triangulated redundancy-free. We then presented an adaptive, automatic strategy for selecting the dual graph on which to enforce RNIC.

Chapter 5

Evaluating RNIC

In this chapter, we empirically evaluate the performance of RNIC as a full lookahead schema for finding the first solution to a CSP using backtrack search. We compare this performance with that of the most commonly used consistency algorithms. We ran our experiments on 2875[1] instances with constraints defined in extension taken from the CSP Solver Competition[2] benchmarks. We limited the CPU time to one and a half hours per instance and the memory to 7GB. In [2010], Lecoutre gives a description of the benchmark problems.

First, we describe our experimental set-up and summarize our results. Then, we discuss in detail the results on individual benchmarks.

5.1 Experimental Setup

We evaluate and compare the performance of the following algorithms for enforcing consistency when used for full lookahead in a backtrack search procedure for finding the first solution of a CSP:

[1] We tested 1915 binary and 960 non-binary instances, grouped in 86 benchmarks.

[2] `http://www.cril.univ-artois.fr/CPAI08/`

- GAC
- wR($*$,m)C for $m = 2, 3, 4$
- RNIC and its variations: wRNIC, triRNIC, wtriRNIC and selRNIC.

5.1.1 Measured parameters

We measured the following parameters:

- For each benchmark category, we report the number of instances in the category, with the number completed by all algorithms in parenthesis, and the range of the number of constraints e.
- d^D: The range of the density of the dual graph on which a given algorithm operates.
- **Time**: The CPU time in milliseconds. Some data points are missing because some algorithms sometimes fail to finish within the allocated time window (90 minutes). For this reason, we consider the data to be right-censored and conduct a survival data analysis [Lee, 1992]. The survival data analysis does not make any assumption about the distribution of the data and yields a calculated mean CPU time for each algorithm. A '-' entry indicates that, even though the corresponding algorithm terminated on some instances, it did not terminate on enough instances to yield an accurate statistical mean.
- **S**: The equivalence classes of CPU performance. To compute the statistically significant categories, we perform a simple effects comparison between every two algorithms for a significance level of 0.05. This comparison requires a normal distribution of the non-censored data. For this analysis, we assume that all censored data points finished at the maximum cutoff time.

- **#C**: The number of instances completed by a given algorithm.
- **#F**: The number of instances on which the given algorithm is the fastest among all tested ones, where ties are awarded to all parties.
- **#BF**: The number of instances solved by a given algorithm in a backtrack-free manner.
- **#NV**: The average number of nodes visited by the corresponding search[3]. The averages are computed over only the instances completed by all tested algorithms, which is the number in parenthesis in the problem description. Thus, the values reported in **#NV** should be considered in light of the number of completed instances. A '-' entry in this column indicates that, even if the corresponding search completed on some instance, no instance completed by this algorithm was completed by all others, and thus no average value can be reported.

5.1.2 Method for statistical analysis

To compute the mean CPU time, we use the product-limit method, also called the Kaplan-Meier method. This method computes the survival time of each algorithm. (For us, survival means that the algorithm is still running.) It is nonparametric test: it makes no assumption about the distribution of the data.

To compute the significance classes between the algorithms, we generate a generalized linear mixed-model for each algorithm on a given benchmark. While generalized linear mixed models do not require that the data be normally distributed, they do not take into account censored data. The models assume that random effects are

[3]Note that the values of nodes visited in all experiments comply with the partial order shown in Figure 4.6.

normally distributed. We use those models to construct an approximate t-test between each pairs of algorithms. Even if the random effects assumption may not hold for our data, our analyses yielded consistent results on the various benchmarks, thus supporting the correctness of our conclusions. For computing the significance of the CPU measurements, the CPU time of each algorithm on a given instance is given as input to the model. We assume all censored data points finished at the maximum cutoff time.

5.2 Global Rankings

We break the 86 benchmark studied into ten categories adapted from those by Lecoutre,[4] slightly refined to better identify structured benchmarks and results similarity. The categories are: academic, assignment, Boolean, crossword, latin square, quasi-random (random benchmarks that have some structure), random, TSP. We do not report results of experiments on benchmarks that did not complete because of:

1. *Insufficient memory.* When the size of the relations is particularly large the index-tree data structure for checking quickly checking the consistency of two tuples becomes a significant overhead that is prohibitively large to store. In such cases, our implementation of RNIC runs out of memory. Another implementation of RNIC without the index-tree data-structure may be able to overcome this obstacle. This situation arises for the case of the following benchmarks: bddSmall, dag-half, lard.

2. *Insufficient memory.* Some benchmarks are not solved by any of the algorithms we tested in the allotted time period. Those benchmarks are: bddLarge, BH-

[4] The benchmark categories are given at `http://www.cril.univ-artois.fr/~lecoutre/benchmarks.htm`

4-13, BH-4-4, BH-4-7, bqwh-15-106, bqwh-18-141, frb50-23, frb53-24, frb56-25, frb59-26, QCP-20, QCP-25, QWH-20, QWH-25, rand-2-25, rand-2-26, rand-2-27, rand-2-50-23-fcd, rand-2-50-23, rand-3-24-24, rand-3-28-28, rand-3-28-28-fcd, renault.

Before summarizing the results of our experiments in Tables 5.1 and 5.2 (binary CSPs) and Table 5.3 (non-binary CSPs), we explain the entries in those tables:

- **Category** denotes the category of the benchmark.
- **Table** indicated the table where the results of the benchmark can be found.
- **#I** gives the number of instances in the benchmark.
- **Best CPU** lists the algorithms that are statistically best in terms of CPU time.
- **Fastest** denotes the algorithm that solved the largest number of instances the fastest.
- **#Comp** denotes the algorithms that solved the largest number of instances.
- **#BT-Free** denotes the algorithm that solved the largest number of instances in a backtrack-free manner.
- 'All' in any column indicates that all of the algorithms are equivalent according to that metric.
- When selRNIC chooses the same RNIC-based technique for all instances in the benchmark, we provide, in parenthesis, the RNIC-based technique selected.

We make the following observations on the results in those tables:

Table 5.1: Overview of the binary benchmarks tested (Part A).

Category	Benchmark	Table	#I	Best CPU	Fastest	#Comp	#BT-Free
Academic	coloring	Table A.34	22	GAC, RNIC, wRNIC, selRNIC	GAC	GAC	triRNIC
	hanoi	Table A.36	5	All	GAC	All	
	langford	Table A.18	4	GAC	GAC	wR(∗,2)C, GAC, wRNIC	
Assignment	driver	Table A.13	7	wR(∗,2)C, GAC	GAC	wR(∗,2)C, GAC	
Latin square	QCP-10	Table A.12	15	wR(∗,2)C, GAC, wRNIC	GAC	GAC	selRNIC (RNIC)
	QCP-15	Table A.20	15	GAC	GAC	GAC	
	QWH-10	Table A.37	10	All except triRNIC	GAC	All except triRNIC	
	QWH-15	Table A.38	10	GAC, selRNIC (RNIC), wRNIC	GAC	GAC, selRNIC (RNIC)	
Quasi-random	composed-25-1-25	Table A.1	10	wR(∗,3)C, wR(∗,4)C, selRNIC (RNIC), wtriRNIC	selRNIC (RNIC)	wR(∗,3)C, wR(∗,4)C, selRNIC (RNIC), wtriRNIC	wR(∗,3)C, wR(∗,4)C, selRNIC (RNIC), wtriRNIC
	composed-25-1-2	Table A.2	10	wR(∗,4)C, selRNIC (RNIC), wtriRNIC	selRNIC (RNIC)	wR(∗,4)C, selRNIC (RNIC), wtriRNIC	wR(∗,4)C, selRNIC (RNIC), wtriRNIC
	composed-25-1-40	Table A.3	10	wR(∗,3)C, wR(∗,4)C, selRNIC (RNIC), wtriRNIC	selRNIC (RNIC)	wR(∗,4)C, selRNIC (RNIC), wtriRNIC	wR(∗,4)C, selRNIC (RNIC), wtriRNIC
	composed-25-1-80	Table A.4	10	wR(∗,3)C, wR(∗,4)C, selRNIC (RNIC), wtriRNIC	selRNIC (RNIC)	wR(∗,4)C, selRNIC (RNIC)	wR(∗,4)C, selRNIC (RNIC)
	composed-25-10-20	Table A.5	10	selRNIC (RNIC)	GAC	selRNIC (RNIC)	
	composed-75-1-25	Table A.6	10	wR(∗,4)C, selRNIC (RNIC), wtriRNIC	selRNIC (RNIC)	wR(∗,4)C, selRNIC (RNIC), wtriRNIC	wR(∗,4)C, selRNIC (RNIC), wtriRNIC
	composed-75-1-2	Table A.7	10	wR(∗,4)C, selRNIC (RNIC), wtriRNIC	selRNIC (RNIC)	wR(∗,4)C, selRNIC (RNIC), wtriRNIC	wR(∗,4)C, selRNIC (RNIC), wtriRNIC
	composed-75-1-40	Table A.8	10	wR(∗,4)C, selRNIC (RNIC)	selRNIC (RNIC)	selRNIC (RNIC)	selRNIC (RNIC)
	composed-75-1-80	Table A.9	10	wR(∗,4)C, selRNIC (RNIC)	selRNIC (RNIC)	wR(∗,4)C, selRNIC (RNIC)	selRNIC (RNIC)
	ehi-85	Table A.10	100	wR(∗,4)C, selRNIC (RNIC)	wR(∗,4)C	wR(∗,4)C, selRNIC (RNIC)	selRNIC (RNIC)
	ehi-90	Table A.11	100	wR(∗,4)C, selRNIC (RNIC)	GAC	selRNIC (RNIC)	selRNIC (RNIC)
	geom	Table A.17	100	GAC	GAC	GAC	

Table 5.2: Overview of the binary benchmarks tested (Part B).

Category	Benchmark	Table	#I	Best CPU	Fastest	#Comp	#BT-Free
Random	frb30-15	A.35	10	wR(∗,2)C, GAC, wRNIC	GAC	wR(∗,2)C, wR(∗,3)C, GAC, wRNIC	
	frb35-17	A.14	10	GAC	GAC	GAC	
	frb40-19	A.15	10	GAC	GAC	GAC	
	frb45-21	A.16	10	GAC	GAC	GAC	
	marc	A.19	10	GAC	GAC	GAC	
	rand-2-23	A.21	10	GAC	GAC	GAC	
	rand-2-24	A.22	10	GAC	GAC	GAC	
	rand-2-30-15-fcd	A.23	50	wR(∗,2)C, GAC, wRNIC	GAC	wR(∗,2)C, GAC, wRNIC	
	rand-2-30-15	A.24	50	GAC	GAC	wR(∗,2)C, GAC, wRNIC	
	rand-2-40-19-fcd	A.25	50	GAC	GAC	GAC	
	rand-2-40-19	A.26	50	GAC	GAC	GAC	
	tightness0.1	A.27	100	GAC	GAC	GAC	
	tightness0.2	A.28	100	GAC	GAC	GAC	
	tightness0.35	A.29	100	GAC	GAC	GAC	
	tightness0.5	A.30	100	GAC	GAC	GAC	
	tightness0.65	A.31	100	GAC	GAC	GAC	
	tightness0.8	A.32	100	GAC	GAC	GAC	
	tightness0.9	A.33	100	GAC	GAC	GAC	

Table 5.3: Overview of the non-binary benchmarks tested.

Category	Benchmark	Table	#I	Best CPU	Fastest	#Comp	#BT-Free
Assignment	modifiedRenault	Table 5.6	50	wR(∗,3)C, wR(∗,4)C, wRNIC, wtriRNIC, selRNIC	wR(∗,2)C	wR(∗,4)C, wtriRNIC	wR(∗,4)C, wtriRNIC
Boolean	aim-100	Table 5.4	24	wR(∗,4)C, selRNIC (RNIC)	wR(∗,2)C, wRNIC	selRNIC (RNIC)	wR(∗,4)C, selRNIC (RNIC), triRNIC
	aim-200	Table 5.4	24	selRNIC (RNIC)	wR(∗,2)C	selRNIC (RNIC)	selRNIC (RNIC), triRNIC
	aim-50	Table A.39	24	All	GAC	All	RNIC, triRNIC, selRNIC
	dubois	Table A.40	13	wR(∗,2)C, and RNIC based	selRNIC (triRNIC)	selRNIC (triRNIC), wtriRNIC	
	jnhSat	Table A.44	16	wR(∗,2)C, GAC	GAC	wR(∗,2)C, GAC	
	jnhUnsat	Table A.45	34	wR(∗,2)C, GAC	GAC	wR(∗,2)C, GAC	wR(∗,2)C, wR(∗,3)C
	pret	Table A.52	8	All	triRNIC	All	
	ssa	Table A.41	8	All except wRNIC, triRNIC, wtriRNIC	GAC	RNIC, selRNIC	
	varDimacs	Table A.55	9	wR(∗,2)C, GAC	GAC	GAC	
Crossword	lexVg	Table 5.7	63	GAC	GAC	GAC	triRNIC, selRNIC (wtriRNIC)
	ogdVg	Table A.49	65	GAC	GAC	GAC	triRNIC, selRNIC (wtriRNIC)
	ukVg	Table A.50	65	GAC	GAC	GAC	triRNIC, selRNIC (wtriRNIC)
	wordsVg	Table A.51	65	GAC	GAC	GAC	triRNIC, selRNIC (wtriRNIC)
Quasi-random	dag-rand	Table 5.5	25	RNIC based	wRNIC	RNIC based	RNIC based
Random	rand-10-20-10	Table A.53	20	All	wR(∗,4)C	All	All except GAC
	rand-3-20-20-fcd	Table A.46	50	GAC	GAC	GAC	
	rand-3-20-20	Table A.47	50	GAC	GAC	GAC	triRNIC
	rand-3-24-24-fcd	Table A.48	50	GAC	GAC	GAC	
	rand-8-20-5	Table A.54	20	wR(∗,2)C	wR(∗,2)C	wR(∗,2)C	
TSP	travellingSalesman-20	Table A.42	15	GAC	GAC	GAC	
	travellingSalesman-25	Table A.43	15	GAC	GAC	GAC	

- On the crossword, random, and TSP benchmark, GAC exhibits the best performance in terms to CPU time and is able to complete the largest number of instances. However, GAC *never* ranks top in terms of solving the largest number of instances in a backtrack-free manner.

- triRNIC is the strongest form of RNIC and the strongest consistency property considered in this thesis. Theoretically speaking, triRNIC is guaranteed to have the smallest amount of nodes visited (and, thus, to solve the largest number of instances in a backtrack-free manner). However, it is also the most costly property to enforce in terms of CPU time, and often does not terminate in the allotted time limit. For that reason, it is not frequently listed in the column **#BT-Free**.

- selRNIC solves the largest number of instances (652) in a backtrack-free manner, followed by wR(∗,4)C (466), RNIC (394), wtriRNIC (247), wR(∗,3)C (237), triRNIC (172), wR(∗,2)C (169), GAC (151), wRNIC (126).

- RNIC and its variations outperform all other algorithms on structured benchmarks.

- In all of our experiments, when comparing selRNIC with a random selection of the four RNIC-based algorithms, within a 50ms error tolerance, selRNIC outperforms all four RNIC-based algorithms in a statistically significant manner. This result establishes that selRNIC is better than choosing any RNIC-based algorithm in a random manner (i.e., selRNIC is better than ‘chance’).

5.3 Detailed Analysis

In this section, we look at the individual results from each benchmark of the non-binary problems. The results on binary CSPs are similar to the non-binary ones and reported in Appendix A for readability. Indeed, on binary CSPs, we know that wRNIC is never stronger than wR($*$,3)C (Theorem 9), and, thus, their study is less interesting.

In Section 5.3.1, we discuss our results on the aim-100 and aim-200 benchmarks where RNIC and selRNIC perform well. In Section 5.3.2, we discuss our results on the dag-rand benchmark where RNIC and its variations outperform all other algorithms despite the high density of the dual graphs in the benchmark In Section 5.3.3, we discuss our results on the modifiedRenault benchmark where the weakened versions of the algorithms (wR($*$,3)C, wR($*$,4)C, wRNIC, wtriRNIC and selRNIC) outperform all others. In Section 5.3.4, we discuss our results on the lexVg benchmark, that has dense dual graphs, where GAC performs the best. Finally in Section 5.3.5, we discuss how the other benchmark map into the four benchmark that we single out. We report our results using the the metrics specified in Section 5.1.

5.3.1 The aim-100 & aim-200 benchmarks

Table 5.4 illustrates the usefulness of RNIC: it completes the largest number of instances (column **#C**), and solves, backtrack free, the largest number of instances (column **#BF**). A boldface value in a column indicates that the best performance in that metric. In terms of significance ranking, GAC, triRNIC, and wRNIC are not competitive, which can be attributed, for the case of triRNIC, to the large density of the dual graph on which it operates (26%–70.5%), and, for the case of wRNIC, to its small density (0.7%–2.7%). selRNIC outperforms all other algorithms in all metrics

Table 5.4: RNIC/selRNIC completes the largest number of instances, and solves, backtrack free, the largest number of instances.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
aim-100: 24(11) instances, $e \in$[150,570]							
wR(∗,2)C	[0.7%,2.7%]	1268786	**6**	B	19	5	324
wR(∗,3)C		1030715	1	B	20	7	152
wR(∗,4)C		946492	0	**A**	20	12	127
GAC	-	2045625	4	D	16	1	9286160
RNIC / selRNIC	[6.3%,8.1%]	**480865**	5	**A**	**22**	**16**	**100**
triRNIC	[26.0%,70.5%]	2905672	0	E	12	12	**100**
wRNIC	[0.7%,2.7%]	1125185	**6**	B	20	7	179
wtriRNIC	[7.1%,12.6%]	1643378	0	C	18	9	146
aim-200: 24(0) instances, $e \in$[302,1169]							
wR(∗,2)C	[0.4%,1.4%]	2736365	**9**	B	12	4	-
wR(∗,3)C		2313714	2	B	15	8	-
wR(∗,4)C		2345388	0	B	14	9	-
GAC	-	3979169	0	C	8	0	-
RNIC / selRNIC	[3.2%,8.5%]	**1346153**	6	**A**	**19**	**13**	-
triRNIC	[21.1%,71.8%]	5069082	0	D	2	2	-
wRNIC	[0.4%,1.4%]	2518443	2	B	13	5	-
wtriRNIC	[6.4%,11.4%]	3878709	0	C	7	7	-

except for **#F**, where it places the second.

5.3.2 The dag-rand benchmark

Like in Table 5.4, Table 5.5 shows a benchmark where RNIC and its variations perform the best. However, for the dag-rand benchmark, the density of the original dual graph is 100% on all the instances. When the density of the dual graph is 100%, RNIC is performing a backtrack search on the full dual encoding of of CSP problem, instead of a smaller sub-problem on the neighborhood. This positive result hints that there might be benefit to conduct search on the relations instead of a search on the variables.

Table 5.5: Despite the high density, RNIC is able to perform well.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
dag-rand: 25(0) instances, e ∈[16,16]							
wR(∗,2)C	[89.2%,98.3%]	-	-	-	-	-	-
wR(∗,3)C		-	-	-	-	-	-
wR(∗,4)C		-	-	-	-	-	-
GAC	-	5359472	0	B	1	0	-
RNIC	100.0%	41238	2	**A**	**25**	**25**	-
triRNIC	100.0%	38741	8	**A**	**25**	**25**	-
wRNIC	[89.2%,98.3%]	**38296**	**15**	**A**	**25**	**25**	-
wtriRNIC / selRNIC	[95%,100%]	179299	0	**A**	**25**	**25**	-

5.3.3 The modifiedRenault benchmark

Table 5.6 illustrates the usefulness of wRNIC and wtriRNIC. As stated above, the sheer number of relations combined with the large density in the dual graphs of the problems in this benchmark prevents us from executing RNIC and triRNIC. This situation demonstrates the benefits of using wRNIC and wtriRNIC, which actually are automatically chosen by selRNIC. Note also that wtriRNIC solves, backtrack free, all instances in this category. We cannot stress enough on the importance of this last fact: It is indicative of the tractability of this class of problems. Notice, despite selRNIC not having the smallest CPU time, there is not a statistically significant difference between the mean CPU time of selRNIC and the mean CPU time of wR(∗,4)C. Once again, GAC is in a lower significance class than selRNIC. So are RNIC and triRNIC, which was expected given the density of the dual graph.

5.3.4 The lexVg benchmark

In Tables 5.4, 5.5, and 5.6, selRNIC largely outperforms GAC by all accounts. Even if one was to use a high-performance GAC implementation such as the one in [Cheng and

Table 5.6: RNIC is hindered by the high density of the dual graph, but its weakened versions outperform all others.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
modifiedRenault: 50(1) instances, $e\in$[125,137]							
wR(∗,2)C	[2.1%,2.3%]	434378	**29**	B	46	41	**111**
wR(∗,3)C		117895	0	**A**	49	47	**111**
wR(∗,4)C		**34238**	0	**A**	**50**	**50**	**111**
GAC	-	2750326	14	C	26	5	**111**
RNIC	[44.7%,52.4%]	4924726	0	D	17	17	**111**
triRNIC	[45.6%,54.6%]	4868644	0	D	11	11	**111**
wRNIC	[2.1%,2.3%]	330987	2	**A**	47	45	**111**
wtriRNIC	[3.6%,5%]	239735	0	**A**	**50**	**50**	**111**
selRNIC	[2.1%,4.2%]	148431	0	**A**	49	48	**111**

Yap, 2010], the number of nodes visited by GAC remains orders of magnitude larger than that by selRNIC, and the number of instances solved backtrack-free significantly smaller. Only in Table 5.7 does GAC outperform the other algorithms in terms of CPU time only. Interestingly, however, on lexVg, and despite the high density

Table 5.7: GAC is best on CPU, triRNIC/selRNIC is best on #BF.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
lexVg: 63(40) instances, $e\in$[8,36]							
wR(∗,2)C	[48.5%,57.1%]	1001230	7	B	54	27	30
wR(∗,3)C		1889953	0	C	44	27	30
wR(∗,4)C		2129365	0	D	41	35	**3**
GAC	-	**116128**	**56**	**A**	**63**	26	25
RNIC	[48.5%,57.1%]	1791844	0	C	45	27	30
triRNIC	[57.6%,78.6%]	1103317	0	B	57	**57**	**3**
wRNIC	[48.5%,57.1%]	1813812	0	C	45	27	30
wtriRNIC / selRNIC	[57.6%,78.6%]	1094851	0	B	57	**57**	**3**

([57.6%,78.6%]) of the redundancy-free triangulated dual graph, wtriRNIC/selRNIC solves in a backtrack-free manner all but six of the instances in this set, thus hinting to the tractability of these instances. (The last six instance hit the time threshold.)

5.3.5 All other results

The 186 instances reported above are representative of the results obtained in our experiments, which were carried over 960 non-binary instances. Below, we classify the remaining tested instances into the four qualitative categories identified by the above four tables. The fifth category lists benchmarks that yielded inconclusive results. All individual tables can be found in Appendix A.

1. Similar to Table 5.4: aim-50 (Table A.39), dubois (Table A.40), ssa (Table A.41)

2. Similar to Table 5.5: All benchmarks shown.

3. Similar to Table 5.6: travellingSalesman-20 (Table A.42), travellingSalesman-25 (Table A.43)

4. Similar to Table 5.7: jnhSat (Table A.44), jnhUnsat (Table A.45), ogdVg (Table A.49), ukVg (Table A.50), wordsVg(Table A.51).

5. All Similar: pret (Table A.52), rand-10-20-10 (Table A.53), rand-8-20-5 (Table A.54), varDimacs (Table A.55).

Summary

In this chapter, we empirically evaluated, on benchmark problems, the advantage of enforcing RNIC and its variations and using them as full lookahead strategies for solving CSPs. These strategies performed statistically better than previous techniques (i.e., GAC2001 and wR($*$,m)C for $m = 2, 3, 4$) on assignment and quasi-random benchmark problems. Furthermore, among the variations of RNIC, selRNIC selected the most appropriate dual graph on which to enforce RNIC in a statistically significant manner.

Chapter 6

Propagation-Queue Management

Freuder identified the importance of the width of the constraint graph (and the corresponding variable ordering) for bounding the effort of solving the CSP [1982]. Dechter and Pearl [1987a] suggested using the induced width (and the corresponding perfect elimination ordering) obtained by some triangulation of the constraint network.[1] Further, in [Dechter and Pearl, 1989], they identified the connection of that approach with tree-decomposition techniques.[2]

In this chapter, we investigate the impact of the topology of the *dual graph* on the management of the propagation queue of RNIC. First, we present the five queue-management strategies that we propose, then evaluate them during pre-processing and as lookahead in a backtrack-search procedure for finding the first solution of a CSP.

[1] A detailed discussion and a historical summary of directional and adaptive consistency methods can be found in Chapter 4 of [Dechter, 2003].

[2] A detailed discussion and a historical summary of tree-decomposition methods can be found in Chapter 9 of [Dechter, 2003].

6.1 Queue-Management Strategies

We explore the following three directions for ordering the relations:

1. Arbitrary ordering of the relations in the propagation queue, as in Chapter 5,
2. Using a *perfect elimination ordering* (PEO) of the vertices of some triangulation of the dual graph, and
3. Using an ordering of the maximal cliques of some triangulation of the dual graph, which corresponds to a *tree-decomposition ordering* (TD).

For the example of Figure 6.1, a perfect elimination ordering obtained by applying the max-cardinality ordering is: $\langle R_6, R_5, R_3, R_4, R_2, R_1 \rangle$ and the maximal cliques ordering is: $\langle C_1, C_2, C_3, C_4 \rangle$, where $C_1 = \{R_4, R_6\}$, $C_2 = \{R_4, R_5\}$, $C_3 = \{R_1, R_3, R_4\}$, and $C_4 = \{R_1, R_2, R_4\}$. In order to triangulate the dual graph, we use the min-fill heuristic

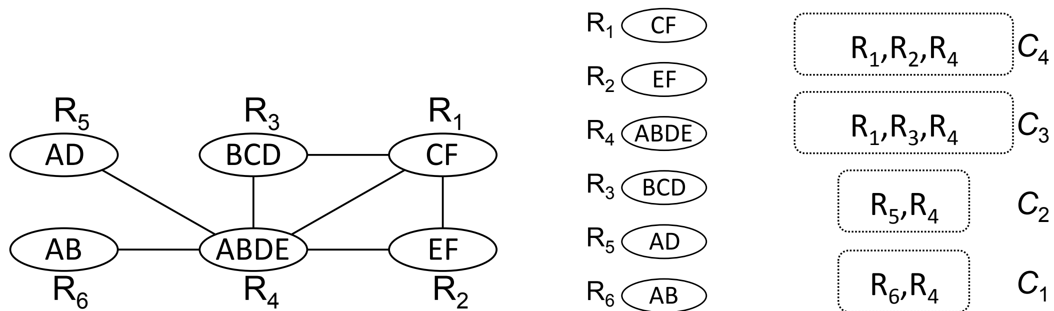


Figure 6.1: A triangulated dual graph (left) along with a perfect elimination ordering (center) and a maximal cliques ordering (right) where the orderings proceed from bottom to top.

[Kjærulff, 1990; Dechter, 2003]. We use a perfect elimination ordering (PEO) obtained by applying the max-cardinality ordering (MCO) of [Tarjan and Yannakakis, 1984] to the triangulated dual graph. Using this PEO, we find the maximal cliques with the algorithm of [Gavril, 1972].

We study the impact of the above orderings in three exact strategies and two approximate strategies (lazy) for managing the propagation queue of the RNIC algorithm in a backtrack search for finding the first solution of a CSP. We limit ourselves to triRNIC and wtriRNIC because the five proposed strategies are based on information obtained from the triangulation step. Below, we describe three exact strategies and two lazy strategies.

6.1.1 Exact strategies

PROCESSQ (Algorithm 1 in Section 4.1) uses two types of queues: A queue of the relations to be revised ($\mathcal{Q}_R$), and For each relation, a queue of tuples for which a support must be found ($\mathcal{Q}_t(R)$). Note that $\mathcal{Q}_R$ is static.[3] However, a relation R is processed only when its $\mathcal{Q}_t(R)$ is not empty. We consider the three exact following strategies:

1. *The arbitrary ordering* (QMS$_a$): The order of the relations in $\mathcal{Q}_R$ is arbitrary.

2. *The perfect elimination ordering* (QMS$_{PEO}$): This ordering aligns the relations in $\mathcal{Q}_R$ following the perfect elimination ordering explained above, and processes them back and forth in that order until quiescence (i.e., until all the tuples in the relations have appropriate supports).

3. *The tree decomposition ordering* (QMS$_{TD}$): This strategy maintains an additional queue, $\mathcal{Q}_C$, that is formed as follows:

   - For each maximal clique, C, a queue of the relations in this clique, $\mathcal{Q}_R(C)$, where the relations are stored in an arbitrary order.

[3] As in AC-1 [Mackworth, 1977] and unlike the queues of most modern consistency algorithms.

- Relations are listed in the queues of all the maximal cliques where they appear.
- The queue $\mathcal{Q}_C$ is a queue of the queues of those maximal-cliques, $\mathcal{Q}_R(C)$, aligned in the tree-decomposition ordering introduced above.
- A relation R is revised only when its queue of tuples, $\mathcal{Q}_t(R)$, is not empty.

The cliques are processed back forth in the order they are listed in $\mathcal{Q}_C$ until quiescence. Each time that a clique is considered, its queue is processed in an arbitrary ordering until quiescence before we can move to the next clique in the sequence.

Note that all three strategies above enforce the same consistency property. However, when the problem is unsolvable, the strategies may differ in the amount of tuples removed before discovering the problem is inconsistent.

6.1.2 Lazy strategies

The QMS$_{TD}$ strategy, described in Section 6.1.1 enforces RNIC, it:

1. processes each clique in the order considered,
2. iterates over the relations in the clique in an arbitrary order until quiescence,
3. then moves to the next cliques,
4. while traversing the ordering back and forth until quiescence.

In this section, we investigate if and when weakening QMS$_{TD}$ can reduce processing time. We examine weakening this strategy in two different ways, resulting in two 'lazy' strategies:

1. QMS_{LTD}: QMS_{LTD} relaxes Step 4 above, traversing the cliques only once, from bottom to top.

2. QMS_{L^2TD}: QMS_{L^2TD} relaxes Step 2 and 4 above, that is, it traverses the relations in the cliques only once, in a random order, and traverses the cliques also only once, from bottom to top.

The two lazy strategies are strictly weaker than the exact strategies.

6.2 Experimental Setup

To solve a CSP instance, one can enforce a given consistency property on the instance in a *pre-processing step* before running backtrack search. Enforcing consistency during search is called *lookahead*, as stated earlier. We study the impact of the queue-management strategies considered (see Table 6.1) on the CPU time to pre-process the CSP and find the first solution of a CSP. There are a total of 25 combinations of queue-

Table 6.1: Proposed queue management strategies.

Exact	QMS_a	Arbitrary ordering of the relations in the propagation queue.
Exact	QMS_{PEO}	The relations are ordered for revision using a perfect elimination ordering and the order is traversed back and forth until quiescence.
Exact	QMS_{TD}	The propagation queue is a sequence of list of relations appearing in the maximal cliques. The cliques are revised in sequence; each clique is revised until quiescence; and the sequence of cliques is revised back and forth until quiescence.
Lazy	QMS_{LTD}	Same as QMS_{TD}, however, the sequence of cliques is traversed only once.
Lazy	QMS_{L^2TD}	Same as QMS_{TD}, but traversing each clique only once and the relations in the cliques only once, in a random order.

management strategies that can be tested for pre-processing and lookahead. Instead

of testing all combinations of strategies, we evaluated the two steps in separation: executing them as a pre-processing step and as a lookahead during search.

- *For pre-processing*, the lazy approaches consistently yield lower CPU times than the exact strategies because they execute fewer revisions. However, they also filter fewer tuples, and as a result, the search space they produce is also larger than that obtained by any of the exact strategies. After some extensive testing, we found that the saving in CPU time is insignificant compared with the loss of filtering power. For this reason, below, we discuss only exact strategies for the pre-processing step, among which we show that QMS_{TD} yields the best performance in terms of CPU time.

- *For lookahead*, starting from a problem where the consistency property has been completely enforced (i.e., after pre-processing by an exact strategy), we find that QMS_{TD} is not only statistically equivalent to the lazy approaches on triangulated dual graphs, but in fact statistically better on triangulated minimal dual graphs.

The strategies were tested on the benchmark problems of the CSP Solver Competition introduced in Section 5.1 with the allocated time of one and a half hours and memory limit of 7GB per instance. We tested 2875 instances of binary and non-binary CSPs (1915 binary, 960 non-binary). We measured the following parameters:

- **Strategy**: One of the five strategies used.

- **Time**: The average CPU time in milliseconds for the strategy for pre-processing or for solving the pre-processed CSP. Time is treated as right-censored data because some experiments fail to finish within the allocated time window (90 minutes).

- **S**: A ranking in terms of equivalence classes of CPU performance. To compute performance equivalence-classes that are statistically significant, we performed a simple effects comparison between every two strategies for a significant level of 0.05. Rank A indicates the best performance, and rank C indicates the worse performance.

- **%**: The percent increase gained by the algorithm compared to the arbitrary strategy (QMS_a). A '-' entry indicates that there was no improvement over the arbitrary strategy.

6.3 Pre-Processing: Empirical Evaluations

We evaluated the exact strategies first on triangulated dual graphs (i.e., enforcing triRNIC) then triangulated minimal dual graphs (i.e., enforcing wtriRNIC).

On solvable instances, the performance of all three tested strategies (i.e., QMS_a, QMS_{PEO}, and QMS_{TD}) for both triRNIC and wtriRNIC are equivalent on solvable instances. However, on unsolvable instances, QMS_{TD} for enforcing triRNIC and QMS_{PEO} for enforcing wtriRNIC are superior to the other two on unsolvable instances, the former by a large margin and the latter by a small one. Below, we discuss the results in details.

6.3.1 Enforcing triRNIC

Table 6.2 reports the results of the queue-management strategies for enforcing triRNIC (i.e., triangulated dual graphs). The boldface values indicate the best performance. Our results indicate that QMS_{TD} exhibits the best performance, improving the CPU time of QMS_a by 46%.

Table 6.2: Pre-processing: QMSs for enforcing triRNIC.

Strategy	**Time**	**S**	**%**
QMS_a	1,410,292	C	-
QMS_{PEO}	1,186,691	B	16%
QMS_{TD}	**765,976**	**A**	**46%**

Tables 6.3 and 6.4 show the results after splitting the instances into solvable and unsolvable instances, respectively. On solvable instances, all three strategies have statistically equivalent performances, while QMS_a shows the lowest average CPU time. On unsolvable instances, QMS_{TD} shows the best performance, indicating that it detects unsatisfiable maximal cliques faster than the other strategies. This results reinforces the common knowledge that it is advantageous to combine higher level consistencies with the topology of the network.

Table 6.3: Pre-processing: QMSs for enforcing triRNIC on solvable instances.

Strategy	**Time**	**S**	**%**
QMS_a	**1,392,779**	**A**	-
QMS_{PEO}	1,411,518	**A**	-
QMS_{TD}	1,488,632	**A**	-

Table 6.4: Pre-processing: QMSs for enforcing triRNIC on unsolvable instances.

Strategy	**Time**	**S**	**%**
QMS_a	1,474,226	C	-
QMS_{PEO}	1,167,993	B	21%
QMS_{TD}	**564,497**	**A**	**62%**

6.3.2 Enforcing wtriRNIC

Table 6.5 reports the results of queue-management strategies for enforcing wtriRNIC (i.e., on triangulated minimal dual graphs). Both strategies QMS_{TD} and QMS_{PEO} show improvement on the CPU time over QMS_a, but the improvement is not statistically significant.

However, when separating the results on solvable (Table 6.6) and unsolvable (Table 6.7) instances, QMS_{PEO} exhibits the best performance in terms of CPU time in a significant manner on unsolvable instances.

Table 6.5: Pre-processing: QMSs for enforcing wtriRNIC.

Strategy	Time	S	%
QMS_a	479,725	**A**	-
QMS_{PEO}	**467,747**	**A**	**2%**
QMS_{TD}	476,604	**A**	1%

Table 6.6: Pre-processing: QMSs for enforcing wtriRNIC on solvable instances.

Strategy	Time	S	%
QMS_a	529,461	**A**	-
QMS_{PEO}	**518,292**	**A**	**2%**
QMS_{TD}	522,313	**A**	1%

Table 6.7: Pre-processing: QMSs for enforcing wtriRNIC on unsolvable instances.

Strategy	Time	S	%
QMS_a	360,769	B	-
QMS_{PEO}	**336,283**	**A**	**7%**
QMS_{TD}	370,795	B	-

6.4 Lookahead: Empirical Evaluations

We study the effect of queue management during lookahead and compare the performance all five proposed strategies in terms of average CPU time. We start from a CSP that has been pre-processed for consistency by triRNIC or wtriRNIC.

Our results report the same parameters as in Section 6.3 over the same 2875 instances of benchmark problems of the CSP Solver Competition. However, the **Time** column now denotes the average CPU time in milliseconds to solve the CSP, treating the data like right-censored data due to the 90 minute time-limit per instance. The CPU time does not include the time to pre-process the instance. We evaluate enforcing both triRNIC and wtriRNIC as full lookahead strategies.

6.4.1 Enforcing triRNIC

Table 6.8 reports the results of the queue-management strategies for enforcing triRNIC (i.e., using the triangulated dual graphs). The lazy strategy QMS_{LTD} shows the smallest average CPU time with a 68% improvement over QMS_a, However, QMS_{LTD}

Table 6.8: Lookahead: QMSs for enforcing triRNIC.

Strategies	Time	S	%
QMS_a	1,243,917	C	-
QMS_{PEO}	900,069	B	28%
QMS_{TD}	416,464	**A**	67%
QMS_{LTD}	**403,766**	**A**	**68%**
QMS_{L^2TD}	434,479	**A**	65%

is statistically equivalent to the two other strategy based on tree decomposition, QMS_{TD} and QMS_{L^2TD}.

Tables 6.9, and 6.10, show the results after splitting the test cases into solvable and unsolvable instances, respectively. On solvable instances, all of five strategies

Table 6.9: Lookahead: QMSs for enforcing triRNIC on solvable instances.

Strategies	Time	S	%
QMS_a	**722,887**	**A**	-
QMS_{PEO}	952,471	**A**	-
QMS_{TD}	1,013,529	**A**	-
QMS_{LTD}	957,518	**A**	-
QMS_{L^2TD}	948,261	**A**	-

Table 6.10: Lookahead: QMSs for enforcing triRNIC on unsolvable instances.

Strategies	Time	S	%
QMS_a	1,395,834	C	-
QMS_{PEO}	898,937	B	36%
QMS_{TD}	247,049	**A**	82%
QMS_{LTD}	**246,914**	**A**	**82%**
QMS_{L^2TD}	289,902	**A**	79%

are statistically equivalent. On unsolvable instances, all strategies based on tree decomposition are statistically equivalent and the best ranking. Interestingly, these are the same strategies that performed the best in Table 6.8.

Similar to the pre-processing results for enforcing triRNIC (Section 6.3.1), the results of this experiments confirm that propagation queues along the tree decomposition are able to detect unsatisfiable partial solutions quicker than an arbitrary ordering. Finally, lazy strategies do not exhibit any statistically significant advantage.

6.4.2 Enforcing wtriRNIC

Table 6.11 reports the results of queue-management strategies for enforcing wtriRNIC (i.e., triangulated minimal dual graphs). Unlike the results for enforcing triRNIC,

Table 6.11: Lookahead: QMSs for enforcing wtriRNIC.

Strategies	Time	S	%
QMS_a	628,523	C	-
QMS_{PEO}	582,629	B	7%
QMS_{TD}	**519,578**	**A**	**17%**
QMS_{LTD}	602,437	C	4%
QMS_{L^2TD}	575,277	C	8%

the two lazy strategies are are equivalent to QMS_a and thus not beneficial. QMS_{TD} performs the best, yielding a 17% improvement in CPU time over QMS_a. QMS_{PEO} is better than QMS_a and the lazy strategies, but not better than QMS_{TD}. Interestingly, QMS_{PEO} shows a higher average CPU time but a better ranking than QMS_{L^2TD}. That result is due to the fact the variance of the results of QMS_{L^2TD} is larger than that of QMS_{PEO}, causing it to be in the same significant category as QMS_a.

Tables 6.12 and 6.13 report the results split into solvable and unsolvable instances, respectively. Unlike the previous results where all of the strategies performed the same

Table 6.12: Lookahead: QMSs for enforcing wtriRNIC on solvable instances.

Strategies	Time	S	%
QMS_a	1.245,384	**A**	-
QMS_{PEO}	**1,214,177**	**A**	**5%**
QMS_{TD}	1,288,581	**A**	-
QMS_{LTD}	1,490,597	B	-
QMS_{L^2TD}	1,443,451	B	-

Table 6.13: Lookahead: QMSs for enforcing wtriRNIC on unsolvable instances.

Strategies	Time	S	%
QMS_a	286,561	C	-
QMS_{PEO}	232,320	B	19%
QMS_{TD}	**92,378**	**A**	**68%**
QMS_{LTD}	109,049	**A**	62%
QMS_{L^2TD}	92,946	**A**	68%

solvable instances, the lazy strategies performed the worst and the exact strategies performed the best. On unsolvable instances, the strategies involving the tree decom-

position, both exact and lazy, are statistically equivalent, and rank best. Unlike the results of the pre-processing data on wtriRNIC (Section 6.3.2), the perfect elimination ordering, QMS_{PEO}, was not the best on satisfiable instances, although it was significantly better than the arbitrary ordering, QMS_a. These results are similar to the results for triRNIC, where the best propagation queues are the ones based on tree decomposition. However, when both solvable and unsolvable instances are mixed, which is the case in practice, the lazy strategies do not perform well.

Summary

In this chapter, we investigated alternative queue-management strategies for enforcing triRNIC and wtriRNIC, and evaluated their performance during pre-processing and as full lookahead. The best strategy is shown to be the exact strategy exploiting the tree decomposition, QMS_{TD}.

Chapter 7

Conclusions and Future Work

Freuder and Elfe [1996] introduced Neighborhood Inverse Consistency (NIC) as a property defined on the values in the variables' domains of a Constraint Satisfaction Problem (CSP). NIC was introduced as a promising consistency property because of its light space requirement and its ability to focus attention on where a variable most tightly interacts with the problem, its neighborhood. However, Debruyne and Bessière [2001] showed that enforcing NIC on binary CSPs is ineffective on sparse graph and too costly on dense graphs. By proposing to enforce NIC on the dual graph instead of on the constraint network, we 'salvaged' the concept of NIC. Indeed, this shift allowed us to propose a new consistency property, Relational Neighborhood Inverse Consistency (RNIC), which we showed to be effective for solving CSPs.

This chapter concludes the thesis and summarizes our contributions and directions for future research.

7.1 Summary of Contributions

We have six main contributions:

1. We introduced a new consistency property, Relational Neighborhood Inverse Consistency (RNIC). The benefit of RNIC is that it adapts to the topology of its neighborhood and does not require the introduction of new relations to the CSP, but instead filters existing relations. We also characterized RNIC on both binary and non-binary CSPs.

2. We introduced an algorithm for enforcing RNIC. The complexity is polynomial for dual graphs of a fixed degree.

3. We introduced two reformulations of the dual graph of the CSP, yielding three variations of the RNIC property:

   a) *Removing Redundant Edges (wRNIC)*: Enforcing RNIC on a minimal dual graph, a redundancy-free dual graph.

   b) *Triangulating the Dual Graph (triRNIC)*: Enforcing RNIC on a triangulated dual graph, which breaks cycles of length four or more.

   c) *Triangulate a Minimal Dual Graph (wtriRNIC)*: Enforcing RNIC on a triangulated minimal dual graph, which has redundant edges removed, and then the resulting dual graph triangulated.

4. We also introduced a selection criteria to select the most appropriate of the four dual graphs (i.e., the original, minimal, triangulated, and triangulated minimal).

5. We evaluated RNIC and its variations when compared to GAC2011 and m-wise consistency on the CSP Solver Competition benchmark problems. We presented situations where RNIC and its variations perform the best, and other situations where GAC performs the best.

6. We proposed four new strategies (two exact and two approximate) for managing the propagation queue of a consistency algorithm. We empirically compared the different strategies, and concluded that the best is the exact strategy that exploits a tree decomposition.

7.2 Directions for Future Research

Below we identify directions for further research:

1. *Databases*: Database systems typically operate on large-sized tables (in terms of number of tuples). However, a join query is typically over few relations. In contrast, CSPs have relatively smaller variables' domains but large number of relations. Consequently, the computational cost models in the two domains differ: former is concerned with the number of access to disk whereas the latter is concerned with the computational cost of the algorithms. As the technology moves towards 'in-memory databases,' we conjecture that consistency techniques, especially ones that are based on filtering relations and enforcing higher levels of consistency, will become of great importance.

2. *Singleton consistency*: Our approach opens the door to the investigation of a new type of singleton consistency properties for CSPs. Instead of assigning the value of a *single* variable before enforcing some level of consistency on the CSP, as it is usually the case for Singleton Arc Consistency (SAC) [Bessiere *et al.*, 2011], we should investigate the effectiveness of 'assigning a tuple to a relation' in the dual problem. Such an approach would yield a new class of relational consistency properties, which could be called *relation-based singleton*

consistency properties. Note however, that, unlike RNIC, maintaining such properties during search is prohibitive in practice [Lecoutre and Prosser, 2006].

3. *Other relation definitions*: Our algorithm operates on relations defined in extension as consistent tuples (supports). Relations defined in extension as conflicts (no-goods) could be converted to supports, as we did here. Further, and also for constraints defined in intension, we could generate support tuples after applying GAC to the original CSP. For cases where it is important to keep all relation definitions in intension, we claim that a similar, albeit weaker, domain pruning can be achieved by executing RNIC on combinations of domain values that are consistent with the relations. We propose to mitigate the loss of information by generating new (support) constraints of some judiciously chosen scopes. We propose to investigate this approach in the future and evaluate its effectiveness.

4. *Redundancy removal*: As discussed in Section 3.4.1, on binary CSPs, it is possible to remove redundant edges in the dual graph to be a triangle-shaped grid. However, because the redundancy-free dual graphs are not unique, there are also non-grid shaped redundancy-free dual graphs. The algorithm used in this thesis, from [Janssen *et al.*, 1989], for removing redundant edges generates these triangle-shaped grids. We propose to investigate and understand why this algorithm favors the triangle-shaped grids. Further, the wRNIC results from this thesis can be compared with enforcing wRNIC using other algorithms for removing redundant edges.

5. *Propagation-queue management*: In Chapter 6, we studied four new queue-management strategies for enforcing triRNIC and wtriRNIC. These strategies could also be studied when used with other consistency algorithms, such as R($*$,m)C of [Karakashian *et al.*, 2010].

7.3 Final Note

Consistency properties are central to the Constraint Processing endeavor. Formalizing new such properties and developing new algorithms for enforcing them allows us to chip away, piece by piece, the barrier posed by complexity.

Appendix A

Data Sets

Below are the tables summarizing experimental results omitted from Chapter 5 in order to reduce clutter. We give the detailed analysis first for binary CSP benchmarks then for non-binary benchmarks.

A.1 Binary CSPs

Below are the extra tables from Section 5.3, giving individual benchmark results:

- RNIC and its variations perform best: composed-25-1-25 (Table A.1), composed-25-1-2 (Table A.2), composed-25-1-40 (Table A.3), composed-25-1-80 (Table A.4), composed-25-10-20 (Table A.5), composed-75-1-25 (Table A.6), composed-75-1-2 (Table A.7), composed-75-1-40 (Table A.8), composed-75-1-80 (Table A.9), ehi-85 (Table A.10), ehi-90 (Table A.11), QCP-10 (Table A.12).

- GAC performs well: driver (Table A.13), frb35-17 (Table A.14), frb40-19 (Table A.15), frb45-21 (Table A.16), geom (Table A.17), langford (Table A.18), marc (Table A.19), QCP-15 (Table A.20), rand-2-23 (Table A.21), rand-2-24

(Table A.22), rand-2-30-15-fcd (Table A.23), rand-2-30-15 (Table A.24), rand-2-40-19-fcd (Table A.25), rand-2-40-19 (Table A.26), tightness0.1 (Table A.27), tightness0.2 (Table A.28), tightness0.35 (Table A.29), tightness0.5 (Table A.30), tightness0.65 (Table A.31), tightness0.8 (Table A.32), tightness0.9 (Table A.33).

- Results are inconclusive on: coloring (Table A.34), frb30-15 (Table A.35), hanoi (Table A.36) QWH-10 (Table A.37), QWH-15 (Table A.38).

Table A.1: Statistical analysis of the composed-25-1-25 benchmark.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
composed-25-1-25: 10(0) instances, $e \in$[247,247]							
wR(∗,2)C	[1.5%,1.5%]	-	-	-	0	0	-
wR(∗,3)C		993	0	**A**	**10**	**10**	-
wR(∗,4)C		6376	0	**A**	**10**	**10**	-
GAC	-	-	-	-	0	0	-
RNIC / selRNIC	[12.3%,12.5%]	**398**	**8**	**A**	**10**	**10**	-
triRNIC	[53.4%,57.1%]	-	-	-	0	0	-
wRNIC	[1.5%,1.5%]	3781896	2	B	3	2	-
wtriRNIC	[6.1%,6.8%]	4828	0	**A**	**10**	**10**	-

Table A.2: Statistical analysis of the composed-25-1-2 benchmark.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
composed-25-1-2: 10(0) instances, $e \in$[224,224]							
wR(∗,2)C	[1.7%,1.7%]	-	-	-	0	0	-
wR(∗,3)C		1080677	0	B	8	7	-
wR(∗,4)C		5561	0	**A**	**10**	**10**	-
GAC	-	-	-	-	0	0	-
RNIC / selRNIC	[12.7%,12.9%]	**316**	**6**	**A**	**10**	**10**	-
triRNIC	[51.6%,53.9%]	-	-	-	0	0	-
wRNIC	[1.7%,1.7%]	3240059	4	C	4	4	-
wtriRNIC	[5.9%,6.4%]	1695	0	**A**	**10**	**10**	-
selRNIC	[12.7%,12.9%]	**316**	**6**	**A**	**10**	**10**	-

Table A.3: Statistical analysis of the composed-25-1-40 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
composed-25-1-40: 10(0) instances, e ∈[262,262]							
wR(∗,2)C	[1.4%,1.4%]	4860530	0	C	1	0	-
wR(∗,3)C		541675	0	**A**	9	6	-
wR(∗,4)C		6933	0	**A**	**10**	**10**	-
GAC	-	-	-	-	0	0	-
RNIC / selRNIC	[12.1%,12.3%]	**480**	7	**A**	**10**	**10**	-
triRNIC	[55.2%,57.5%]	-	-	-	0	0	-
wRNIC	[1.4%,1.4%]	3240199	3	B	4	1	-
wtriRNIC	[6.2%,6.7%]	5931	0	**A**	**10**	**10**	-

Table A.4: Statistical analysis of the composed-25-1-80 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
composed-25-1-80: 10(0) instances, e ∈[302,302]							
wR(∗,2)C	[1.3%,1.3%]	1687824	0	B	7	0	-
wR(∗,3)C		730063	0	**A**	9	4	-
wR(∗,4)C		8332	0	**A**	**10**	**10**	-
GAC	-	3848558	2	C	4	0	-
RNIC / selRNIC	[11.8%,12%]	**779**	**7**	**A**	**10**	**10**	-
triRNIC	[61.1%,63.6%]	-	-	-	0	0	-
wRNIC	[1.3%,1.3%]	1741182	1	B	7	0	-
wtriRNIC	[5.7%,5.9%]	576111	0	**A**	9	6	-

Table A.5: Statistical analysis of the composed-25-10-20 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
composed-25-10-20: 10(0) instances, e ∈[620,620]							
wR(∗,2)C	[0.6%,0.6%]	2472898	0	B	6	0	-
wR(∗,3)C		2713972	0	C	6	1	-
wR(∗,4)C		1906876	0	B	7	1	-
GAC	-	2179966	**5**	B	6	0	-
RNIC / selRNIC	[4.9%,5%]	**301220**	3	**A**	**10**	**10**	-
triRNIC	[26.5%,28.1%]	-	-	-	0	0	-
wRNIC	[0.6%,0.6%]	2504953	0	B	6	0	-
wtriRNIC	[3%,3.5%]	3235090	0	D	5	2	-

Table A.6: Statistical analysis of the composed-75-1-25 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
composed-75-1-25: 10(0) instances, e ∈[83,83]							
wR(∗,2)C	[0.6%,0.6%]	4861207	0	C	1	0	-
wR(∗,3)C		1082373	0	B	8	6	-
wR(∗,4)C		18150	0	**A**	**10**	**10**	-
GAC	-	-	-	-	0	-	-
RNIC / selRNIC	[4.8%,4.9%]	**1186**	**10**	**A**	**10**	**10**	-
triRNIC	[48.1%,50.2%]	-	-	-	0	-	-
wRNIC	[0.6%,0.6%]	4861555	0	C	1	0	-
wtriRNIC	[5.8%,6.7%]	201033	0	**A**	**10**	**10**	-

Table A.7: Statistical analysis of the composed-75-1-2 benchmark.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
composed-75-1-2: 10(0) instances, $e \in$[624,624]							
wR(∗,2)C	[0.6%,0.6%]	-	-	-	0	0	-
wR(∗,3)C		2161663	0	B	6	6	-
wR(∗,4)C		17363	0	**A**	**10**	**10**	-
GAC	-	-	-	-	0	0	-
RNIC / selRNIC	[4.9%,5.0%]	**1080**	9	**A**	**10**	**10**	-
triRNIC	[47%,49.9%]	-	-	-	0	0	-
wRNIC	[0.6%,0.6%]	4860070	1	C	1	1	-
wtriRNIC	[5.6%,6%]	41763	0	**A**	**10**	**10**	-
selRNIC	[4.9%,5%]	**1080**	9	**A**	**10**	**10**	-

Table A.8: Statistical analysis of the composed-75-1-40 benchmark.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
composed-75-1-40: 10(0) instances, $e \in$[662,662]							
wR(∗,2)C	[0.6%,0.6%]	4860197	0	D	1	0	-
wR(∗,3)C		2702992	0	C	5	2	-
wR(∗,4)C		556931	0	**A**	9	9	-
GAC	-	-	-	-	0	0	-
RNIC / selRNIC	[4.8%,4.9%]	**1257**	**10**	**A**	**10**	**10**	-
triRNIC	[48.7%,51.1%]	-	-	-	0	0	-
wRNIC	[0.6%,0.6%]	4860283	0	D	1	0	-
wtriRNIC	[5.8%,6.6%]	1714948	0	B	8	8	-

Table A.9: Statistical analysis of the composed-75-1-80 benchmark.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
composed-75-1-80: 10(0) instances, $e \in$[702,702]							
wR(∗,2)C	[0.5%,0.5%]	3255454	0	C	4	0	-
wR(∗,3)C		1622932	0	B	7	1	-
wR(∗,4)C		216955	0	**A**	**10**	8	-
GAC	-	3780016	3	C	3	0	-
RNIC / selRNIC	[4.7%,4.8%]	**1543**	**7**	**A**	**10**	**10**	-
triRNIC	[50.7%,53.3%]	-	-	-	0	0	-
wRNIC	[0.5%,0.5%]	3272328	0	C	4	0	-
wtriRNIC	[5.5%,6.3%]	4028376	0	C	4	1	-

Table A.10: Statistical analysis of the ehi-85 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
ehi-85: 100(0) instances, e ∈[4081,4137]							
wR(∗,2)C	[0.1%,0.1%]	2855886	2	C	59	0	-
wR(∗,3)C		2339890	12	B	61	2	-
wR(∗,4)C		**151627**	**38**	**A**	**100**	89	-
GAC	-	2930293	35	D	58	0	-
RNIC / selRNIC	[2.6%,2.6%]	179562	8	**A**	**100**	**100**	-
triRNIC	[36%,41.9%]	-	-	-	0	0	-
wRNIC	[0.1%,0.1%]	2654968	5	C	60	1	-
wtriRNIC	[3.3%,4.1%]	-	-	-	0	0	-

Table A.11: Statistical analysis of the ehi-90 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
ehi-90: 100(0) instances, e ∈[4343,4400]							
wR(∗,2)C	[0.1%,0.1%]	3254714	0	D	49	0	-
wR(∗,3)C		2516736	10	B	57	2	-
wR(∗,4)C		237040	31	**A**	99	86	-
GAC	-	3106651	**41**	C	44	0	-
RNIC / selRNIC	[2.5%,2.5%]	**187288**	15	**A**	**100**	**100**	-
triRNIC	[36.6%,41.8%]	-	-	-	0	0	-
wRNIC	[0.1%,0.1%]	2988408	3	C	50	1	-
wtriRNIC	[3.3%,4.1%]	-	-	-	0	0	-

Table A.12: Statistical analysis of the QCP-10 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
QCP-10: 15(0) instances, e ∈[822,822]							
wR(∗,2)C	[0.5%,0.5%]	560920	0	**A**	14	2	-
wR(∗,3)C		1004741	0	B	13	4	-
wR(∗,4)C		1059011	0	B	13	4	-
GAC	-	**51615**	**13**	**A**	**15**	4	-
RNIC / selRNIC	[3.8%,3.8%]	817085	1	B	13	**9**	-
triRNIC	[29.1%,38.8%]	-	-	-	0	0	-
wRNIC	[0.5%,0.5%]	648826	1	**A**	14	2	-
wtriRNIC	[3.8%,4.2%]	2516534	0	C	9	4	-

Table A.13: Statistical analysis of the driver benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
driver: 7(1) instances, e ∈[217,17447]							
wR(∗,2)C	[0%,1.5%]	2440546	0	**A**	**6**	**1**	**71**
wR(∗,3)C		3744919	0	C	3	**1**	**71**
wR(∗,4)C		3895826	0	C	2	**1**	**71**
GAC	-	**1801784**	**4**	**A**	**6**	**1**	**71**
RNIC / selRNIC	[1.7%,5.5%]	4628591	0	D	1	**1**	**71**
triRNIC	[12.9%,21.3%]	4628690	0	D	1	**1**	**71**
wRNIC	[0%,1.5%]	3138109	0	B	5	**1**	**71**
wtriRNIC	[0.5%,3.7%]	4628586	0	D	1	**1**	**71**

Table A.14: Statistical analysis of the frb35-17 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
frb35-17: 10(0) instances, $e \in$[260,273]							
wR(∗,2)C	[1.4%,1.4%]	2683904	0	B	9	0	-
wR(∗,3)C		4947205	0	C	1	0	-
wR(∗,4)C		5233668	0	C	1	0	-
GAC	-	**137656**	**10**	**A**	**10**	0	-
RNIC / selRNIC	[11.1%,11.3%]	5322187	0	C	1	0	-
triRNIC	[56.7%,59.3%]	-	-	-	0	0	-
wRNIC	[1.4%,1.4%]	3196734	0	B	6	0	-
wtriRNIC	[6.6%,7.4%]	-	-	-	0	0	-

Table A.15: Statistical analysis of the frb40-19 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
frb40-19: 10(0) instances, $e \in$[308,326]							
wR(∗,2)C	[1.2%,1.2%]	-	-	-	0	0	-
wR(∗,3)C		-	-	-	0	0	-
wR(∗,4)C		-	-	-	0	0	-
GAC	-	**670469**	**10**	**A**	**10**	0	-
RNIC / selRNIC	[9.6%,10%]	-	-	-	0	0	-
triRNIC	[55.9%,58.6%]	-	-	-	0	0	-
wRNIC	[1.2%,1.2%]	-	-	-	0	0	-
wtriRNIC	[6.4%,7.2%]	-	-	-	0	0	-

Table A.16: Statistical analysis of the frb45-21 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
frb45-21: 10(0) instances, $e \in$[369,394]							
wR(∗,2)C	[1%,1%]	-	-	-	0	0	-
wR(∗,3)C		-	-	-	0	0	-
wR(∗,4)C		-	-	-	0	0	-
GAC	-	**4598528**	**2**	**A**	**2**	0	-
RNIC / selRNIC	[8.6%,8.8%]	-	-	-	0	0	-
triRNIC	[57.1%,58.4%]	-	-	-	0	0	-
wRNIC	[1%,1%]	-	-	-	0	0	-
wtriRNIC	[6%,6.7%]	-	-	-	0	0	-

Table A.17: Statistical analysis of the geom benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
geom: 100(0) instances, $e \in$[339,555]							
wR(∗,2)C	[0.7%,1.1%]	982397	1	B	86	9	-
wR(∗,3)C		1588468	0	C	77	10	-
wR(∗,4)C		2240421	0	D	69	10	-
GAC	-	**295277**	**99**	**A**	**100**	19	-
RNIC / selRNIC	[7.9%,8.9%]	2373910	0	D	65	**57**	-
triRNIC	[25.1%,44.9%]	-	-	-	0	0	-
wRNIC	[0.7%,1.1%]	1032776	0	B	85	9	-
wtriRNIC	[3.9%,5.7%]	5206957	0	E	18	4	-

Table A.18: Statistical analysis of the langford benchmark.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
langford: 4(1) instances, $e \in$[28,528]							
wR(∗,2)C	[0.7%,12.7%]	1958420	0	B	**4**	**1**	**8**
wR(∗,3)C		2747773	0	B	2	**1**	**8**
wR(∗,4)C		2949355	0	B	2	**1**	**8**
GAC	-	**140138**	**4**	**A**	**4**	**1**	**8**
RNIC	[11.8%,44.4%]	2862045	0	B	2	**1**	**8**
triRNIC	[70.6%,81.5%]	4050008	0	C	1	**1**	**8**
wRNIC	[0.7%,12.7%]	2246965	0	B	**4**	**1**	**8**
wtriRNIC	[2.9%,22.2%]	4050005	0	C	1	**1**	**8**
selRNIC	[11.8%,22.2%]	2862045	0	C	2	**1**	**8**

Table A.19: Statistical analysis of the marc benchmark.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
marc: 10(0) instances, $e \in$[3160,4560]							
wR(∗,2)C	[0%,0.1%]	-	-	-	0	0	-
wR(∗,3)C		-	-	-	0	0	-
wR(∗,4)C		-	-	-	0	0	-
GAC	-	**39692**	**10**	**A**	**10**	**5**	-
RNIC selRNIC	[8.2%,9.9%]	-	-	-	0	0	-
triRNIC	[68.2%,68.3%]	-	-	-	0	0	-
wRNIC	[0%,0.1%]	-	-	-	0	0	-
wtriRNIC	[0.6%,0.8%]	-	-	-	0	0	-

Table A.20: Statistical analysis of the QCP-15 benchmark.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
QCP-15: 15(0) instances, $e \in$[2519,2520]							
wR(∗,2)C	[0.2%,0.2%]	-	-	-	0	0	-
wR(∗,3)C		-	-	-	0	0	-
wR(∗,4)C		-	-	-	0	0	-
GAC	-	**3673337**	**5**	**A**	**6**	0	-
RNIC / selRNIC	[3.7%,3.7%]	5222044	1	B	1	0	-
triRNIC	[25.9%,30.1%]	-	-	-	0	0	-
wRNIC	[0.2%,0.2%]	-	-	-	0	0	-
wtriRNIC	[2.4%,2.7%]	-	-	-	0	0	-

Table A.21: Statistical analysis of the rand-2-23 benchmark.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-2-23: 10(0) instances, $e \in$[253,253]							
wR(∗,2)C	[1.5%,1.5%]	5367140	0	B	1	**0**	-
wR(∗,3)C		-	-	-	0	**0**	-
wR(∗,4)C		-	-	-	0	**0**	-
GAC	-	**1766384**	**10**	**A**	**10**	**0**	-
RNIC	[16.7%,16.7%]	-	-	-	0	**0**	-
triRNIC	[72.2%,72.2%]	-	-	-	0	**0**	-
wRNIC	[1.5%,1.5%]	-	-	-	0	**0**	-
wtriRNIC	[4.9%,4.9%]	-	-	-	0	**0**	-
selRNIC	[1.5%,1.5%]	-	-	-	0	**0**	-

Table A.22: Statistical analysis of the rand-2-24 benchmark.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-2-24: 10(0) instances, e ∈[276,276]							
wR(∗,2)C	[1.4%,1.4%]	-	-	-	0	**0**	-
wR(∗,3)C		-	-	-	0	**0**	-
wR(∗,4)C		-	-	-	0	**0**	-
GAC	-	**4039559**	**6**	**A**	**6**	**0**	-
RNIC	[16%,16%]	-	-	-	0	**0**	-
triRNIC	[72%,72%]	-	-	-	0	**0**	-
wRNIC	[1.4%,1.4%]	-	-	-	0	**0**	-
wtriRNIC	[4.7%,4.7%]	-	-	-	0	**0**	-
selRNIC	[1.4%,1.4%]	-	-	-	0	**0**	-

Table A.23: Statistical analysis of the rand-2-30-15-fcd benchmark.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-2-30-15-fcd: 50(0) instances, e ∈[208,230]							
wR(∗,2)C	[1.6%,1.8%]	262169	0	**A**	**50**	0	-
wR(∗,3)C		1303252	0	B	47	0	-
wR(∗,4)C		3295023	0	C	32	0	-
GAC	-	**10283**	**50**	**A**	**50**	0	-
RNIC / selRNIC	[12.7%,13.1%]	3617208	0	C	26	**1**	-
triRNIC	[57.5%,62.4%]	-	-	-	0	0	-
wRNIC	[1.6%,1.8%]	357474	0	**A**	**50**	0	-
wtriRNIC	[7.2%,8.2%]	4764364	0	D	11	0	-

Table A.24: Statistical analysis of the rand-2-30-15 benchmark.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-2-30-15: 50(0) instances, e ∈[208,230]							
wR(∗,2)C	[1.6%,1.8%]	439593	0	B	**50**	0	-
wR(∗,3)C		2338485	0	C	45	0	-
wR(∗,4)C		4563283	0	D	15	0	-
GAC	-	**21514**	**50**	**A**	**50**	0	-
RNIC	[12.7%,13.1%]	4634325	0	D	14	0	-
triRNIC	[57.5%,62.4%]	-	-	-	0	0	-
wRNIC	[1.6%,1.8%]	552505	0	B	**50**	0	-
wtriRNIC	[7.2%,8.2%]	5158351	0	E	4	0	-
selRNIC	[12.7%,13.1%]	4634325	0	D	14	0	-

Table A.25: Statistical analysis of the rand-2-40-19-fcd benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
rand-2-40-19-fcd: 50(0) instances, $e \in$[325,351]							
wR(∗,2)C	[1.1%,1.2%]	5011099	0	B	5	0	-
wR(∗,3)C		5354294	0	B	2	0	-
wR(∗,4)C		-	-	-	0	0	-
GAC	-	**1895733**	**45**	**A**	**45**	0	-
RNIC	[9.6%,9.9%]	-	-	-	0	0	-
triRNIC	[56.2%,60.1%]	-	-	-	0	0	-
wRNIC	[1.1%,1.2%]	5331479	0	B	1	0	-
wtriRNIC	[5.8%,6.8%]	-	-	-	0	0	-
selRNIC	[9.6%,9.9%]	-	-	-	0	0	-

Table A.26: Statistical analysis of the rand-2-40-19 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
rand-2-40-19: 50(0) instances, $e \in$[325,351]							
wR(∗,2)C	[1.1%,1.2%]	-	-	-	0	0	-
wR(∗,3)C		-	-	-	0	0	-
wR(∗,4)C		-	-	-	0	0	-
GAC	-	**3445956**	**37**	**A**	**37**	0	-
RNIC	[9.6%,9.9%]	-	-	-	0	0	-
triRNIC	[56.2%,60.1%]	-	-	-	0	0	-
wRNIC	[1.1%,1.2%]	-	-	-	0	0	-
wtriRNIC	[5.8%,6.8%]	-	-	-	0	0	-
selRNIC	[9.6%,9.9%]	-	-	-	0	0	-

Table A.27: Statistical analysis of the tightness0.1 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
tightness0.1: 100(0) instances, $e \in$[746,753]							
wR(∗,2)C	[0.5%,0.5%]	5130817	0	B	11	0	-
wR(∗,3)C		5376351	0	B	2	0	-
wR(∗,4)C		-	-	-	0	0	-
GAC	-	**839531**	**100**	**A**	100	0	-
RNIC / selRNIC	[9.8%,9.8%]	-	-	-	0	0	-
triRNIC	[68.8%,69.3%]	-	-	-	0	0	-
wRNIC	[0.5%,0.5%]	5146666	0	B	11	0	-
wtriRNIC	[2.2%,2.5%]	-	-	-	0	0	-

Table A.28: Statistical analysis of the tightness0.2 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
tightness0.2: 100(0) instances, $e \in$[414,414]							
wR(∗,2)C	[0.9%,0.9%]	4104489	0	B	47	0	-
wR(∗,3)C		5296318	0	C	4	0	-
wR(∗,4)C		-	-	-	0	0	-
GAC	-	**404025**	**100**	**A**	**100**	0	-
RNIC / selRNIC	[9.7%,10%]	-	-	-	0	0	-
triRNIC	[59.8%,63.8%]	-	-	-	0	0	-
wRNIC	[0.9%,0.9%]	4339907	0	B	40	0	-
wtriRNIC	[4.8%,5.5%]	-	-	-	0	0	-

Table A.29: Statistical analysis of the tightness0.35 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
tightness0.35: 100(0) instances, e ∈[250,250]							
wR(∗,2)C	[1.5%,1.5%]	2987884	0	B	74	0	-
wR(∗,3)C		5110330	0	D	12	0	-
wR(∗,4)C		5392481	0	E	1	0	-
GAC	-	**188324**	**100**	**A**	**100**	0	-
RNIC / selRNIC	[9.5%,10.1%]	5362778	0	D	1	0	-
triRNIC	[50.3%,56.1%]	-	-	-	0	0	-
wRNIC	[1.5%,1.5%]	3545104	0	C	61	0	-
wtriRNIC	[7.5%,9%]	-	-	-	0	0	-

Table A.30: Statistical analysis of the tightness0.5 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
tightness0.5: 100(0) instances, e ∈[180,180]							
wR(∗,2)C	[2%,2%]	3313395	0	B	68	0	-
wR(∗,3)C		4986286	0	D	13	0	-
wR(∗,4)C		5340159	0	E	3	0	-
GAC	-	**215932**	**100**	**A**	**100**	0	-
RNIC / selRNIC	[9.4%,10.2%]	5260525	0	E	5	0	-
triRNIC	[43.8%,49.6%]	-	-	-	0	0	-
wRNIC	[2%,2%]	4174121	0	C	41	0	-
wtriRNIC	[9.2%,11%]	-	-	-	0	0	-

Table A.31: Statistical analysis of the tightness0.65 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
tightness0.65: 100(0) instances, e ∈[40,40]							
wR(∗,2)C	[0%,2.5%]	2993901	0	B	74	0	-
wR(∗,3)C		4887847	0	D	21	0	-
wR(∗,4)C		5302422	0	E	3	0	-
GAC	-	**213607**	**100**	**A**	**100**	0	-
RNIC / selRNIC	[9.2%,10.3%]	5131256	0	D	11	0	-
triRNIC	[37.1%,42.9%]	-	-	-	0	-	-
wRNIC	[0%,2.5%]	3910508	0	C	48	0	-
wtriRNIC	[10%,12.1%]	-	-	-	0	-	-

Table A.32: Statistical analysis of the tightness0.8 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
tightness0.8: 100(0) instances, e ∈[103,103]							
wR(∗,2)C	[0%,3.2%]	2861521	1	B	65	0	-
wR(∗,3)C		4231747	0	D	37	0	-
wR(∗,4)C		5085661	0	E	14	0	-
GAC	-	**552869**	**96**	**A**	**97**	0	-
RNIC / selRNIC	[9.1%,10.7%]	5116586	0	E	11	0	-
triRNIC	[30%,36.7%]	-	-	-	0	0	-
wRNIC	[0%,3.2%]	3548849	0	C	51	0	-
wtriRNIC	[10.2%,12.3%]	-	-	-	0	0	-

Table A.33: Statistical analysis of the tightness0.9 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
tightness0.9: 100(0) instances, e ∈[84,84]							
wR(∗,2)C	[0%,3.8%]	2781337	3	B	71	0	-
wR(∗,3)C		3806532	0	C	44	0	-
wR(∗,4)C		4620144	0	D	22	0	-
GAC	-	**812311**	**91**	**A**	**94**	0	-
RNIC / selRNIC	[8.9%,10.8%]	5035376	0	E	11	0	-
triRNIC	[24.6%,31.5%]	-	-	-	0	0	-
wRNIC	[0%,3.8%]	3819519	0	C	40	0	-
wtriRNIC	[10.1%,12.3%]	-	-	-	0	0	-

Table A.34: Statistical analysis of the coloring benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
coloring: 22(8) instances, e ∈[78,5714]							
wR(∗,2)C	[0.1%,4.1%]	1675845	0	B	16	3	255
wR(∗,3)C		1804961	0	B	15	4	251
wR(∗,4)C		2066693	0	B	14	4	247
GAC	-	**91665**	**22**	**A**	**22**	5	113
RNIC	[1.8%,15%]	609841	0	**A**	20	7	19
triRNIC	[23.2%,85.7%]	2216801	0	C	13	**13**	**7**
wRNIC	[0.1%,4.1%]	484962	0	**A**	21	7	254
wtriRNIC	[5.1%,32.4%]	1731333	0	B	15	9	246
selRNIC	[1.8%,23.2%]	609741	0	**A**	20	10	15

Table A.35: Statistical analysis of the frb30-15 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
frb30-15: 10(0) instances, e ∈[208,217]							
wR(∗,2)C	[1.7%,1.8%]	173231	0	**A**	**10**	0	-
wR(∗,3)C		982760	0	B	**10**	0	-
wR(∗,4)C		3665867	0	C	6	0	-
GAC	-	**6806**	**10**	**A**	**10**	0	-
RNIC	[12.8%,12.9%]	3726715	0	C	5	0	-
triRNIC	[57.8%,60.3%]	-	-	-	0	0	-
wRNIC	[1.7%,1.8%]	215756	0	**A**	**10**	0	-
wtriRNIC	[7.6%,7.9%]	4680734	0	D	3	0	-
selRNIC	[12.8%,12.9%]	3726715	0	C	5	0	-

Table A.36: Statistical analysis of the hanoi benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
hanoi: 5(5) instances, $e \in$[5,125]							
wR(∗,2)C	[1.6%,40%]	6912	0	**A**	**5**	**5**	**48**
wR(∗,3)C		8246	1	**A**	**5**	**5**	**48**
wR(∗,4)C		12362	0	**A**	**5**	**5**	**48**
GAC	-	**2282**	**4**	**A**	**5**	**5**	**48**
RNIC	[1.6%,40%]	12270	0	**A**	**5**	**5**	**48**
triRNIC	[1.6%,40%]	18984	0	**A**	**5**	**5**	**48**
wRNIC	[1.6%,40%]	12200	0	**A**	**5**	**5**	**48**
wtriRNIC	[1.6%,40%]	19264	0	**A**	**5**	**5**	**48**
selRNIC	[1.6%,40%]	18984	0	**A**	**5**	**5**	**48**

Table A.37: Statistical analysis of the QWH-10 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
QWH-10: 10(0) instances, $e \in$[756,756]							
wR(∗,2)C	[0.5%,0.5%]	3005	0	**A**	**10**	1	-
wR(∗,3)C		7263	0	**A**	**10**	1	-
wR(∗,4)C		32874	0	**A**	**10**	1	-
GAC	-	**250**	**10**	**A**	**10**	3	-
RNIC / selRNIC	[3.9%,3.9%]	7107	0	**A**	**10**	8	-
triRNIC	[28.8%,37.2%]	-	-	-	0	0	-
wRNIC	[0.5%,0.5%]	1985	0	**A**	**10**	1	-
wtriRNIC	[4.1%,4.6%]	712606	0	**A**	**10**	1	-

Table A.38: Statistical analysis of the QWH-15 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
QWH-15: 10(0) instances, $e \in$[2324,2324]							
wR(∗,2)C	[0.2%,0.2%]	1322299	0	B	9	0	-
wR(∗,3)C		2431407	0	C	7	0	-
wR(∗,4)C		3669685	0	D	4	0	-
GAC	-	888310	**5**	**A**	**10**	0	-
RNIC / selRNIC	[3.8%,3.8%]	**402069**	3	**A**	**10**	**4**	-
triRNIC	[25.9%,29.5%]	-	-	-	0	0	-
wRNIC	[0.2%,0.2%]	1077665	2	**A**	9	0	-
wtriRNIC	[2.6%,3%]	-	-	-	0	0	-

A.2 Non-Binary CSPs

Below are the tables for the tested non-binary CSPs that were omitted from Section 5.3:

- RNIC and its variations perform best: aim-50 (Table A.39), dubois (Table A.40), ssa (Table A.41)
- Redundancy removal helps: travellingSalesman-20 (Table A.42), travellingSalesman-25 (Table A.43)
- GAC performs well: jnhSat (Table A.44), jnhUnsat (Table A.45), rand-3-20-20-fcd (Table A.46), rand-3-20-20 (Table A.47), rand-3-24-24-fcd (Table A.48), ogdVg (Table A.49), ukVg (Table A.50), wordsVg (Table A.51).
- Results are inconclusive on: pret (Table A.52), rand-10-20-10 (Table A.53), rand-8-20-5 (Table A.54), varDimacs (Table A.55).

Table A.39: Statistical analysis of the aim-50 benchmark.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
aim-50: 24(24) instances, $e \in$[69,289]							
wR(∗,2)C	[1.3%,5.4%]	850	7	**A**	**24**	7	6086
wR(∗,3)C		5207	0	**A**	**24**	13	2165
wR(∗,4)C		2581	0	**A**	**24**	18	84
GAC	-	779	**8**	**A**	**24**	1	42938
RNIC	[12.5%,16.1%]	418	7	**A**	**24**	**24**	**33**
triRNIC	[34.7%,74.1%]	3844	0	**A**	**24**	**24**	**33**
wRNIC	[1.3%,5.4%]	615	2	**A**	**24**	11	2318
wtriRNIC	[7.8%,15.6%]	6257	0	**A**	**24**	15	709
selRNIC	[1.3%,14.9%]	76	7	**A**	**24**	21	38

Table A.40: Statistical analysis of the dubois benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
dubois: 13(5) instances, e ∈[40,200]							
wR(∗,2)C	[1.5%,7.4%]	2561245	0	**A**	8	0	4875876
wR(∗,3)C		3282055	0	B	7	0	4875876
wR(∗,4)C		3834642	0	B	5	0	2437937
GAC	-	3391539	0	B	6	0	179830778
RNIC	[1.5%,7.4%]	2509636	0	**A**	8	0	4875876
triRNIC / selRNIC	[2%,10%]	**2268505**	**6**	**A**	9	0	**1454897**
wRNIC	[1.5%,7.4%]	2515364	3	**A**	8	0	4875876
wtriRNIC	[2%,10%]	2275396	0	**A**	9	0	**1454897**

Table A.41: Statistical analysis of the ssa benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
ssa: 8(6) instances, e ∈[177,22141]							
wR(∗,2)C	[0%,3%]	677093	1	**A**	7	6	1283
wR(∗,3)C		679968	0	**A**	7	6	1278
wR(∗,4)C		696211	0	**A**	7	6	1278
GAC	-	675408	**5**	**A**	7	6	2048
RNIC	[0.2%,3%]	**44208**	1	**A**	**8**	**7**	1278
triRNIC	[1.7%,83.6%]	952826	0	B	7	6	1197
wRNIC	[0%,3%]	1352081	1	C	6	5	1165
wtriRNIC	[0.1%,9.9%]	1406144	0	C	6	5	**1161**
selRNIC	[0.2%,2%]	291744	1	**A**	**8**	**7**	1197

Table A.42: Statistical analysis of the travellingSalesman-20 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
travellingSalesman-20: 15(0) instances, e ∈[230,230]							
wR(∗,2)C	[1.6%,1.6%]	1368405	0	B	14	**1**	-
wR(∗,3)C		2839793	0	C	10	**1**	-
wR(∗,4)C		3420639	0	C	6	**1**	-
GAC	-	**211698**	**15**	**A**	**15**	**1**	-
RNIC / selRNIC	[14.6%,14.6%]	3502673	0	D	6	**1**	-
triRNIC	[59.8%,59.8%]	-	-	-	0	0	-
wRNIC	[1.6%,1.6%]	1421642	0	B	14	**1**	-
wtriRNIC	[5.1%,5.1%]	4366787	0	E	5	**1**	-

Table A.43: Statistical analysis of the travellingSalesman-25 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
travellingSalesman-25: 15(0) instances, e ∈[350,350]							
wR(∗,2)C	[1.1%,1.1%]	3430739	0	B	6	0	-
wR(∗,3)C		4458430	0	C	5	0	-
wR(∗,4)C		5236257	0	D	1	0	-
GAC	-	**1722287**	**12**	**A**	**12**	0	-
RNIC / selRNIC	[12.4%,12.4%]	-	-	-	0	0	-
triRNIC	[61.7%,61.7%]	-	-	-	0	0	-
wRNIC	[1.1%,1.1%]	3490410	0	B	6	0	-
wtriRNIC	[3.9%,3.9%]	-	-	-	0	0	-

Table A.44: Statistical analysis of the jnhSat benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
jnhSat: 16(0) instances, $e \in$[726,819]							
wR(∗,2)C	[1.4%,1.7%]	350571	0	**A**	**16**	**1**	-
wR(∗,3)C		3766828	0	B	8	**1**	-
wR(∗,4)C		-	-	-	0	0	-
GAC	-	**107405**	**16**	**A**	**16**	**1**	-
RNIC	[23.2%,26.1%]	-	-	-	0	0	-
triRNIC	[87%,90.6%]	-	-	-	0	0	-
wRNIC / selRNIC	[1.4%,1.7%]	4572006	0	C	3	0	-
wtriRNIC	[15.1%,20.8%]	-	-	-	0	0	-

Table A.45: Statistical analysis of the jnhUnsat benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
jnhUnsat: 34(0) instances, $e \in$[714,834]							
wR(∗,2)C	[1.3%,1.5%]	203864	0	**A**	**34**	**2**	-
wR(∗,3)C		4186059	0	B	12	**2**	-
wR(∗,4)C		-	-	-	0	0	-
GAC	-	**45037**	**34**	**A**	**34**	1	-
RNIC	[23.2%,24.2%]	-	-	-	0	0	-
triRNIC	[100%,100%]	-	-	-	0	0	-
wRNIC / selRNIC	[1.3%,1.5%]	4893060	0	C	4	1	-
wtriRNIC	[14.3%,17.8%]	-	-	-	0	0	-

Table A.46: Statistical analysis of the rand-3-20-20-fcd benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
rand-3-20-20-fcd: 50(0) instances, $e \in$[55,60]							
wR(∗,2)C	[5.5%,6.7%]	2932041	1	B	35	0	-
wR(∗,3)C		4921784	0	C	11	0	-
wR(∗,4)C		5366204	0	D	2	0	-
GAC	-	**605024**	**49**	**A**	**50**	0	-
RNIC	[37.7%,43.2%]	-	-	-	0	0	-
triRNIC	[73%,81.6%]	5356726	0	D	3	**3**	-
wRNIC	[5.5%,6.7%]	5119193	0	C	6	0	-
wtriRNIC	[14%,17.4%]	-	-	-	0	0	-
selRNIC	[5.5%,6.7%]	5119193	0	C	6	0	-

Table A.47: Statistical analysis of the rand-3-20-20 benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
rand-3-20-20: 50(0) instances, $e \in$[55,60]							
wR(∗,2)C	[5.5%,6.7%]	3863001	0	B	26	0	-
wR(∗,3)C		5310399	0	C	3	0	-
wR(∗,4)C		-	-	-	0	0	-
GAC	-	**992268**	**49**	**A**	**49**	0	-
RNIC	[37.7%,43.2%]	-	-	-	0	0	-
triRNIC	[73%,81.6%]	5349334	0	C	1	**1**	-
wRNIC	[5.5%,6.7%]	5266419	0	C	3	0	-
wtriRNIC	[14%,17.4%]	-	-	-	0	0	-
selRNIC	[5.5%,6.7%]	5266419	0	C	3	0	-

Table A.48: Statistical analysis of the rand-3-24-24-fcd benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
rand-3-24-24-fcd: 50(0) instances, $e \in$[72,76]							
wR(∗,2)C		5326646	0	B	1	0	-
wR(∗,3)C	[4.5%,5.3%]	-	-	-	0	0	-
wR(∗,4)C		-	-	-	0	0	-
GAC	-	**4784077**	**13**	**A**	**13**	0	-
RNIC	[32.5%,37.6%]	-	-	-	0	0	-
triRNIC	[72.3%,79%]	-	-	-	0	0	-
wRNIC	[4.5%,5.3%]	-	-	-	0	0	-
wtriRNIC	[12.6%,16%]	-	-	-	0	0	-
selRNIC	[4.5%,5.3%]	-	-	-	0	0	-

Table A.49: Statistical analysis of the ogdVg benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
ogdVg: 65(0) instances, $e \in$[8,36]							
wR(∗,2)C		3479105	0	B	24	9	-
wR(∗,3)C	[48.5%,57.1%]	4432513	0	C	14	8	-
wR(∗,4)C		5159987	0	E	4	4	-
GAC	-	**2714970**	**36**	**A**	**36**	**11**	-
RNIC	[48.5%,57.1%]	4277079	0	C	15	9	-
triRNIC	[57.6%,78.6%]	4609824	0	D	11	**11**	-
wRNIC	[48.5%,57.1%]	-	-	-	0	0	-
wtriRNIC / selRNIC	[57.6%,78.6%]	4600512	0	D	11	**11**	-

Table A.50: Statistical analysis of the ukVg benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
ukVg: 65(0) instances, $e \in$[8,36]							
wR(∗,2)C		3564557	0	B	24	4	-
wR(∗,3)C	[48.5%,57.1%]	4482837	0	C	15	4	-
wR(∗,4)C		5185381	0	D	4	4	-
GAC	-	**2814316**	**34**	**A**	**34**	3	-
RNIC	[48.5%,57.1%]	4369614	0	C	15	4	-
triRNIC	[57.6%,78.6%]	4444664	0	C	13	**13**	-
wRNIC	[48.5%,57.1%]	-	-	-	0	0	-
wtriRNIC / selRNIC	[57.6%,78.6%]	4465829	0	C	13	**13**	-

Table A.51: Statistical analysis of the wordsVg benchmark.

Algorithm	d^D	Time	#F	S	#C	#BF	#NV
wordsVg: 65(30) instances, $e \in$[8,36]							
wR(∗,2)C		1461943	1	B	50	24	7
wR(∗,3)C	[48.5%,57.1%]	2344076	0	C	39	24	7
wR(∗,4)C		2788160	0	D	34	32	**2**
GAC	-	**483924**	**63**	**A**	**64**	25	4
RNIC	[48.5%,57.1%]	2292799	0	C	40	24	7
triRNIC	[57.6%,78.6%]	2451473	0	C	39	**39**	**2**
wRNIC	[48.5%,57.1%]	2262265	0	C	40	24	7
wtriRNIC / selRNIC	[57.6%,78.6%]	2475434	0	C	38	38	**2**

Table A.52: Statistical analysis of the pret benchmark.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
pret: 8(4) instances, $e \in$[40,100]							
wR(∗,2)C	[3%,7.7%]	2755360	0	**A**	**4**	**0**	6852604
wR(∗,3)C		2789194	0	**A**	**4**	**0**	3791356
wR(∗,4)C		2918613	0	**A**	**4**	**0**	3791356
GAC	-	2719468	0	**A**	**4**	**0**	12198226
RNIC / selRNIC	[3%,7.7%]	2733806	0	**A**	**4**	**0**	3791356
triRNIC	[6.5%,15.9%]	**2716555**	**4**	**A**	**4**	**0**	**473596**
wRNIC	[3%,7.7%]	2732806	0	**A**	**4**	**0**	3791356
wtriRNIC	[6.5%,15.9%]	2716838	0	**A**	**4**	**0**	**473596**

Table A.53: Statistical analysis of the rand-10-20-10 benchmark.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-10-20-10: 20(12) instances, $e \in$[5,5]							
wR(∗,2)C	[80%,100%]	2160147	1	**A**	**12**	**12**	**0**
wR(∗,3)C		2160132	4	**A**	**12**	**12**	**0**
wR(∗,4)C		**2160126**	**7**	**A**	**12**	**12**	**0**
GAC	-	2165543	**0**	**A**	**12**	0	210
RNIC	[100%,100%]	2160207	0	**A**	**12**	**12**	**0**
triRNIC	[100%,100%]	2160203	0	**A**	**12**	**12**	**0**
wRNIC	[80%,100%]	2160192	0	**A**	**12**	**12**	**0**
wtriRNIC / selRNIC	[80%,100%]	2160841	**0**	**A**	**12**	**12**	**0**

Table A.54: Statistical analysis of the rand-8-20-5 benchmark.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-8-20-5: 20(0) instances, $e \in$[18,18]							
wR(∗,2)C	[47.1%,59.5%]	**2846233**	**12**	**A**	**15**	0	-
wR(∗,3)C		-	-	-	0	0	-
wR(∗,4)C		-	-	-	0	0	-
GAC	-	3494000	4	B	13	0	-
RNIC	[98%,100%]	5287209	0	C	3	3	-
triRNIC	[99.3%,100%]	5262199	0	C	2	2	-
wRNIC	[47.1%,59.5%]	-	-	-	0	0	-
wtriRNIC / selRNIC	[68%,84.3%]	-	-	-	0	0	-

Table A.55: Statistical analysis of the varDimacs benchmark.

Algorithm	d^D	**Time**	**#F**	**S**	**#C**	**#BF**	**#NV**
varDimacs: 9(4) instances, $e \in$[133,1737]							
wR(∗,2)C	[0%,2.9%]	1254343	0	**A**	7	**4**	43257
wR(∗,3)C		1736946	0	B	7	**4**	43257
wR(∗,4)C		2206924	0	C	6	**4**	43257
GAC	-	**739091**	**8**	**A**	**8**	**4**	33469
RNIC	[0.5%,10%]	1859209	0	B	6	3	43298
triRNIC	[1.9%,83.3%]	2617127	0	D	5	2	**8357**
wRNIC	[0%,2.9%]	1876991	0	B	6	2	44475
wtriRNIC	[1.4%,83.3%]	2454323	0	C	5	2	43296
selRNIC	[0.5%,19.9%]	1863079	0	B	6	3	40611

Appendix B

Code Documentation

This documentation gives an overview to the solver used to generate the experimental results. It gives an overview to the data structures and algorithms used. Further, it describes how to install and run the solver.

Below, the additional source files used to implement RNIC to the scsp-code package created by Shant Karakashian are documented. The code repository is located on the Computer Science and Engineering department of the University of Nebraska-Lincoln SVN server located at: `https://cse.unl.edu/svn/scsp`. First we give an overview of the data structures, then the methods added.[1]

B.1 Data Structures

Below is the documentation to the data structures used.

[1]The documentation is generated using Doxygen, `http://www.doxygen.org/`.

B.1.1 constraint_graph Struct Reference

Public Attributes

- constraint * **root**
- int **constraint_node_count**
- int **edge_count**
- int **cgn_i_size**
- **constraint_graph_node** ** **cgn_i**
- **llist** * **node_list**
- **constraint_graph_edge** ** **cge_i**
- **llist** * **edge_list**
- **constraint_graph_edge** *** **matrix**
- **llist** * **left_deep_list**
- int **fill_edges_count**
- int **max_arity**

The documentation for this struct was generated from the following file:

- query_graph.h

B.1.2 constraint_graph_edge Struct Reference

Public Attributes

- int **id**
- **constraint_graph_node** * **node1**
- **constraint_graph_node** * **node2**
- int **weight**

- **htable** ∗ **table**
- q_node ∗ **query_node**
- **llist** ∗ **common_vars**

The documentation for this struct was generated from the following file:

- query_graph.h

B.1.3 constraint_graph_node Struct Reference

Public Attributes

- constraint ∗ **constr**
- **set** ∗ **neighbours**
- int **color**
- char ∗ **name**
- **htable** ∗ **hash_table**
- int **table_changed_after_suspend**
- int **id**
- int **generation**
- int **pgeneration**
- int **combinations_removed**
- **llist** ∗ **children**
- **llist** ∗ **parents**
- int **out_of_order**
- int **peo**
- int **fill_in**

- int **heap_pos**

The documentation for this struct was generated from the following file:

- query_graph.h

B.1.4 htable Struct Reference

Public Attributes

- int **capacity**
- int **total_hight**
- int **last_pos**

  last free position
- int **width**
- int **id**
- int **tmpid**
- char * **name**
- char * **short_name**
- int **is_first**
- int **join_cutoff**
- char ** **col_names**
- **llist** * **col_names_l**
- int * **vars**

  The variables the hashtable is involved in, position 0 is the count.
- int **clustered1**
- int **clustered2**

- int ∗∗ **table**
- **llist** ∗ **tuple_list**
- **llist_node** ∗∗ **tuple_list_array**
- **llist** ∗∗ **tuple_pointed_at_from**
- **llist** ∗∗ **supports**

  list of supports that the tuple supports.

- **llist_node** ∗∗ **deleted_time_node**

  the node in deleted_times lists list that corrosponds to this tuple

- **llist** ∗ **deleted_times**

  list of lists, each list for list of tuples deleted at time t. the ibody is the time

- struct **htable** ∗ **next**
- struct **htable** ∗ **prev**
- struct **htable** ∗ **last**
- struct **histogram** ∗ **histo**
- **llist** ∗ **tree_maps**
- int **complete**
- int **incomplete_i**
- int **incomplete_j**
- int **incomplete_clustereda**
- int **incomplete_clusteredb**
- int **in_tree**
- int **no_destroy**
- constraint ∗ **constr**
- **llist** ∗ **in_combinations**

- **llist** ∗ **in_combinations_hashed**
- **llist** ∗ **index_of_ht_in_combinations**
- int **deleted**
- int **iterator**
- hashmap ∗ **htiterator**
- int ∗ **com_cols_with_next_in_order**
- struct tree_node ∗ **tn**
- struct **constraint_graph_node** ∗ **cgn**
- hashmap ∗ **jthm**
- struct **htable** ∗ **jthmht**
- int ∗ **jtcolpos**
- **llist** ∗∗ **neighbour_list_per_tupple**
- int **context**
- **llist** ∗ **comb_datas**
- int **tuple_lost**
- double ∗ **tuple_values**
- int **problemtable**
- int **flag**
- int **markgeneration**
- struct **tuple_tag** ∗∗ **table_tags**

  table tags that RNIC uses, once for each tuple (corresponds to the same entries in the table)

- **set** ∗ **needs_tags_support**

  The queue of tuples that need support.

- **llist * clusters**

    Clusters that the hash table is involved with.

The documentation for this struct was generated from the following file:

- hashtable.h

B.1.5 light_stack Struct Reference

Public Attributes

- struct **light_stack_node * head**

The documentation for this struct was generated from the following file:

- llist.h

B.1.6 light_stack_node Struct Reference

Public Attributes

- int **value**
- void * **body**
- struct **light_stack_node * next**

The documentation for this struct was generated from the following file:

- llist.h

B.1.7 llist Struct Reference

Stores the llist.

Public Attributes

- struct **llist_node** * **head**

  The head of the llist.

- struct **llist_node** * **tail**

  The tail of the llist.

- struct **llist_node** * **min**
- struct **llist_node** * **max**
- int **id**

  The ID of the list.

- int **count**

  How many elements are in the list.

B.1.7.1 Detailed Description

Stores the llist.

The documentation for this struct was generated from the following file:

- llist.h

B.1.8 llist_node Struct Reference

Stores an element in the llist.

Public Attributes

- void * **body**

  The body of the node.

- int **ibody**

The integer value of the node.

- struct **llist_node** * **next**

  The next element in the list (NULL if none)

- struct **llist_node** * **previous**

  The previous element in the list (NULL if none)

- struct **llist** * **list**

  A pointer back to the list where this node appears.

B.1.8.1 Detailed Description

Stores an element in the llist.

The documentation for this struct was generated from the following file:

- llist.h

B.1.9 s_node Struct Reference

Public Attributes

- int **key**
- void * **body**

The documentation for this struct was generated from the following file:

- set.h

B.1.10 set Struct Reference

Public Attributes

- **llist** * **list**

- char * **map**
- **llist_node** ** **element_ptrs**
- int **last_removed**
- int **size**

The documentation for this struct was generated from the following file:

- set.h

B.1.11 tree_map Struct Reference

Public Attributes

- **llist** * **indexed_cols**
- **htable** * **htto**
- **llist** * **lists**
- int **id**
- int **count**
- int **deleted**

The documentation for this struct was generated from the following file:

- tree_map.h

B.1.12 tree_map_node Struct Reference

Public Attributes

- int **id**
- **llist** * **nexts**

- int **full**
- int **object_id**

The documentation for this struct was generated from the following file:

- tree_map.h

B.1.13 tuple_tag Struct Reference

Public Attributes

- int **id**

  Id.

- **htable** * **hash_table**

  Store the tuple and the relation.

- int **location**

  Store the tuple location.

- struct **tuple_tag** ** **supports**

  List of supports that are currently being used by this tuple (In the same order as the neighbors of the node)

- **llist** * **supported_by**

  List of tuples that are supported by this tuple.

- **llist** *** **pair_common_cols**

  List of common columns.

- **llist** ** **this_common_cols**

  List of common columns with this tuple.

- int ** **pair_common_cols_twisted**

If the pair common columns are twisted (If the [0] and the [1] need to be reversed)

- int * **this_common_cols_twisted**

  If the pair common columns are twisted with this tuple (If the [0] and the [1] need to be reversed)

- **tree_map** *** **pair_tms**

  Trees maps between two nodes.

- **tree_map** ** **this_tms**

  Trees maps with this tuple.

- **llist** * **neighbours**

  A list of the neighbours of this tuple. (Those that are within the distance requirement). This is a list of constraint_graph_edge's.

- **llist** * **nodes**

  The set of nodes that this tuple has.

- int * **deg_count**

  The degree count for all of the nodes.

- **llist** * **futureVars**

  List of future vars (Should always be set the the list of nodes...)

- int **visitCount**

  How many times this tuple has been visited.

B.1.13.1 Detailed Description

The data to track for a tuple

The documentation for this struct was generated from the following file:

- **nic.h**

B.2 File Documentation

Below is the documentation for the two C files added to the scsp-code package.

B.2.1 nic2.c File Reference

Functions

- **llist** ** **iNeighborsFilterAll** (**constraint_graph** *cg, int distance, float filter_cutoff)
- **llist** * **iNeighborsFilter** (**constraint_graph_node** *cgn, int distance, float filter_cutoff, **constraint_graph** *cg)
- void **iNeighborsFilterLevelParent** (**llist** **nodesByLevel, int currentDistance, **llist** *returnNodes, int maxSize, **constraint_graph_node** *child)
- void **iNeighborsFilterLevelSingle** (**llist** **nodesByLevel, int currentDistance, **llist** *returnNodes, int maxSize, **constraint_graph_node** *child)
- **llist** ** **iNeighborsAll** (**constraint_graph** *cg, int distance)
- **llist** * **iNeighbors** (**constraint_graph_node** *cgn, int distance, **constraint_graph** *cg)
- void **destroyTupleTagsTri** (**constraint_graph** *cg)
- int **createTupleTagsTri** (**constraint_graph** *cg)
- **tuple_tag** * **createTupleTag** (**constraint_graph_node** *cgn, int location, **llist** ***pair_common_cols, int **pair_common_cols_twisted, **llist** **this_common_cols, int *this_common_cols_twisted, **tree_map** ***pair_tms, **tree_map** **this_tms, **llist** *neighbours, **llist** *nodes, **llist** *futureVars, int *deg_count)
- int **filterNICTri** (**set** *lost_support_set, int num_tuples, int time, **set** *undo_set)

- int **filterNICTriRelation** (**set** ∗relationQueue, int time, **set** ∗undo_set, int useCluster)
- int **filterNICTriCluster** (**set** ∗lost_support_set, int num_tuples, int time, **set** ∗undo_set)
- int **filterNICTriClusterTree** (**set** ∗lost_support_set, int num_tuples, int time, **set** ∗undo_set)
- int **find_supports_tuple7Set** (**tuple_tag** ∗tuple)
- void **cleanup_test** (**tuple_tag** ∗tuple, **llist_node** ∗curVar, int searchDone)
- int **find_supports_label7Set** (**tuple_tag** ∗startTuple, int curLoc, **llist_node** ∗curVar, **set** ∗∗∗tuples, **llist** ∗futureVars, **llist** ∗∗pastVars, **llist** ∗∗undoDeg)
- int **tuple_tag_forwardcheck5Set** (**tuple_tag** ∗startTuple, int curLoc, **llist_node** ∗curVar, int curTuple, **set** ∗∗∗tuples, **llist** ∗futureVars)
- int **find_supports_unlabel7Set** (**tuple_tag** ∗startTuple, int curLoc, **llist_node** ∗curVar, **llist** ∗pastVars, **set** ∗∗∗tuples, int lastLoc)
- int **common_cols_ordered** (**llist** ∗common_cols, int twisted)
- **llist_node** ∗ **nic_choose_varSet** (**tuple_tag** ∗tuple, **llist** ∗remaining_vars, **llist** ∗undo_deg, int curLevel, **set** ∗∗∗tuples)
- **llist_node** ∗ **nic_unchoose_var7** (**tuple_tag** ∗tuple, **llist** ∗pastVars, **llist** ∗undo_deg)
- void **neatoNeighborhood** (**llist** ∗∗nodes, int number_nodes)
- void **neatoNeighborhoodDecomp** (decomposed_tree ∗root)
- void **neatoNeighborhoodDecompNode** (FILE ∗f, decomposed_tree ∗root)
- void **print_supported_by** (**tuple_tag** ∗tt)
- void **print_supports** (**tuple_tag** ∗tt)
- void **nic_validate_tuples** (int checkSupportedBy)

- void **nic_validate_deg** (void)
- void **nic_validate_deg_for_tuple** (**tuple_tag** *tuple)
- **llist** ** **nic_get_dual_elimination_ordering** (**constraint_graph** *cg)
- **llist** ** **nic_triangulate_min_deg** (**constraint_graph** *cg)
- **llist** ** **nic_max_cardinality** (**constraint_graph** *cg)
- int **nic_count_dual_fill_edges** (char **matrix, **constraint_graph_node** *node, int max_nodes, char **matrix_pre_rr)
- void **nic_add_dual_fill_edges** (char **matrix, **constraint_graph_node** *node, heap *h, **llist** **allNeighbors, int max_nodes, char **matrix_pre_rr)
- **llist** * **nic_dual_cliques** (**llist** **ordering, **constraint_graph** *cg)
- decomposed_tree * **nic_build_decomposed_tree_vertices** (**llist** *cliques, **llist** *tree_nodes, int constraint_count, **constraint_graph** *cg)
- int **nic_find_danglesSet** (**tuple_tag** *tuple, **llist_node** *curVar, int curVarTuple, **llist** *futureVars, **llist** *pastVars, int curLevel, **set** ***tuples, **llist** *undo_deg, **llist** *neighborsList)
- void **findDangleSupportsSet** (**tuple_tag** *tuple, **llist** **prevVars, **llist** **neighborsList, **set** ***tuples, int lastAssign)
- int **nic_reviseSet** (**tuple_tag** *startTuple, int curLoc, **llist_node** *curVar, int curTuple, **set** ***tuples, **llist_node** *otherVar)
- void **sortNodesBySize** (**llist** *nodes)
- int **copyAliveTuplesOriginal** (int location, int variable)
- int **copyAliveTuplesOriginal_exclude** (int location, int variable, int excludeLocation)
- void **requeue_tuple_cluster** (**tuple_tag** *tuple, **set** *needsClusterSupports)
- void **requeue_tuple_clusterTree** (**tuple_tag** *tuple, **set** *needsClusterSupports)

- **llist_node** ∗∗ **computeVarShallowestClique** (**llist** ∗cliques)
- **llist** ∗∗∗ **computerClusterNeighborhoods** (**llist** ∗dualCliques, **llist** ∗∗allNodes)
- **llist** ∗ **flattenClusters** (**llist** ∗dualCliques)
- void **print_filtering** (int status)
- void **print_filterRel** (void)
- void **print_tupleVisitCount** (void)
- void **checkFutureTuples** (**tuple_tag** ∗startTuple, **set** ∗∗∗tuples, **set** ∗∗∗aliveTuplesSet_original, int lastLoc)
- void **print_csp_info** (void)

Variables

- **set** ∗∗∗ **aliveTuplesSet** = NULL
- **set** ∗∗∗ **aliveTuplesSet_original** = NULL
- **llist** ∗∗ **nicUndoDeg** = NULL
- int **aliveTuplesCount** = 0
- **llist** ∗∗ **nicPastVars7** = NULL
- **llist** ∗ **nicPEO** = NULL
- **llist** ∗∗ **neighborsList** = NULL
- **llist** ∗ **cliquesOrdering** = NULL
- **llist_node** ∗∗ **varShallowCliques** = NULL
- **set** ∗∗ **varCliques** = NULL
- **llist** ∗∗∗ **clusterNeighborhoods** = NULL
- int **tupleTag_ids** = 0

- **llist** $*$ **pastVarsTri** = NULL
- **llist** $*$ **undoDegTri** = NULL
- **set** $*$ **needsSupports6** = NULL
- int **totalFiltered** = 0
- int **totalLookaheadFiltered** = 0
- int **totalAssignedFiltered** = 0
- int $*$ **numberFiltered** = NULL
- int $*$ **numberFilteredRel** = NULL
- **set** $*$ **filteredRelSet** = NULL
- int **nic_relationLoopCount** = 0
- **set** $*$ **needsClusterSupports** = NULL
- **set** $**$ **clusterQueues** = NULL
- int **nic_clusterLoopCount** = 0
- **llist** $**$ **nic_allNeighbors** = NULL

B.2.1.1 Detailed Description

This file contains most of the code required to enforce Relational Neighborhood Inverse Consistency (RNIC).

B.2.1.2 Function Documentation

B.2.1.3 int common_cols_ordered (llist * *common_cols,* int *twisted*)

Check to see if the common columns are ordered correctly, give if they are twisted or not

Parameters

The	common columns to check
If	the common columns are twisted or not

Returns

If the common columns are ordered correctly

B.2.1.4 llist_node** computeVarShallowestClique (llist * *cliques*)

Computes the shallowest clique that a variable appears in (The shallowest relation that has a variable in its scope, and the queue that it appears in).

Parameters

cliques	The list of cliques

Returns

The shallowest cliques that each variable appears in

B.2.1.5 tuple_tag* createTupleTag (constraint_graph_node * *cgn*, int *location*, llist *** *pair_common_cols*, int ** *pair_common_cols_twisted*, llist ** *this_common_cols*, int * *this_common_cols_twisted*, tree_map *** *pair_tms*, tree_map ** *this_tms*, llist * *neighbours*, llist * *nodes*, llist * *futureVars*, int * *deg_count*)

Creates the tuple data structures for an individual tuple

Parameters

cgn	The relation that this tuple is in
location	The location of the tuple in the table of supports
pair_common_cols	Pointers to the list of the common scopes between every two relations in the neighborhood
pair_common_cols_twisted	The order of the common scopes for pair_common_cols
this_common_cols	Pointers to the list of common scpoes between this relation and all the relations in the neighborhood
this_common_cols_twisted	The order of the common scopes for this_common_cols
pair_tms	The index-tree data structures for each relation in the neighborhood
this_tms	The index-tree data structures for this relation
nodes	The list of the neighborhood
futureVars	A list of the future variables to search on
deg_count	A list of the degree of each relation in the induced subproblem of this relation and its neighborhood

Returns

The data structure used to store the tuples

B.2.1.6 int createTupleTagsTri (constraint_graph * *cg*)

Creates the data structures needed to enforce RNIC. If set, it will also triangulate the neighborhoods.

Parameters

cg	The constraint graph (dual graph) to work on

Returns

The total number of tuples in the dual graph.

B.2.1.7 void destroyTupleTagsTri (constraint_graph * *cg*)

Destroys all of the data structures that RNIC used

Parameters

cg	The constraint graph (dual graph) to work on

B.2.1.8 int filterNICTri (set * *lost_support_set*, int *num_tuples*, int *time*, set * *undo_set*)

Runs RNIC on the problem

Parameters

lost_support-_set	Record all the tuples that lost a support
num_tuples	The total number of tuples
time	The current label in search to label when tuples are deleted
undoSet	A set of relations that were modified

Returns

If the CSP is consistent or not after enforcing RNIC

B.2.1.9 int filterNICTriCluster (set ∗ *lost_support_set*, int *num_tuples*, int *time*, set ∗ *undo_set*)

Process the cluster queue of RNIC using the maximal cliques (clusters)

Parameters

lost_support-_set	Record all the tuples that lost a support
num_tuples	The total number of tuples
time	The current label in search to label when tuples are deleted
undoSet	A set of relations that were modified

Returns

If the CSP is consistent or not after enforcing RNIC

B.2.1.10 int filterNICTriClusterTree (set * ***lost_support_set,*** int ***num_tuples,*** int ***time,*** set * ***undo_set***)

Process the cluster queue of RNIC using the tree decomposition

Parameters

lost_support-_set	Record all the tuples that lost a support
num_tuples	The total number of tuples
time	The current label in search to label when tuples are deleted
undoSet	A set of relations that were modified

Returns

If the CSP is consistent or not after enforcing RNIC

B.2.1.11 int filterNICTriRelation (set * ***relationQueue,*** int ***time,*** set * ***undo_set,*** int ***useCluster***)

Process the relation queue of RNIC

Parameters

relation-Queue	The queue of relations to process

time	The current label in search to label when tuples are deleted
undoSet	A set of relations that were modified
useCluster	If the relationQueue is for a specific cluster or not

Returns

If the CSP is consistent or not after enforcing RNIC

B.2.1.12 int find_supports_label7Set (tuple_tag * *startTuple*, int *curLoc*, llist_node * *curVar*, set *** *tuples*, llist * *futureVars*, llist ** *pastVars*, llist ** *undoDeg*)

The label procedure of a back-track search on the neighborhood of a tuple to see if it has a valid support

Parameters

startTuple	The tuple the search is being conducted on
curLoc	The current level of the search
curVar	The current dual variable being instantiated
tuples	The set of tuples still alive in the neighborhood
futureVars	The list of future variables yet to be instantiated
pastVars	The list of past variables already instantiated
undoDeg	List of changes to the degree of the variables after instantiation, used to undo during backtrack.

Returns

If the current variable has a consistent assignment

B.2.1.13 int find_supports_tuple7Set (tuple_tag * *tuple*)

Conducts a back-track search on the neighborhood of a tuple to see if it has a valid support

Parameters

tuple	The tuple to check if it has a support

Returns

If the tuple has a support in its neighborhood

B.2.1.14 int find_supports_unlabel7Set (tuple_tag * *startTuple*, int *curLoc*, llist_node * *curVar*, llist * *pastVars*, set *** *tuples*, int *lastLoc*)

The unlabel procedure of a back-track search on the neighborhood of a tuple to see if it has a valid support

Parameters

startTuple	The tuple the search is being conducted on
curLoc	The current level of the search
curVar	The current dual variable being instantiated
pastVars	The list of past variables already instantiated

tuples	The set of tuples still alive in the neighborhood
lastLoc	The last location where we did a label/unlabel

Returns

If the current variable has a consistent assignment

B.2.1.15 void findDangleSupportsSet (tuple_tag * *tuple,* llist ** *prevVars,* llist ** *neighborsList,* set *** *tuples,* int *lastAssign*)

After search, assigns a single support to each of the dangles

Parameters

tuple	The tuple the search is being conducted on
pastVars	The list of past variables already instantiated
neighbors-List	The neighbors for the dangles that were discovered
tuples	The set of tuples still alive in the neighborhood
lastAssign	The last level where an assignment took place

B.2.1.16 llist* flattenClusters (llist * *dualCliques*)

Flattens the relations inside of the cliques into a linear ordering

Parameters

dualCliques	All of the cliques of the dual graph
A	list of the flattened relations

B.2.1.17 llist* iNeighbors (constraint_graph_node * *cgn*, int *distance*, constraint_graph * *cg*)

Gets the neighborhood of a vertex (relation) in the dual graph

Parameters

cgn	The vertex (relation) to find the neighborhood
distnace	The neighborhood distance (1=its immediate neighborhood, 2=include all of the neighbors of the neighbors, etc.)
cg	The constraint graph (dual graph) to work on

Returns

A llist of all of the neighbors of the vertex. The first node in the list is the original relation

B.2.1.18 llist** iNeighborsAll (constraint_graph * *cg*, int *distance*)

Gets the neighborhood of all the vertices (relations) in the dual graph

Parameters

cg	The constraint graph (dual graph) to work on
distnace	The neighborhood distance (1=its immediate neighborhood, 2=include all of the neighbors of the neighbors, etc.)

Returns

A llist of all of the neighbors for each relation. The first node in each list is the original relation

B.2.1.19 llist∗ iNeighborsFilter (constraint_graph_node ∗ *cgn,* int *distance,* float *filter_cutoff,* constraint_graph ∗ *cg*)

Gets the neighborhood of a vertex (relation) in the dual graph

Parameters

cgn	The vertex (relation) to find the neighborhood
distnace	The neighborhood distance (1=its immediate neighborhood, 2=include all of the neighbors of the neighbors, etc.)
filter_cutoff	Determine how many neighbors each vertex has, and take the top filter_cutoff percent (As a decimal percent: between 0-1)
cg	The constraint graph (dual graph) to work on

Returns

A llist of all of the neighbors of the vertex. The first node in the list is the original relation

B.2.1.20 llist** iNeighborsFilterAll (constraint_graph * *cg,* int *distance,* float *filter_cutoff*)

Gets the neighborhood of all the vertices (relations) in the dual graph

Parameters

cg	The constraint graph (dual graph) to work on
distnace	The neighborhood distance (1=its immediate neighborhood, 2=include all of the neighbors of the neighbors, etc.)
filter_cutoff	Determine how many neighbors each vertex has, and take the top filter_cutoff percent (As a decimal percent: between 0-1)

Returns

A llist of all of the neighbors for each relation. The first node in each list is the original relation

B.2.1.21 void neatoNeighborhood (llist ** *nodes,* int *number_nodes*)

Writes to the file 'neighborhood.neato' in the current directory, which constructs the dual graph of the CSP. The file can be converted to a PDF using 'neato -Tpdf -o neighborhood.pdf neighborhood.neato'

Parameters

nodes	The list of neighborhoods for all of the variables
number_-nodes	The number of dual variables in the problem

B.2.1.22 void neatoNeighborhoodDecomp (decomposed_tree * *root*)

Writes to the file 'neighborhood_decomp.dot' in the current directory, which constructs the tree decomposition of the dual graph. The file can be converted to a PDF using 'dot -Tpdf -o neighborhood_decomp.pdf neighborhood_decomp.dot'

Parameters

root	The root of the tree

B.2.1.23 void neatoNeighborhoodDecompNode (FILE * *f*, decomposed_tree * *root*)

Writes to a file the nodes and connections of a single vertex of the tree decomposition.

Parameters

f	The file to write to
root	The root of the tree

B.2.1.24 void nic_add_dual_fill_edges (char ** *matrix*, constraint_graph_node * *node*, heap * *h*, llist ** *allNeighbors*, int *max_nodes*, char ** *matrix_pre_rr*)

Adds the fill-in edges to a dual variable.

Parameters

matrix	A matrix representation of the relations
node	The node to add the fill-in edges for
h	A heap ordering of the relations, sorted by the number of fill in edges
all-Neighborhoos	A list of the current neighborhoods of all of the dual varaibles
max_nodes	The maximum number of nodes in the dual graph
matrix_pre_-rr	The matrix representation of the relations prior to redundancy removal

B.2.1.25 decomposed_tree* nic_build_decomposed_tree_vertices (llist * *cliques*, llist * *tree_nodes*, int *constraint_count*, constraint_graph * *cg*)

Given a set of cliques, builds the tree decomposition

See also

build_decomposed_tree_vertices

Parameters

cliques	The set of cliques
tree_nodes	A linked list where the tree nodes are stored
constraint_-count	The number of constraints
cg	The constraint graph (dual graph) to work on.

Returns

The root of the tree

B.2.1.26 llist_node* nic_choose_varSet (tuple_tag * *tuple,* llist * *remaining_vars,* llist * *undo_deg,* int *curLevel,* set *** *tuples*)

Chooses the next dual variable to assign using a deg/domain heuristic

Parameters

	tuple	The tuple the search is being conducted on
	remaining_-vars	The remaining variables to be instantiated
`out`	*undo_deg*	The current level's undo degree, for fast re-assignment during an unlabel
	curLevel	The current level of the search
	tuples	The set of tuples still alive in the neighborhood

Returns

The next variable to assign

B.2.1.27 int nic_count_dual_fill_edges (char ** *matrix*, constraint_graph_node * *node*, int *max_nodes*, char ** *matrix_pre_rr*)

Counts the number of fill-in edges for a dual variable.

Parameters

matrix	A matrix representation of the relations
node	The node to count the fill-in edges for
max_nodes	The maximum number of nodes in the dual graph
matrix_pre_rr	The matrix representation of the relations prior to redundancy removal

Returns

The number of fill in edges

B.2.1.28 llist* nic_dual_cliques (llist ** *ordering*, constraint_graph * *cg*)

Given a perfect eliminiation ordering, computes the maximal cliques of the dual graph.

Parameters

ordering	A list of the nodes neighborhoods (Where the node is the first entry in the list), sorted in the perfect elimination ordering.
cg	The constraint graph (dual graph) to work on.

Returns

The maximal cliques of the dual graph

B.2.1.29 int nic_find_danglesSet (tuple_tag * *tuple,* llist_node * *curVar,* int *curVarTuple,* llist * *futureVars,* llist * *pastVars,* int *curLevel,* set *** *tuples,* llist * *undo_deg,* llist * *neighborsList*)

Finds the dangles and applies directional arc conssitency (2-wise-consistency) to update the domains (relations)

Parameters

tuple	The tuple the search is being conducted on
curVar	The current dual variable being instantiated
curVarTuple	The value that is being instantiated to the current dual variable
futureVars	The list of future variables yet to be instantiated
pastVars	The list of past variables already instantiated
curLevel	The current level of the search
tuples	The set of tuples still alive in the neighborhood

undoDeg	List of changes to the degree of the variables after instantiation, used to undo during backtrack.
neighbors-List	The alive neighbors for the dangles that were discovered

Returns

If the problem is consistent or not

B.2.1.30 llist** nic_get_dual_elimination_ordering (constraint_graph * *cg*)

Gets the elimination ordering of the dual graph (triangulates the dual graph). This method uses the minFill triangulation method.

Parameters

cg	The constraint graph (dual graph) to work on

Returns

A list of all of the nodes with their triangulated neighborhood. The first node in the list is the node whose neighborhood it is.

B.2.1.31 llist** nic_max_cardinality (constraint_graph * *cg*)

Gets the max cardinality ordering of a triangulated dual graph.

Parameters

cg	The constraint graph (dual graph) to work on

Returns

A list of all of the nodes with their triangulated neighborhood in the max cardinality ordering. The first node in the list is the node whose neighborhood it is.

B.2.1.32 int nic_reviseSet (tuple_tag * *startTuple*, int *curLoc*, llist_node * *curVar*, int *curTuple*, set *** *tuples*, llist_node * *otherVar*)

Revies another dual variable based on the current dual variable

Parameters

startTuple	The tuple the search is being conducted on
curLoc	The current level of the search
curVar	The current dual variable being instantiated
curTuple	The current tuple assigned to the current variable
tuples	The set of tuples still alive in the neighborhood
otherVar	The other dual variable, to be revised

Returns

If the problem is consistent (the other dual variable has tuples)

B.2.1.33 llist** nic_triangulate_min_deg (constraint_graph * *cg*)

Gets the elimination ordering of the dual graph (triangulates the dual graph). This method uses the minimum degree heuristic (Take the node with the minimum degree, breaking ties by the minimum number of fill-in edges).

Parameters

cg	The constraint graph (dual graph) to work on

Returns

A list of all of the nodes with their triangulated neighborhood. The first node in the list is the node whose neighborhood it is.

B.2.1.34 llist_node* nic_unchoose_var7 (tuple_tag * *tuple*, llist * *pastVars*, llist * *undo_deg*)

Un-chooses the a dual variable (Placing it back in the future variables and restoring the degree)

Parameters

tuple	The tuple the search is being conducted on
pastVars	The past, already instantiated, variables
undo_deg	The current level's undo degree, for fast re-assignment

Returns

The next variable to assign

B.2.1.35 void nic_validate_deg (void)

Validates that the degree of each tuple is set correctly

B.2.1.36 void nic_validate_deg_for_tuple (tuple_tag * *tuple*)

Validates that the degree for an individual tuple.

B.2.1.37 void nic_validate_tuples (int *checkSupportedBy*)

Validates that all of the supports in all the tuples are there

Parameters

checkSupportedBy	If the supportedBy structure should also be checked

B.2.1.38 void print_csp_info (void)

Prints to stdout information about the CSP (about clique size and dual degree).

B.2.1.39 void print_supported_by (tuple_tag * *tt*)

Prints to stdout all of the supported_by elements of a tuple

Parameters

tt	The tuple to print

B.2.1.40 void print_supports (tuple_tag * *tt*)

Prints to stdout all of the supports of a tuple

Parameters

tt	The tuple to print

B.2.1.41 void print_tupleVisitCount (void)

Prints to stdout information about the the number of tuples visisted/loops taken in RNIC.

B.2.1.42 void requeue_tuple_cluster (tuple_tag * *tuple,* set * *needsClusterSupports*)

Requeues everything that used a tuple as a support into the cluster queues

Parameters

tuple	The tuple that was deleted
needs-Cluster-Supports	The queue of all the clusters

B.2.1.43 void requeue_tuple_clusterTree (tuple_tag * *tuple*, set * *needsClusterSupports*)

Requeues everything that used a tuple as a support into the tree decomposition cluster queues

Parameters

tuple	The tuple that was deleted
needsClusterSupports	The queue of all the clusters

B.2.1.44 void sortNodesBySize (llist * *nodes*)

Sort the neighborhoods of a relation by their tuple size, to save on space

Parameters

nodes	The neighborhood nodes of a relation (The relation is the first element in the list)

B.2.1.45 int tuple_tag_forwardcheck5Set (tuple_tag * *startTuple*, int *curLoc*, llist_node * *curVar*, int *curTuple*, set *** *tuples*, llist * *futureVars*)

Forward checks a the future variables to see if the current partial assignment is consistent

Parameters

startTuple	The tuple the search is being conducted on
curLoc	The current level of the search
curVar	The current dual variable being instantiated
curTuple	The tuple trying to be instantiated to the current variable
tuples	The set of tuples still alive in the neighborhood
futureVars	The list of future variables yet to be instantiated

Returns

If the current variable with the current tuple is a consistent assignment

B.2.2 nicprocedures.c File Reference

Functions

- int **nic_bcssp** (int ccp_count)
- variable * **nic_label** (variable *var_i, int *consistant, int time, int ccp_count)
- variable * **nic_unlabel** (variable *var_i, int *consistant, **main_structure** *m_s)
- int **nic_forward_check** (variable *var, int val, int time, int ccp_count)
- void **nic_undo_reductions** (variable *var)
- variable * **nic_choose_variable** (**main_structure** *m_s)

B.2.2.1 Detailed Description

This file contains the backtrack search algorithm that uses RNIC as lookahead.

B.2.2.2 Function Documentation

B.2.2.3 int nic_bcssp (int *ccp_count*)

The backtrack search procedure, which enforces RNIC as lookahead

Parameters

ccp_count	The total number of tuples

Returns

The state after search

B.2.2.4 variable* nic_choose_variable (main_structure * *m_s*)

A special variable selection that uses the instanation ordering returned from the perfect eliminiation ordering

Parameters

m_s	the main structure about the CSP

Returns

The next variable to instantiate

B.2.2.5 int nic_forward_check (variable * *var*, int *val*, int *time*, int *ccp_count*)

The forward check procedure for the backtrack search procedure, which enforces R-NIC as lookahead

Parameters

var	The variable to that we are instantiating
val	The value to try instantiating the variable to
time	The time to label removed tuples
ccp_count	The total number of tuples

Returns

If the CSP is consistent or not with that instantiation

B.2.2.6 variable* nic_label (variable * *var_i,* int * *consistant,* int *time,* int *ccp_count*)

The label procedure for the backtrack search procedure, which enforces RNIC as lookahead

Parameters

var_i	The variable to instantiate
consistant	If the CSP is consistent or not
time	The time to label removed tuples
ccp_count	The total number of tuples

Returns

The next variable to label if consistent, else the current variable

B.2.2.7 void nic_undo_reductions (variable * *var*)

The undo reductions procedure for the backtrack search procedure, which enforces RNIC as lookahead. This will undo the effect of the forward checking

Parameters

var	What variable's effects to undo

B.2.2.8 variable* nic_unlabel (variable * *var_i*, int * *consistant*, main_structure * *m_s*)

The unlabel procedure for the backtrack search procedure, which enforces RNIC as lookahead

Parameters

var_i	The variable to uninstantiate
consistant	If the CSP is consistent or not
m_s	The main structure that holds information about the CSP

Returns

The next variable

Bibliography

[Bacchus *et al.*, 2002] Fahiem Bacchus, Xinguang Chen, Peter Van Beek, and Toby Walsh. Binary vs. Non-Binary Constraints. *Artificial Intelligence*, 140:1–37, 2002.

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

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

[Bessiere *et al.*, 2011] Christian Bessiere, Stéphane Cardon, Romuald Debruyne, and Christophe Lecoutre. Efficient Algorithms for Singleton Arc Consistency. *Constraints*, 16 (1):25–53, 2011.

[Bessiere, 2006] Christian Bessiere. *Handbook of Constraint Programming*, chapter Constraint Propagation. Elsevier, 2006.

[Cheng and Yap, 2010] Kenil C.K. Cheng and Roland H.C. Yap. An MDD-Based Generalized Arc Consistency Algorithm for Positive and Negative Table Constraints and Some Global Constraints. *Constraints*, 15 (2):265–304, 2010.

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

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

[Dechter and Pearl, 1987a] Rina Dechter and Judea Pearl. Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence*, 34:1–38, 1987.

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

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

[Dechter and van Beek, 1997] Rina Dechter and Peter van Beek. Local and Global Relational Consistency. *Theoretical Computer Science*, 173(1):283–308, 1997.

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

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

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

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

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

[Gavril, 1972] Fanica Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM J. Comput.*, 1 (2):180–187, 1972.

[Gent *et al.*, 2000] Ian Gent, Kostas Stergiou, and Toby Walsh. Decomposable Constraints. *Artificial Intelligence*, 123 (1-2):133–156, 2000.

[Golumbic, 2004] Martin C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Elsevier, 2004. Annals of Discrete Mathematics, Vol 75.

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

[Janssen *et al.*, 1989] Philippe Janssen, Philippe Jégou, B. 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.

[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 *24th AAAI Conference on Artificial Intelligence (AAAI 10)*, pages 101–107, 2010.

[Kjærulff, 1990] Uffe Kjærulff. Triagulation of Graphs - Algorithms Giving Small Total State Space. Research Report R-90-09, Aalborg University, Denmark, 1990.

[Lecoutre and Prosser, 2006] Christophe Lecoutre and Patrick Prosser. Maintaining Singleton Arc Consistency. In *CPAI 06 Workshop on Symmetry in Constraint Satisfaction Problems (SymCon 10)*, pages 47–61, 2006.

[Lecoutre, 2010] Christophe Lecoutre. *Constraint Networks: Techniques and Algorithms*. Wiley, 2010.

[Lee, 1992] Elisa T. Lee. *Statistical Methods for Survival Data Analysis*. John Wiley & Sons, New York, NY, second edition, 1992.

[Lim, 2006] Ryan Way Hoong Lim. GTAAP: An Online System For Managing and Assigning Graduate Teaching Assistants to Academic Tasks. Master's Project. Department of Computer Science & Engineering, University of Nebraska-Lincoln, 2006.

[Luchtel, 2011] Keith Luchtel. Personal communication, 2011.

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

[Mohr and Masini, 1988] Roger Mohr and Gérald Masini. Good Old Discrete Relaxation. In *European Conference on Artificial Intelligence (ECAI-88)*, pages 651–656, Munich, W. Germany, 1988.

[Revesz, 2001] Peter Revesz. *Introduction to Constraint Databases*. Springer-Verlag, New York, 2001.

[Stergiou, 2007] Kostas Stergiou. Strong Inverse Consistencies for Non-Binary CSPs. In *Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence*, volume 1 of *ICTAI 07*, pages 215–222, 2007.

[Tarjan and Yannakakis, 1984] Robert Endre Tarjan and Mihalis Yannakakis. Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, 1984.

[Woodward *et al.*, 2011a] Robert Woodward, Shant Karakashian, Berthe Y. Choueiry, and Christian Bessiere. Adaptive Neighborhood Inverse Consistency as Lookahead for Non-Binary CSPs. In *25th AAAI Conference on Artificial Intelligence (AAAI 11)*, pages 1–2, 2011.

[Woodward *et al.*, 2011b] Robert Woodward, Shant Karakashian, Berthe Y. Choueiry, and Christian Bessiere. Solving Difficult CSPs with Relational Neighborhood Inverse Consistency. In *25th AAAI Conference on Artificial Intelligence (AAAI 11)*, pages 1–8, 2011.

[Woodward *et al.*, 2011c] Robert J. Woodward, Shant Karakashian, Berthe Y. Choueiry, and Christian Bessiere. Reformulating the Dual Graphs of CSPs to Improve the Performance of Relational Neighborhood Inverse Consistency. In *Ninth International Symposium on Abstraction, Reformulation and Approximation (SARA 2011)*, pages 1–8. AAAI Press, 2011.

[Woodward *et al.*, 2011d] Robert J. Woodward, Shant Karakashian, Berthe Y. Choueiry, and Christian Bessiere. Relational Neighborhood Inverse Consistency for Constraint Satisfaction. Technical Report TR-UNL-CSE-2011-0007, Constraint Systems Laboratory, University of Nebraska-Lincoln, Lincoln, NE, 2011.

[Zabih, 1990] Ramin Zabih. Some Applications of Graph Bandwidth to Constraint Satisfaction Problems. In *Proceedings of AAAI-90*, pages 46–51, Boston, MA, 1990.