Spring 5-3-2013

Practical Tractability of CSPS by Higher Level Consistency and Tree Decomposition

Shant Karakashian
University of Nebraska-Lincoln, shantgk@gmail.com

PRACTICAL TRACTABILITY OF CSPS BY HIGHER LEVEL CONSISTENCY AND TREE DECOMPOSITION

by

Shant Karakashian

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Doctor of Philosophy

Major: Computer Science

Under the Supervision of Professor Berthe Y. Choueiry

Lincoln, Nebraska

May, 2013

PRACTICAL TRACTABILITY OF CSPS BY HIGHER LEVEL CONSISTENCY AND TREE DECOMPOSITION

Shant Karakashian, Ph. D.

University of Nebraska, 2013

Adviser: Berthe Y. Choueiry

Constraint Satisfaction is a flexible paradigm for modeling many decision problems in Engineering, Computer Science, and Management. Constraint Satisfaction Problems (CSPs) are in general $\mathcal{NP}$-complete and are usually solved with search. Research has identified various islands of tractability, which enable solving certain CSPs with backtrack-free search. For example, one sufficient condition for tractability relates the consistency level of a CSP to treewidth of the CSP's constraint network. However, enforcing higher levels of consistency on a CSP may require the addition of constraints, thus altering the topology of the constraint network and increasing its treewidth. This thesis addresses the following question: How close can we approach in practice the tractability guaranteed by the relationship between the level of consistency in a CSP and the treewidth of its constraint network?

To achieve "practical tractability," this thesis proposes: (1) New local consistency properties and algorithms for enforcing them without adding constraints or altering the network's topology; (2) Methods to enforce these consistency properties on the clusters of a tree decomposition of the CSP; and (3) Schemes to bolster the propagation between the clusters of the tree decomposition.

Our empirical evaluation shows that our techniques allow us to achieve practical tractability for a wide range of problems, and that they are both applicable (i.e., require acceptable time and space) and useful (i.e., outperform other consistency properties).

We theoretically characterize the proposed consistency properties and empirically evaluate our techniques on benchmark problems. Our techniques for higher level consistency exhibit their best performances on difficult benchmark problems. They solve a larger number of difficult problem instances than algorithms enforcing weaker consistency properties, and moreover they solve them in an almost backtrack-free manner.

DEDICATION

To my family and my wife

ACKNOWLEDGMENTS

I am indebted to all those who helped me throughout the years of my graduate studies. I am grateful to my adviser, Professor Berthe Y. Choueiry, for her guidance, help, support and patience. I would like to thank the members of my Supervising Committee for their support and input, in particular the two readers of my dissertation Professor Rina Dechter and Professor Matthew Dwyer for the constructive feedback and corrections. The research described in this dissertation was inspired by some of the research contributions of Professor Dechter, and the discussion related in Section 6.5.4 is a direct result of her input.

I would like to acknowledge research collaborations and the scientific input of the following individuals: Professor Rina Dechter and Dr. Radu Marinescu for discussions about tree decomposition, Mr. Robert J. Woodward for collaboration on Chapters 3 and 5, Professor Stephen G. Hartke for collaboration on Appendix A, Dr. Christian Bessiere for collaboration on Chapters 3, 5, and Appendix B, Mr. Christopher G. Reeson for collaboration on Chapter 3, Professor Kostas Stergiou for sharing his data and random problem generator, and Professor Stephen D. Scott and Mr. Daniel Geschwender for collaboration on Chapter 4.

My gratitude goes to Mrs. Gassia Akilian, Mr. Daniel Geschwender, Ms. Phyllis O'Neil, Mr. Tony Schneider, and Mr. Robert J. Woodward for carefully proofreading this dissertation. Special thanks to Professor David Swanson, Mr. Tom Harvil and the entire team of the Holland Computing Center team for supporting me with the use of computing resources necessary for the experiments.

Special thanks to Professor Eugene C. Freuder, Dr. Stephen Prestwich, and Dr. Richard Wallace for hosting me at 4C during Summer 2010, which was a wonderful research experience and invaluable training.

I would like to thank all the members of my family and my wife for their unconditional support, infinite sacrifice, and unwavering encouragement throughout this journey. My sincere gratitude goes to Dr. Jihad Boulos and Dr. Walid Keirouz for their mentorship and friendship, and for being there for me during the most critical stages of my academic journey.

GRANT INFORMATION

This research supported by a NSF CAREER Award #0133568 and NSF Grant No. RI-111795. Experiments were conducted on the equipment of the Holland Computing Center at UNL.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Constraint Satisfaction is a general and flexible paradigm for modeling many decision problems in Engineering, Computer Science, and Management. The formulation of a Constraint Satisfaction Problem (CSP) can easily be extended to include optimization criteria, thus allowing the modeling of optimization tasks. In this thesis, we will focus on decision problems.

Generally speaking, CSPs are $\mathcal{NP}$-complete, thus, likely intractable. Research on this topic started as early as the 1960's [Sutherland, 1963; Fikes, 1970; Montanari, 1974; Waltz, 1975], and the field has matured into an independent research area in Artificial Intelligence with textbooks [Tsang, 1993; Dechter, 2003; Lecoutre, 2009], a handbook [Rossi *et al.*, 2006], a professional association[1] and an international conference series.[2]

Two fundamental mechanisms are used to solve CSP instances: constraint propagation and backtrack search, respectively called *inference* and *conditioning* [Dechter, 2003]. Typically, in a pre-processing step, we enforce a given consistency property on a problem instance. We do this by constraint propagation, hoping to uncover inconsistency before starting search. This overhead can be worthwhile because constraint

[1] Association for Constraint Programming `http://4c.ucc.ie/a4cp`.
[2] International Conference on Principles and Practice of Constraint Programming.

propagation algorithms are usually efficient (i.e., run in polynomial time) whereas search is usually not. Then, we interleave constraint propagation with (systematic, exhaustive) backtrack search in a complete and sound algorithm to solve a CSP instance. Local search is another approach for solving CSPs, but it yields techniques that are generally neither complete nor sound. Such techniques are beyond the scope of this thesis.

The contributions reported in this thesis are mainly concerned with inference-based techniques. More specifically, we define new consistency properties, design algorithms for enforcing them, and develop structure-based mechanisms for propagating them. Below, we motivate our investigations and summarize our contributions.

1.1 Motivation & Claim

Although a CSP is in general $\mathcal{NP}$-complete, research in Constraint Processing (CP) has identified various *islands of tractability* as classes of CSPs that can be solved in polynomial time in the size of the input [Gottlob and Szeider, 2008]. We single out the tractability condition specified by a relationship between

- A structural parameter of the constraint network of a CSP such as the *treewidth* or the *hypertree width*, and
- The *level of a consistency* that the corresponding CSP possesses.

Larger network widths typically require higher levels of consistency to guarantee backtrack-free search [Freuder, 1982; Dechter and Pearl, 1987]. This approach is hindered in practice by two main obstacles:

1. Finding the treewidth or hypertree width of a constraint network is an $\mathcal{NP}$-hard task [Arnborg *et al.*, 1987; Gottlob *et al.*, 2002].

2. Enforcing higher levels of consistency may require adding constraints to the CSP, thus modifying its structure and width parameters.

As a result, few researchers have exploited this important tractability result for solving CSPs or for counting a CSP's number of solutions. Exceptions include the following: *a*) Theoretical studies [Freuder, 1982; 1985]; *b*) Exact algorithmic techniques [Dechter and Pearl, 1987; 1989; Gyssens *et al.*, 1994; Jeavons *et al.*, 1994; Dechter, 1996; Gottlob *et al.*, 2000; Jégou and Terrioux, 2003]; and *c*) Techniques that provide upper bounds on the number of solutions [Dechter, 1997; Favier *et al.*, 2009; Rollon and Dechter, 2010]. Given the difficulty of finding the treewidth of the constraint network of a CSP, most algorithmic techniques approximate the treewidth of the constraint network using a *tree decomposition* embedding of that network.

In this thesis, we address the following question: *How close can we approach in practice the tractability guaranteed by the relationship between the level of consistency in a CSP and the width of its constraint network?* We propose to achieve "practical tractability," that is, recognizing, as we are solving them, problems whose complexity is inherently bounded, as follows:

1. Propose new local consistency properties whose level is controlled by a parameter and design algorithms for enforcing them that do not modify the structure of the constraint network.

2. Control the cost of our algorithms by localizing them to the subproblems delimited by a tree decomposition of the CSP.

3. Bolster constraint propagation and enhance communications between subproblems by adding redundant constraints that do not affect the treewidth of the tree decomposition used.

1.2 Approach

Our approach is centered on enforcing higher level consistencies in the context of a tree decomposition in order to reap the benefits of such consistencies for solving CSPs.

1.2.1 Higher level consistencies

Generally, consistency algorithms operate on combinations of variables or constraints. In practice, the most popular algorithms consider combinations of at most three variables or two relations. Except for the simplest cases, all may in general require the addition of new constraints [Freuder, 1978; Dechter and van Beek, 1997]. Moreover, different problems require different levels of consistency. For this reason, it becomes important to explore new properties:

1. Whose level of consistency can be controlled (i.e., parameterized consistency), but

2. That do not modify the structure of the constraint network, and thus, do not increase its width.

1.2.2 Tree decomposition

The main techniques that exploit the structure of the constraint network for solving the CSP use a tree embedding of the constraint network. Because finding the optimal decomposition is $\mathcal{NP}$-Hard, heuristics are used to find a 'good' decomposition. Techniques for finding tree decompositions include the following: join tree or tree clustering [Dechter and Pearl, 1989], hinge decomposition [Cohen *et al.*, 2008], and hypertree decomposition [Gottlob and Scarcello, 2001]. We will use the tree clustering. It is the

most commonly used technique on graphical models, can be computed efficiently, and produces trees of relatively good quality.

The vertices of a tree decomposition are subproblems of the CSP. In order to ensure the consistency of the solutions of two adjacent subproblems, we must synthesize and store a global constraint over their 'overlap' (i.e., their separator). The space for storing such global constraints is a major obstacle.

We exploit the tree decomposition in our framework as follows:

1. Localize the application of our consistency algorithms to the subproblems induced by the tree decomposition.

2. Guide the constraint-propagation process along the branches of the tree to favor the most constrained paths for early conflict detection.

3. Enhance constraint propagation by adding properly chosen redundant constraints between adjacent subproblems.

1.3 Contributions

In this section, we summarize our main contributions. We divide them into core contributions, which support the main claim of this dissertation, and secondary contributions, which are 'peripheral' to the main claim. Our core contributions are the following:

1. *Relational Consistency.* We propose and theoretically characterize a new relational consistency property, R($*$,m)C, parameterized by the number of constraints over which it applies.

2. *Two algorithms for enforcing R(∗,m)C.* We propose and empirically evaluate two algorithms for enforcing R(∗,m)C. A key feature of our algorithms is that they do not modify the topology of the constraint network, and thus do not affect its treewidth. We show that the two algorithms perform best under complementary conditions, and use machine learning techniques to build a decision tree to determine when best to use each algorithm.

3. *Localized consistency.* We localize our mechanisms for relational consistency to the subproblems delimited by a tree decomposition of the constraint network in order to reduce the number of combinations of m relations to which R(∗,m)C is applied.

4. *Structure-based constraint propagation.* We organize the propagation of consistency algorithms along the tree decomposition, thus speeding up the propagation process. We propose three such propagation schemas, and empirically show that the benefits drawn from exploiting the structure largely dominate the other improvements we envisaged.

5. *Bolstering propagation at separators.* We propose to add redundant constraints at the separators in order to improve propagation between clusters. A unique global constraint over a separator ensures a perfect communication between two clusters. In practice, the size of the corresponding relation is prohibitive. We propose three approximations of the ideal case. We identify and characterize the consistency properties that result from this mechanism.

Our secondary contributions are the following:

1. *Witness-based solutions counting.* The technique known as "backtrack with tree decomposition" (BTD) is a backtrack search that exploits a tree decomposition of

a CSP [Jégou and Terrioux, 2003]. Favier *et al.* use the BTD to count the number of solutions of a CSP [2009]. We propose witness-based BTD (WitnessBTD) as an improvement to the performance of the BTD for solutions counting. For a given partial solution, WitnessBTD guarantees that the partial solution is not a nogood by finding a 'witness' before attempting to count the number of other partial solutions based on this partial solution. We compare the performance of the BTD, WitnessBTD and our various strategies based on bolstering.

2. *Computing all k-connected subgraphs.* We propose a new algorithm for computing all connected subgraphs of size k. This algorithm is important for computing all combinations of m relations of a CSP, represented by the vertices of its dual graph. It dramatically outperforms the naive algorithm for enumerating these combinations for large sparse graph and small values of k.

3. *Complexity of solution covering.* We prove that finding the minimum number of solutions that cover all the tuples of a minimal CSP is $\mathcal{NP}$-hard.

Whenever applicable, we theoretically characterize and empirically evaluate and compare the proposed new concepts and mechanisms.

1.4 Document Structure

The rest of this dissertation is organized as follows:

Chapter 2 reviews background information.

Chapter 3 defines the new relation consistency property R($*$,m)C, introduces a first algorithm, PerTuple (Algorithm 1), for enforcing it, and empirically evaluates

the performance of this algorithm. Preliminary results from this chapter have been published [Karakashian *et al.*, 2010b; 2010a].

Chapter 4 introduces the algorithm AllSol (Algorithm 2) as an alternative to PerTuple (Algorithm 1), and discusses a decision-tree procedure built using Machine Learning techniques for choosing the appropriate algorithm. Preliminary results from this chapter appeared in technical report [Karakashian *et al.*, 2012].

Chapter 5 discusses localized consistency and structure-based constraint propagation. Results from this chapter have been published [Karakashian *et al.*, 2013].

Chapter 6 discusses three strategies for bolstering constraint propagation at the separators in a tree decomposition by addition of redundant constraints. Results from this chapter have been published [Karakashian *et al.*, 2013].

Chapter 7 introduces our improvement to solution counting by the BTD via the production of a witness solution.

Chapter 8 concludes this dissertation and suggests directions for future research.

In order to maintain the coherence of this document, incidental results and complementary information that are not central to the core contributions are organized in five appendices:

Appendix A introduces a new algorithm for computing all combinations of k-sized connected subgraphs of a given graph.

Appendix B introduces the solution-cover problem and shows that it is $\mathcal{NP}$-complete.

Appendix C provides all the proofs of all theorems in the dissertation, removed to appendices in order to increase readability.

Appendix D provides an iterative version of the algorithm WitnessBTD (Algorithm 8), introduced recursively in Chapter 7.

Appendix E describes the characteristics of the benchmark data used in the experiments in Chapters 5, 6, and 7.

Summary

This chapter introduced our motivation and claim, listed our contributions, and established the structure of this document.

Chapter 2

Background Information

In this chapter, we introduce some relevant background information. In particular, we discuss CSPs, their graphical representations, consistency properties, and algorithms for enforcing such properties. Then, we define a tree decomposition, and give an overview of exact and approximate methods for solving CSPs that exploit a tree structure of the problem.

This chapter summarizes results from the literature that are relevant to the contributions of this thesis. However, subsequent chapters may discuss related work locally in order to draw better contrast and comparisons to the specific contributions discussed in each chapter.

2.1 Constraint Satisfaction Problems (CSPs)

Below, we give a formal definition of a CSP and its components, describe its graphical representations and the general methods for solving it.

2.1.1 Problem definition

A Constraint Satisfaction Problem (CSP) is defined by $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where

1. $\mathcal{X} = \{A_1, A_2, \ldots, A_n\}$ is a set of n variables.
2. $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$ is the set of respective domains, where D_i, the domain of variable A_i, is a set of values that can be assigned to variable A_i. In this thesis, we consider only finite domains.
3. $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$ is a set of m constraints, restricting the allowed combination of values to variables. (Section 2.1.2 discusses constraint definitions in more detail.)

A *solution* to a CSP is an assignment of one value to each variable such that all constraints are simultaneously satisfied. Solving a CSP corresponds to finding a solution, which is a satisfiability problem, or finding all solutions, which is a counting problem. Generally speaking, the satisfiability problem is $\mathcal{N}P$-complete and the counting problem is in #P. The optimization problem of finding the least constrained solution (e.g., MAX-CSP [Freuder and Wallace, 1992]) is $\mathcal{N}P$-hard.

Example 1 *Consider $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ where: a) $\mathcal{X} = \{A, B, C, D, E, F, G\}$; b) $\mathcal{D} = \{D_A, D_B, D_C, D_D, D_E, D_F, D_G\}$, with $D_{i \in \mathcal{X}} = \{0, 1\}$ (such a CSP is thus called a Boolean CSP); and c) The relations of the constraints in $\mathcal{C}$ are given in Figure 2.1.*

The tuples highlighted in Figure 2.1 represent a solution to this CSP and correspond to the following assignments of values to the variables:

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

R_1		
A	E	F
0	0	1
0	1	0
0	1	1
1	0	1
1	1	0

R_2	
B	E
0	1
1	0

R_3		
A	B	C
0	0	1
0	1	0
0	1	1
1	0	1
1	1	0

R_4		
A	D	G
0	0	1
0	1	0
0	1	1
1	0	1
1	1	0

R_5	
B	D
0	1
1	0

Figure 2.1: A simple CSP example.

2.1.2 Defining constraints

A constraint C_i is defined by its scope, denoted $scope(C_i)$, and a relation $R_i = rel(C_i)$.

The scope of C_i, $scope(C_i) \subseteq X$, is the set of variables to which the constraint applies. The *arity* of a constraint is the cardinality of its scope, $|scope(C_i)|$. For a unary constraint, $|scope(C_i)| = 1$, and for a binary constraint $|scope(C_i)| = 2$. When $|scope(C_i)| > 2$, the constraint is said to be non-binary.

The relation R_i is a subset of the Cartesian product of the domains of the variables in $scope(C_i)$: $rel(C_i) \subseteq \prod_{A_x \in scope(C_i)} D_x$. The relation can be defined in *intension* by a predicate function that determines whether or not a tuple is allowed, i.e., whether the tuple is *consistent* with the constraint. Alternatively, the relation can be defined in *extension* by explicitly listing the elements in the subset. Each element of this subset is a *tuple*. The tuples can be the allowed combinations of values, called *supports*, or the forbidden combinations, called *conflicts* or *nogoods*. When the relation is defined in extension, the constraint is called a *table constraint*. While the discussion in this thesis is limited to table constraints, some of the techniques can be extended to constraints defined in intension. Such extensions are beyond the scope of this thesis.

A *universal constraint* is a binary constraint that allows every combination in the cross product of the domains of the two variables.

2.1.3 Graphical representations

There are several graphical representations for a CSP:

- *Hypergraph*: The vertices represent the variables of the CSP and the hyperedges represent the scopes of the constraints. Figure 2.2 shows the hypergraph of the CSP in Example 1.

- *Primal graph*: The vertices represent the variables and (binary) edges link every two variables that appear in the scope of some constraint. Figure 2.3 shows the primal graph of the CSP in Example 1.

- *Dual graph*: The vertices represent constraints of the CSP, and are labeled by the constraints' respective scopes. An edge connects two vertices corresponding to constraints whose scopes overlap. Figure 2.4 shows the dual graph of the CSP in Example 1. Thus, the dual CSP is a binary CSP where: (1) variables are the constraints of the original CSP; (2) the variables' domains are the tuples of the corresponding relations; and (3) the binary constraints enforce *equalities* over the shared variables to prevent a given variable from having different values.

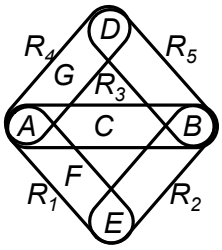

Figure 2.2: Hypergraph.

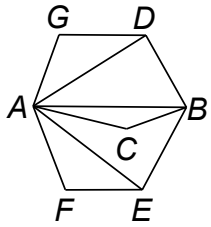

Figure 2.3: Primal graph.

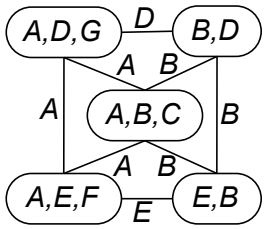

Figure 2.4: Dual graph.

Figure 2.5 shows two alternative hypergraphs of a slightly more complex example than the one in Figure 2.2. Here, the hyperedges are represented by two different but equivalent ways. Figure 2.6 shows the corresponding primal and dual graphs.

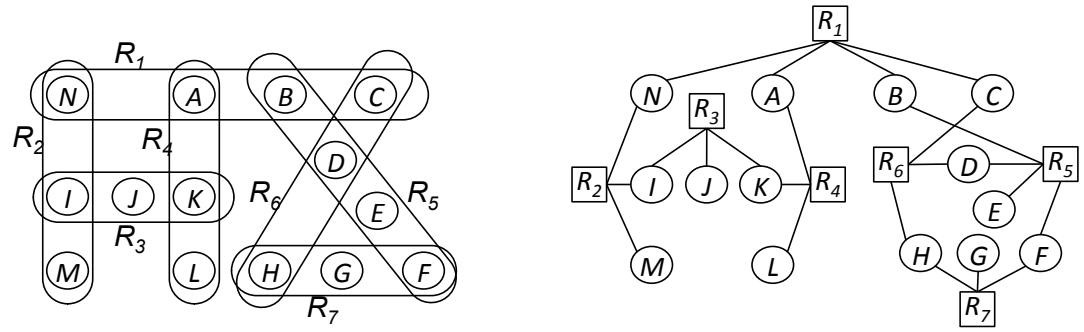

Figure 2.5: Two representations of the hypergraph of a non-binary CSP.

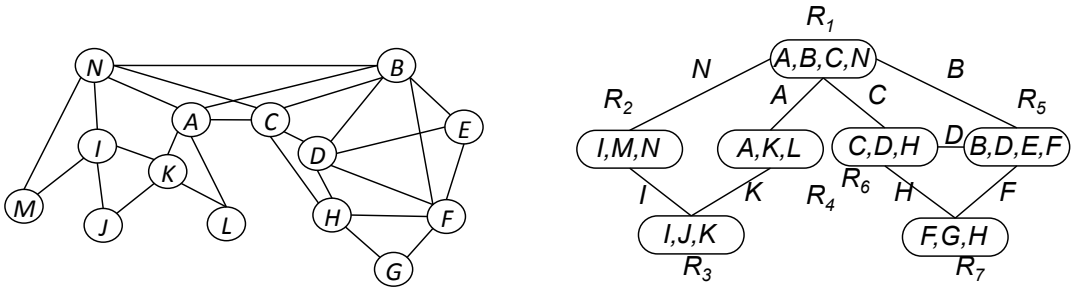

Figure 2.6: The primal *(left)* and dual *(right)* graphs of the CSP in Figure 2.5.

A particularly interesting opportunity arises when we consider the dual graph of a CSP. In the dual graph, edges enforce the equality of the shared variables of two adjacent vertices. It was observed that an edge between two vertices is *redundant* if there exists an alternate path between the two vertices such that the shared variables appear in every vertex in the path [Dechter and Dechter, 1987; Dechter and Pearl, 1989; Janssen *et al.*, 1989; Dechter, 2003]. Such redundant edges can be removed without modifying the set of solutions of the CSP. Figure 2.7 shows the dual graph of a CSP, where the edges drawn in dashed lines are redundant. Indeed, the value for variable A is enforced between R_1 and R_3 through R_4, and for variable C between R_2 and R_3 through R_5. Figure 2.8 shows two redundant edges in the dual graph of Figure 2.4.

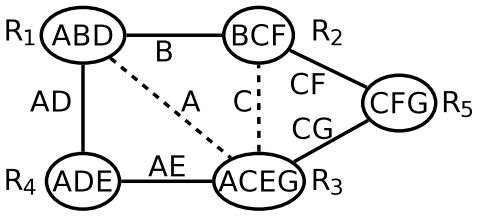

Figure 2.7: Showing the redundant edges of the dual graph of a CSP.

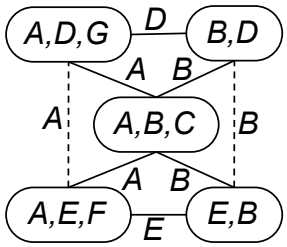

Figure 2.8: Two redundant edges of the graph in Figure 2.4.

Janssen *et al.* [1989] introduced an efficient algorithm for computing the *minimal dual graph* by removing redundant edges. Many minimal graphs may exist, but they are all guaranteed to have the same number of remaining edges.

2.1.4 Solving CSPs

A CSP can be solved by search (i.e., *conditioning*) or by synthesizing and propagating constraints (i.e., *inference*) [Dechter, 2003]. Search can be based on backtracking or on iterative repair (also called local search). In this thesis, we focus on backtrack search and do not discuss local search.

Backtrack search is a constructive, systematic, exhaustive exploration of the combinations of assignments of values to variables. It proceeds in a depth-first manner by instantiating the variables, one at a time, iteratively building a consistent partial solution. When the currently built path cannot be extended, instantiations are undone by backtracking.

Variable ordering is known to significantly affect the performance of the search process. The common wisdom is to first instantiate the 'most constrained variable' in an effort to reduce the branching factor of the tree. This general principal is implemented by a variety of *variable-ordering heuristics*.

In order to reduce the size of the search space, we typically interleave backtrack

search with constraint propagation in what is called a *look-ahead* schema. Typically, every time that a variable is instantiated during search, the effects of this decision are propagated over the unassigned variables by removing from their domains values that do not agree with the current instantiation. Look-ahead can be partial as in *forward checking*, which updates the domains of only the variables adjacent to the variable being instantiated. A *full look-head* schema enforces, after each variable instantiation, a given consistency level over all uninstantiated variables.

Typically, inference methods operate on a CSP by enforcing consistency and propagating constraints, with or without generating new constraints (constraint synthesis).

2.2 Consistency Properties

A *consistency property* guarantees that the values of all combinations of a given number of CSP variables (alternatively, the tuples of all combinations of a given size of CSP relations) are consistent with the constraints that apply to them. This condition is necessary but not sufficient for the values (or the tuples) to appear in a solution to the CSP.

2.2.1 Variable-based consistency properties

Variable-based consistency properties are defined on combinations of variables, and guarantee that the values of the variables in these combinations are consistent with the set of constraints defined over the variables.

The Arc Consistency (AC) property is a popular consistency property for binary CSPs [Mackworth, 1977]. AC is considered to be a low level consistency: it considers every combination of only two variables and guarantees that every value in the domain

of one variable is consistent with at least one value in the domain of the other variable. For non-binary CSPs, Generalized Arc Consistency (GAC) is popular. A CSP is GAC if and only if, for every constraint, any value in the domain of any variable in the scope of the constraint can be extended to a tuple satisfying the constraint.

More generally, k-consistency requires that every consistent assignment to every $k-1$ variables can be extended to every k^{th} variable (i.e., the assignment to any k variables satisfies all the constraints that apply to them) [Freuder, 1978]. Obviously, if a given combination of values for $k-1$ variables cannot be extended to some k^{th} variable, the combination cannot possibly appear in any solution to the CSP. Enforcing k-consistency may require adding constraints of arity $k-1$ in order to disallow all $(k-1)$-tuples that cannot be extended to some k^{th} variable. Such an operation may change the topology of the constraint network, which is one reason such consistency properties are avoided for $k > 2$.

The more general (i, j)-consistency property requires that every consistent assignment of i variables can be extended into a consistent assignment to every j other variables [Freuder, 1985]. Thus, AC is (1,1)-consistency and k consistency is $(k-1, 1)$-consistency.

Such parameterized consistency properties allow one to control the strength of the property via the parameters i, j, k. While theoretically interesting, higher level consistencies pose great practical challenges. The challenges are due to the high processing cost and the memory cost for storing the added constraints.

AC and GAC are particularly popular in practice because of their relatively low cost, high benefits, and the fact that they do not change the topology of the network. AC for binary constraints and GAC for non-binary constraints are the most widely used consistency properties.

2.2.2 Relation-based consistency properties

In this category, we target consistency properties defined on all combinations of a fixed size of relations. The most general work in this area is the work of Dechter and van Beek [1997] on relational m-consistency and relational (i,m)-consistency.

- Relational m-consistency considers every combination of m constraints and requires that, if s is the union of the scopes of the m constraints, every consistent assignment of every $|s|-1$ variables of s can be extended in a consistent manner to the s^{th} variable. In practice, enforcing relational m-consistency may require adding constraints of arity $|s|-1$.

- Relational (i,m)-consistency is designed to bind the arity of the added constraints by the parameter i. Hence, it requires that every consistent partial solution of length i be extended to a consistent partial solution of length s. Again, s is the set of variables in the scopes of every combination of m constraints.

The only other consistency property defined over combinations of constraints of which we are aware is m-wise consistency. This property was proposed in the area of relational databases [Gyssens, 1986], where it was defined but never used in practice. To the best of our knowledge no algorithm was proposed for enforcing it. *It is the basis of the main contribution of this thesis.*

For $m = 2$, this property is known as *pairwise consistency* [Beeri *et al.*, 1983]. It requires that every tuple in a relation can be extended to a tuple in every other relation such that both relations are satisfied. Pairwise consistency was studied by Janssen *et al.* [1989].

2.2.3 Global consistency properties

When the property is restricted to combinations of variables or combinations of constraints of a fixed size, it is said to be *local*, such as the ones discussed in Sections 2.2.1 and 2.2.2 above. When the property is defined over the entire CSP, it is said to be *global*. Examples of global consistency properties are *minimality* and *decomposability* [Montanari, 1974].

Constraint minimality requires that every tuple in a constraint appears in a solution. Decomposability guarantees that every consistent partial solution of any length can be extended to a complete solution. Decomposability is a highly desirable property: it guarantees that the CSP can be solved in a backtrack-free manner.

While local consistency properties are typically tractable, global constraints are in general likely to be intractable.

2.2.4 Comparing consistency properties

In order to compare two consistency properties, we use the terminology introduced by Debruyne and Bessiere [1997] and Bessiere *et al.* [2008]:

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

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.

2.3 Algorithms for Enforcing Consistency

More than one algorithm may exist for enforcing a given consistency property. A *consistency algorithm* or *constraint propagation algorithm* can operate by removing values from the domains of the variables (domain filtering), adding constraints (constraint synthesis), or removing tuples from the constraint definitions (relation filtering). The added constraints, if any, are said to be *implicit* or *redundant* because they are entailed by the original set of constraints but are needed to guarantee a given level of consistency. Enforcing consistency reduces the search space, makes the solutions more 'apparent,' but never changes the set of solutions to the CSP.

2.3.1 Domain filtering

Domain filtering algorithms remove a value from the domain of a variable because it is inconsistent with a constraint or a combination of constraints. Removal of values in one part of the problem may cause other values to become inconsistent. The process is repeated until quiescence, which is guaranteed in finite CSPs.

The Arc Consistency (AC) property is thoroughly studied and many algorithms have been proposed for enforcing it on a CSP. For example, AC3 [Mackworth, 1977] (noteworthy for its simplicity) and AC2001/3.1 [Bessiere *et al.*, 2005] (popular for its improved performance). Given the importance of AC, designing new algorithms for enforcing it remains a popular research topic [Lecoutre and Hemery, 2007; Lecoutre *et al.*, 2008].

Dechter and Pearl provide algorithms for *directionally* enforcing AC and the parameterized properties mentioned in Section 2.2.1 along a given fixed order of the CSP variables [1987]. Enforcing directional consistency is cheaper than otherwise (and in our experience, worthwhile) even though it results in a weaker form of the enforced property.

2.3.2 Constraint synthesis

In order to enforce consistency properties of a level higher than AC or GAC, one may have to generate new constraints. *Constraint synthesis* is a general algorithm that iteratively generates constraints of arity $k+1$ using all constraints of arity k starting from $k = 2$ (arc consistency) to $k = n$ (which guarantees solvability) [Freuder, 1978]. We are not aware of any practical implementation that exploits this approach, which remains, however, conceptually appealing.

2.3.3 Relation filtering

Similarly to filtering out inconsistent values from the domains of the CSP variables, one can imagine removing inconsistent tuples from the domains of the constraints. Such mechanisms have received relatively little attention. The only mechanism in this category of which we are aware is an algorithm for enforcing pairwise consistency by running an arc consistency algorithm on the dual encoding of a CSP [Janssen *et al.*, 1989].

A central contribution of this thesis is the design and evaluation of two algorithms of this category for enforcing a parameterized relational consistency property that generalizes pairwise consistency. These algorithms are introduced and discussed in Chapters 3 and 4.

2.3.4 Techniques for improving performance in practice

Various data structures are used to improve the performance of consistency algorithms in practice. The most popular example is the *support* structure used in AC2001/3.1 [Bessiere *et al.*, 2005], which remembers the support of every value of a variable in the domain of the other variable. While constraints are being propagated, one does not seek new supports unless the previously found ones were deleted by AC. Lecoutre *et al.* [2003] generalized this approach to multiple supports by exploiting the symmetry of the support relation (i.e., when a value supports another one, then it is also supported by this other value), thus reducing the effort of seeking supports.

More recently, the multiple supports approach has been closely coupled with backtrack search to save further on the effort of finding supports by exploiting residual supports from the previous step during search [Likitvivatanavong *et al.*, 2007; Lecoutre and Hemery, 2007; Lecoutre *et al.*, 2008].

2.4 Tree-Structured Constraint Networks

When the constraint network of a CSP has a tree structure, a well-established result shows that:

- The existence of a solution (i.e., solvability or consistency) is efficiently guaranteed by directional arc consistency (DAC), which is enforced in the following manner: starting from the leaves of the tree up to an arbitrary chosen root node, the domain of each variable is revised by arc consistency with its children [Freuder, 1982; Dechter and Pearl, 1987].

- After enforcing DAC, a solution can be built in a backtrack-free manner starting from the root and proceeding down towards the leaves of the tree [Freuder, 1982;

Dechter and Pearl, 1987]. Further,

- The number of solutions of the CSP can be efficiently counted by multiplying the number of 'extensions' in the children of a variable that are rooted at a value of the variable and by summing up the number of partial solutions rooted at the values of the variable [Dechter and Pearl, 1987].

In the case of a non-binary CSP, similar conditions hold when the dual graph is a tree structure, which is called a *join tree*. Such a structure may be hidden by the existence of redundant edges (see Section 2.1.3). Dechter recalls two procedure for determining whether a dual graph has a join tree (see Section 9.1 [Dechter, 2003]).

When such an advantageous tree structure is not readily available, one can approximate it by a tree decomposition, which can bind the cost of solving the CSP by a structural parameter of the generated tree structure such as its treewidth. Such techniques have yielded new tractability results for CSPs based on single-parameter complexity.

2.5 Tree Decomposition

A *tree decomposition* of a CSP is a tree embedding of the constraint network. Below, we formally define it and discuss CSP solving methods that utilize it.

2.5.1 Definition

A tree decomposition of a CSP is a tree embedding of the constraint network of the CSP. It is defined by a triple $\langle \mathcal{T}, \chi, \psi \rangle$, where $\mathcal{T}$ is a tree, and χ and ψ are two functions that determine which CSP variables and constraints appear in which nodes of

the tree (see Chapter 9 [Dechter, 2003]). The tree nodes are thus *clusters* of variables and constraints. A tree decomposition must satisfy two conditions:

1. Each constraint appears in at least one cluster, and the variables in its scope must appear in this cluster.

2. For every variable, the clusters where the variable appears induce a connected subtree.

Figure 2.9 shows a tree decomposition of the CSP in Figure 2.2 (left) and Figure 2.5 (right). A *separator* of two adjacent clusters is the set of variables in both clusters. For example, the separator of clusters C_1 and C_2 in the left figure is $\{A, E\}$.

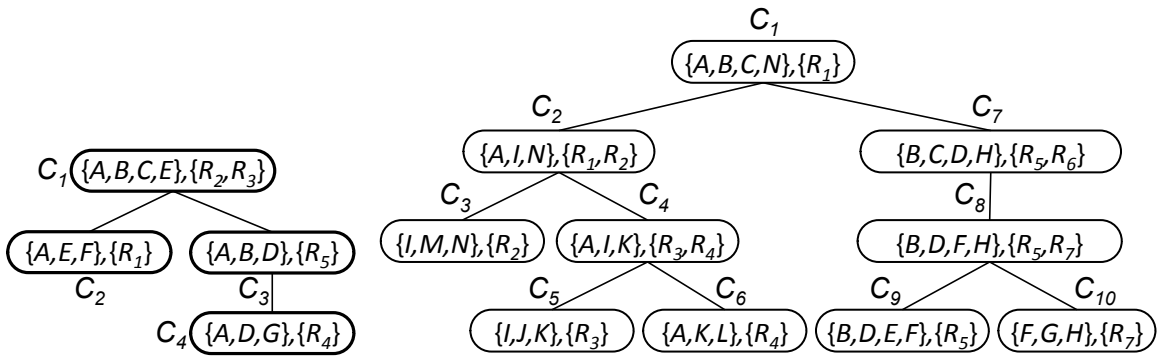

Figure 2.9: A tree decomposition of the CSP in Figure 2.2 *(left)* and Figure 2.5 *(right)*.

A given tree decomposition is characterized by its *treewidth*, which is the maximum number of variables in a cluster. The treewidths of both tree decompositions in Figure 2.9 are four. The treewidth of a CSP is the minimum treewidth of all its tree decompositions. Determining the treewidth of a CSP is, in general, $\mathcal{N}P$-hard. CSPs with a fixed treewidth can be solved in polynomial time and are thus tractable [Arnborg, 1985; Robertson and Seymour, 1986].

A tree decomposition is also a *hypertree decomposition* if it satisfies the following additional condition:

3. Each variable in the cluster must appear in the scope of a constraint that appears in the cluster.

Similarly, a given hypertree decomposition is characterized by its *hypertree width*, which is the maximum number of constraints in a cluster. The hypertree widths of both tree decompositions in Figure 2.9 are two. The hypertree width of a CSP is the minimum hypertree width of all its hypertree decompositions. Determining the hypertree width of a CSP is also $\mathcal{NP}$-hard in general. CSPs with fixed hypertree widths can be solved in polynomial time and are thus tractable.

2.5.2 Structural parameters and tractability

The higher the level of consistency in a CSP, the more likely that a current path in the search tree can be expanded towards a consistent assignment of the variables, and thus, towards a solution. This observation prompts an important question: what is the level of consistency that one must enforce on the CSP in order to guarantee a backtrack-free search?

Freuder [1985] provided a sufficient condition for a backtrack-free search by linking the consistency level to a *structural parameter* of the constraint graph of a binary CSP: its *width*. However, in practice, enforcing a given level of consistency may require adding constraints, thus modifying the width of the constraint graph, which, in turn, increases the level of consistency required to guarantee a backtrack-free search.

2.5.3 Main tree-decomposition techniques

Finding the optimal tree-decomposition is $\mathcal{NP}$-Hard, thus most approaches are heuristics that attempt to find a 'good' decomposition. The main techniques for generating tree decompositions and hypertree decompositions include the following: join tree

[Dechter and Pearl, 1989], hinge decomposition [Jeavons *et al.*, 1994], hypertree decomposition [Gottlob *et al.*, 1999; 2000], and Cut-and-Traverse (CaT) [Zheng and Choueiry, 2005].

Tree clustering proposed by Dechter and Pearl [1989] is a tree decomposition method that clusters the maximal cliques in the triangulated primal graph of a CSP. The quality of the decomposition depends on the heuristic used for triangulating the graph.

Hypertree decomposition is more general than tree clustering. Algorithms proposed by Gottlob and Samer [2009] find the optimal hypertree decomposition, though the heuristic methods proposed by Dermaku *et al.* [2008] have better run times and achieve near optimal results.

The hinge decomposition presented by Cohen *et al.* [2008] and hinge+ by Zheng and Choueiry [2005] also implement tree decompositions. The hinge decomposition is combined with tree clustering by Gyssens *et al.* [1994] to yield a more general tree decomposition than each taken separately.

2.6 Solving CSPs with Tree Decomposition

As stated above, one important early result by Freuder [1985] provides a sufficient condition for a backtrack-free search by linking the level of consistency satisfied by a CSP to the *width* of its constraint graph. This result, along with theoretical developments in relational databases, probabilistic reasoning, and dynamic programming, is at the foundation of structure-based techniques for solving CSPs (see Section 9.5 of [Dechter, 2003]). These techniques are each represented by one of the following three categories:

1. Cluster-centered methods

2. Variable-centered methods

3. Backtrack-search based methods

While the first two types of methods are inference based (i.e., by constraint synthesis and propagation), the third one is, for the most part, based on conditioning (i.e., search). However, all three types of methods are tightly linked in that they exploit some tree decomposition of the CSP. The time complexity of these techniques is bounded by the size of the treewidth of the tree decomposition (called induced width in the case of variable-based methods) and the space complexity by the size of the largest separator in the tree. For problems with a fixed treewidth, these techniques are said to be efficient, with polynomial time worst-case complexity [Gottlob and Szeider, 2008].

2.6.1 Cluster-centered methods

These methods apply directly to a tree decomposition of the CSP. They process the clusters of the tree decomposition (typically by inference), then channel information between clusters along the paths of the tree (typically by propagation or message passing), as demonstrated in Figure 2.10. Examples of such techniques include the

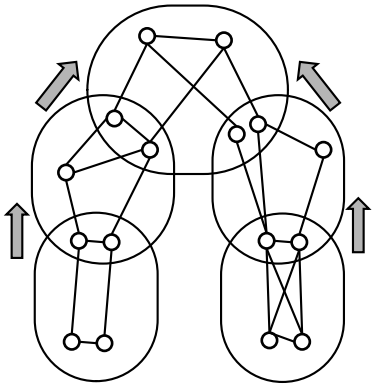

Figure 2.10: Cluster-centered methods.

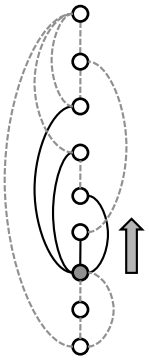

Figure 2.11: Variable-centered methods.

following: tree clustering (a.k.a. join-tree clustering) [Dechter and Pearl, 1989], cluster-tree elimination (a.k.a. bucket-tree elimination) [Kask *et al.*, 2005], and mini-cluster tree elimination [Dechter *et al.*, 2001].

2.6.2 Variable-centered methods

These methods exploit some fixed, linear ordering of the variables. Proceeding bottom-up along the chosen ordering, they process the constraints, typically by inference, that link the considered variable to the ones at higher levels in the ordering and then channel the effects of this processing to the other variables in the scope of those constraints, typically by propagation or by constraint synthesis (i.e, generation). Figure 2.11 illustrates this process. Examples of such methods include the following: adaptive consistency [Dechter and Pearl, 1987], bucket elimination [Dechter, 1996; 1999], and mini-bucket elimination [Dechter, 1997; Dechter and Rish, 2003].

A typical ordering of the variables is a *perfect elimination ordering*[1] of the vertices of some triangulation[2] of the primal graph of a CSP. Figure 2.12 shows the primal graph and a triangulated primal graph of the CSP in Figure 2.5.

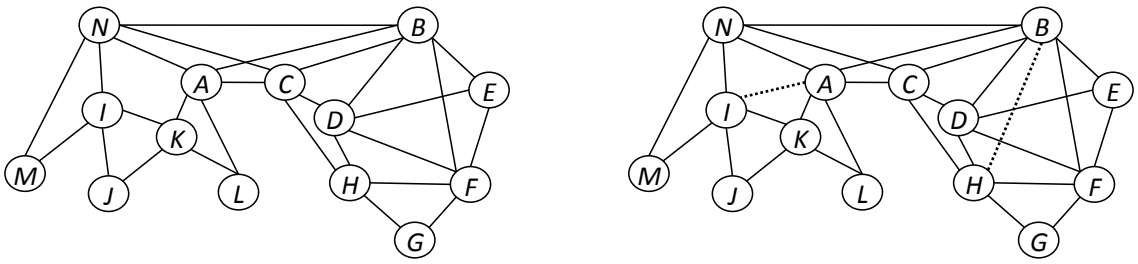

Figure 2.12: The primal *(left)* and a triangulated primal *(right)* graphs of the CSP in Figure 2.5.

[1] A perfect elimination ordering of a graph is an ordering of the vertices of the graph such that, for each vertex v, v and the neighbors of v that occur after v in the order form a clique.

[2] A graph is triangulated, or chordal, if every cycle of length four or more in the graph has an edge between two non-adjacent vertices.

At the left of Figure 2.13, we illustrate the operation of the bucket-elimination (BE) method, a fundamental variable-centered method [Dechter, 1996; 1999].

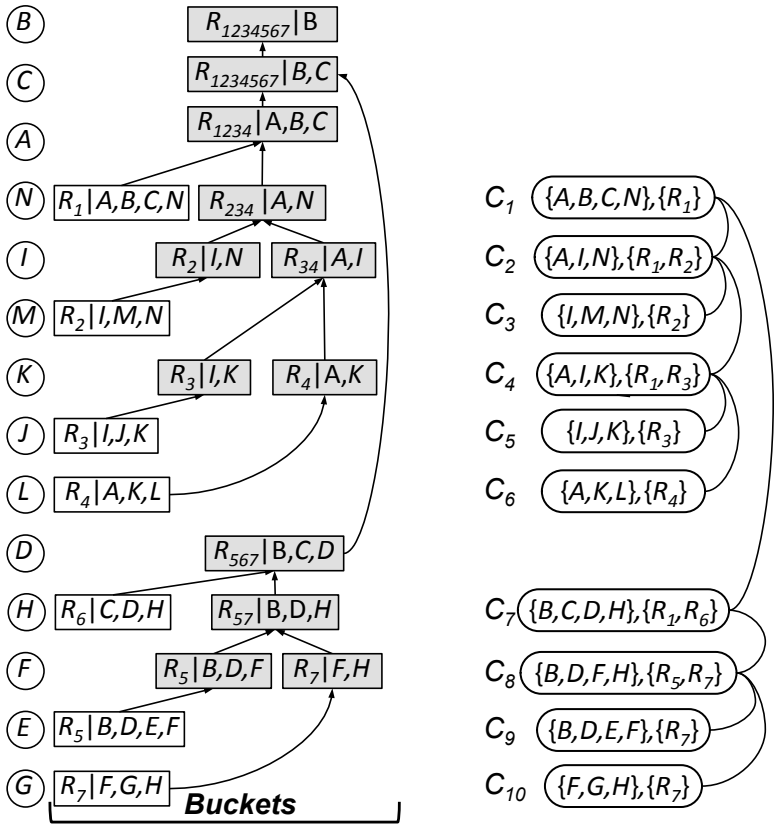

Figure 2.13: Illustrating the correspondence between a variable-based method *(left)*, bucket elimination, and the clusters of a tree decomposition *(right)* of the CSP in Figure 2.5.

The variables are listed from bottom to top following a perfect elimination ordering of the graph shown at the right of Figure 2.12. A *bucket* is associated with each variable. Each relation in the problem is placed in the bucket associated with the deepest variable in its scope. The original relations of the problem are shown with a shaded background. In this simple example, there is at most one relation from the original CSP in the bucket of a variable. Naturally, in general, there could be any number of relations in the bucket. The relations computed by inference have a shaded background. The inferred relations are generated as follows: buckets are processed

along the perfect elimination ordering; that is, from bottom to top in the example of Figure 2.13. The relations in a bucket are joined, and their join is projected out to eliminate the current variable from the relation; the resulting relation is added to the bucket of some other variable placed higher up in the ordering.

It is easy to see that variable-based and cluster-based methods are tightly related in the sense that:

- A perfect elimination order can be mapped to a chain of clusters, where each variable and its parents in the ordering form a 'cluster.' And,
- The relations generated by inference in the bucket elimination method are nothing but the materialization of the messages between clusters via the separators in the decomposition shown at the right of Figure 2.13.

2.6.3 Backtrack-search based methods

While the techniques in the above two categories are based on inference, the technique known as 'backtrack search with tree decomposition' (BTD) [Jégou and Terrioux, 2003] applied backtrack search (i.e., conditioning) on some tree decomposition of the CSP. Figure 2.14 tries to illustrate the operation of BTD on the tree decomposition of Figure 2.9.

Search follows the ordering of the variables in the clusters of the tree decomposition. While variables can be instantiated in any order inside a cluster, they may not be instantiated before any of the variables of the parent cluster. Unlike the previous two types of methods, BTD does not generate the relations of all the separators by inference. Instead, BTD generates and stores, as search proceeds, partial solutions that succeed (i.e., goods) or fail (i.e., nogoods) in order to prevent the search process from re-exploring known partial solutions. Indeed, these goods and nogoods allow

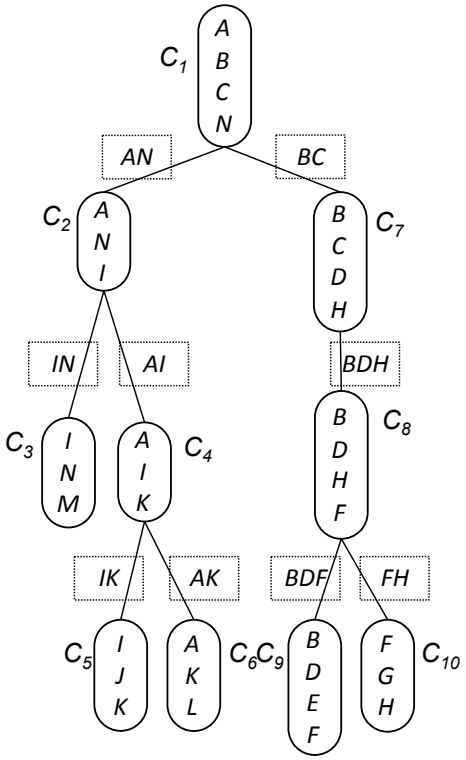

Figure 2.14: BTD on the tree decomposition at the right of Figure 2.9. The rectangles between the clusters denote 'materializations' of partial solutions that appear in a complete solution to the CSP (goods) or not (nogoods).

BTD to avoid visiting the subtrees rooted at the corresponding separator when the same partial assignments of the variables in a separator are encountered again. This memoization process is particularly important when counting the number of solutions of a CSP (see Section 2.4). BTD has been successfully used for solving CSPs [Jégou and Terrioux, 2003] and for counting the number of solutions of a CSP [Favier *et al.*, 2009].

2.6.4 Approximation techniques

Tree-decomposition methods attempt to limit the time necessary for solving CSPs by 'channeling' interactions between clusters and storing information at the separators.

The space requirements can be a serious bottleneck in practice. Therefore, approximation techniques to reduce their severity have been proposed, mainly pioneered by Dechter.

- For cluster-based methods, Mini-Cluster Tree Elimination (MCTE) [Dechter *et al.*, 2001] algorithm approximates Cluster-Tree Elimination (CTE) by partitioning the clusters into mini-clusters of manageable sizes.

- Similarly, for variable-based methods, Mini-Bucket Elimination (MBE) [Dechter, 1997; Dechter and Rish, 2003] is an approximation of Bucket Elimination (BE) [Dechter and Rish, 1994].[3] Figure 2.15 shows an example of BE and MBE applied to the same CSP. In MBE, the buckets are divided into mini-buckets of manageable sizes. MBE has been successfully used to solve optimization problems [Dechter and Rish, 2003; Marinescu and Dechter, 2007].

Another contribution of this thesis is three new strategies for bolstering the propagation of constraints across separators while reducing the space necessary for storing constraints at the separators. Those techniques are discussed in Chapter 6.

Summary

In this chapter, we defined CSPs and discussed their graphical representations and different techniques for solving them. We surveyed the main consistency properties and the algorithms for enforcing them. Then, we defined the tree decomposition of a CSP, the structural parameters of a decomposition, and the relation of these parameters to the tractability of solving a CSP. Finally, we discussed solving CSPs with tree

[3] Again, Bucket Elimination (BE) is a special case of the Cluster-Tree Elimination (CTE), where a bucket is assigned to each variable, and given a variable ordering, the messages are passed between the buckets following the ordering of the corresponding variables.

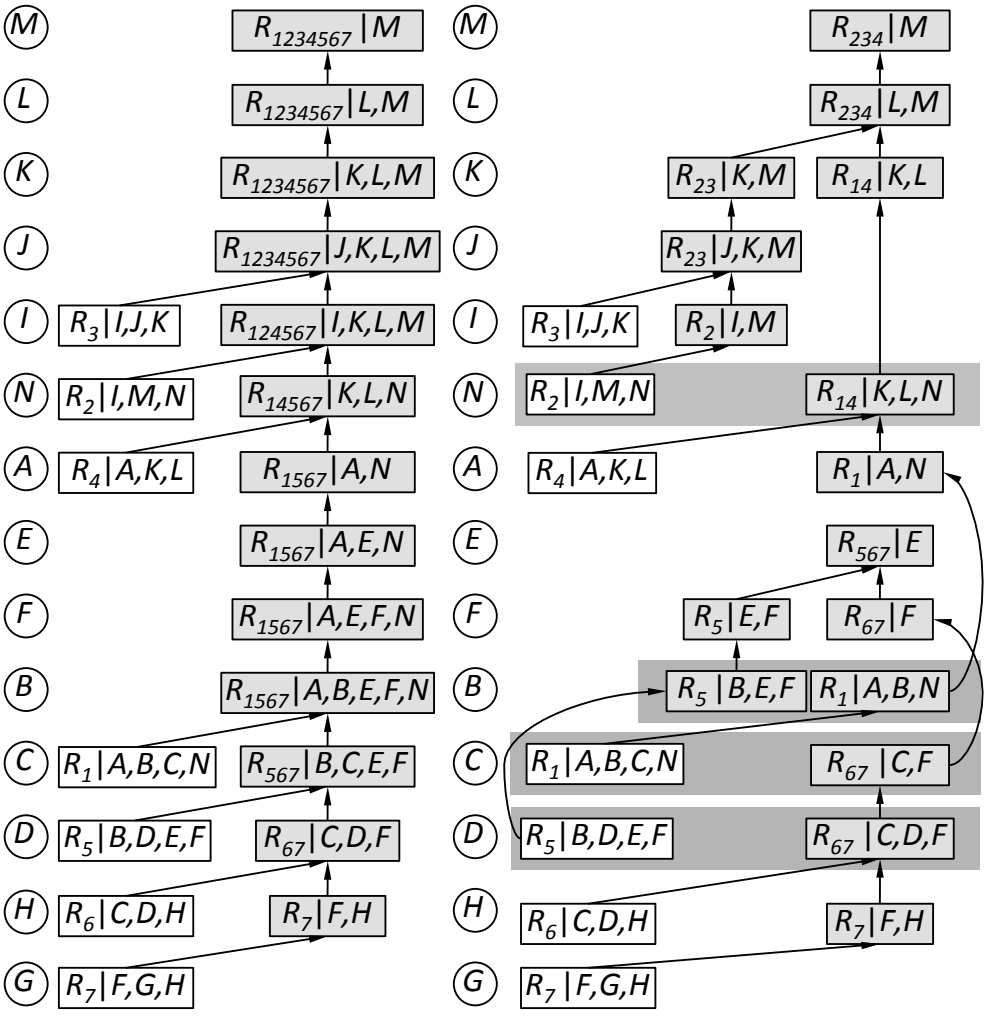

Figure 2.15: Bucket elimination *(left)* and its approximation by mini-bucket elimination (MBE) *(right)*.

decomposition, specifically, cluster-centered methods, variable-centered methods and backtrack-search based methods, in addition to approximation techniques for counting the number of solutions to a CSP.

Chapter 3

Consistency Property R(∗,m)C

Local consistency is at the heart of the success of Constraint Programming and perhaps best distinguishes this field from other scientific disciplines that study the same combinatorial problems. In this chapter, we study the relational consistency property R(∗,m)C, which is equivalent to m-wise consistency proposed in relational databases [Gyssens, 1986]. We also define wR(∗,m)C, a weaker variant of this property obtained by removing redundant edges from the dual graph of the CSP, and theoretically characterize the resulting consistency properties in terms of existing ones. We propose an algorithm for enforcing these properties on a CSP, by tightening the existing relations and without introducing new constraints, and a new data structure for facilitating its implementation. We compare the impact of our approach on the performance of problem solving with that of other consistency properties, and empirically show that wR(∗,m)C solves in a backtrack-free manner all the instances of some CSP benchmark classes, thus hinting at the tractability of those classes. Preliminary results from this chapter have been published [Karakashian *et al.*, 2010b; 2010a].

3.1 Overview

R(∗,m)C requires that every consistent assignment of variables appearing in the scope of a constraint can be extended to a consistent assignment of the variables in the scope of every $(m-1)$ other constraints. Enforcing R(∗,m)C filters existing relations but does not add any new constraint to the problem.

We borrow the notation 'relational (i,m)-consistency' from [Dechter and van Beek, 1997; Dechter, 2003], and abbreviate it to 'R(∗,m)C', where '∗' indicates that the property is concerned with only 'the scopes of the m considered constraints whatever their sizes are.' An obvious algorithm for enforcing R(∗,m)C is joining every combination of m constraints and projecting the result on their respective scopes: $\forall R_i \in \{R_1, \cdots, R_m\}, R_i = \pi_{scope(R_i)}(\bowtie_{j=1}^{m} R_j)$. The space complexity of this obvious algorithm is too prohibitive to be useful in practice. We propose an alternative algorithm that overcomes that limitation. When enforcing R(∗,m)C on every combination of m relations in the problem, much of this work is redundant and could be avoided. We introduce a weakened variant of R(∗,m)C, which we call wR(∗,m)C and obtain by removing redundant edges from the dual graph of the CSP [Dechter and Dechter, 1987; Dechter and Pearl, 1989; Janssen *et al.*, 1989; Dechter, 2003]. The contributions of this chapter are as follows:

1. The introduction and characterization of the relational consistency properties R(∗,m)C and wR(∗,m)C.

2. The design of a parameterized algorithm for enforcing those properties along with a new data structure for locating tuples in large relations.

3. The analysis of the worst-case complexity of the new algorithm.

4. The empirical evaluation of our approach on benchmark problems.

3.2 R(∗,m)C

Below, we introduce R(∗,m)C using the definition format of R(i,m)C [Dechter and van Beek, 1997].

Definition 1 *A set of m relations $\mathcal{R} = \{R_1, \cdots, R_m\}$ with $m \geq 2$ is said to be R(∗,m)C iff every tuple in each relation $R_i \in \mathcal{R}$ can be extended to the variables in $\bigcup_{R_j \in \mathcal{R}} scope(R_j) \setminus scope(R_i)$ in an assignment that satisfies all the relations in $\mathcal{R}$ simultaneously. A network is R(∗,m)C iff every set of m relations, $m \geq 2$, is R(∗,m)C.*

Informally, in every given set φ of m relations, every tuple τ in every relation $R \in \varphi$ can be extended to a tuple τ' in each $R' \in \varphi \setminus \{R\}$ such that all those tuples form a consistent solution to the relations in φ. Figure 3.1 demonstrates how every tuple in every relation is extended to a tuple in each of the $m-1$ relations. R(∗,m)C

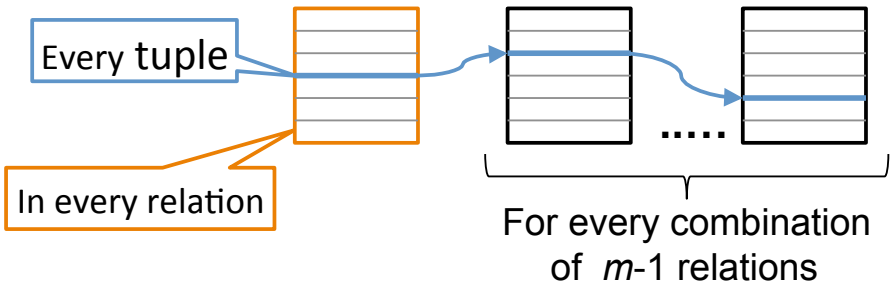

Figure 3.1: The application of R(∗,m)C on a combination of m relations.

can be enforced by filtering the existing relations and without introducing any new relations to the CSP as follows. We repeatedly apply the following operation to all combinations of m relations $\{R_1, \cdots, R_m\}$ until quiescence:

$$\forall R_i \in \{R_1, \cdots, R_m\}, R_i = \pi_{scope(R_i)}(\bowtie_{j=1}^{m} R_j) \tag{3.1}$$

Expression (3.1) gives us an obvious algorithm for R($*$,m)C, but the space requirement is prohibitive in practice. Note that π and $\bowtie$ denote the relational operators project and join, respectively.

After enforcing R($*$,m)C on a constraint network, variable domains are filtered by projecting the filtered relations on the domains of the variables. Interestingly, these domain reductions do not break the R($*$,m)C property.

Theorem 1 *If a network is R($*$,m)C, domain filtering by GAC cannot enable further constraint filtering by R($*$,m)C.*

Proof: See Appendix C.1.

Now we compare R($*$,m)C with RmC [Dechter and van Beek, 1997]. For a given set $\{R_1, \cdots, R_m\}$ of m relations, RmC requires the projection of the joined relations on all subsets $A \subseteq \bigcup_{i=1}^{m} scope(R_i)$. Hence, every subset introduces a new constraint, except those that have the same scope as existing constraints. In contrast, R($*$,m)C projects the joined relations on the scope of each of its original relations, without adding any new constraints.

Theorem 2 *RmC is strictly stronger than R($*$,m)C.*

Proof: See Appendix C.1.

We also compare R($*$,m)C with *maxRPWC* [Bessiere *et al.*, 2008]. *maxRPWC* requires that every value in every variable has a matching tuple τ in every constraint. In addition, τ should have a matching tuple in every other constraint. All the matching tuples should be valid, meaning all the values in the tuples should be alive in the domains of the corresponding variables.

Theorem 3 *R($*$,2)C is strictly stronger than maxRPWC.*

Proof: See [Bessiere *et al.*, 2008].

3.3 Weakening R(∗,m)C

We propose wR(∗,m)C, a weakened version of R(∗,m)C, which requires significantly less time and space than R(∗,m)C while slightly reducing the amount of pruning.

In the dual graph, edges enforce the equality of the shared variables of two adjacent vertices. It was observed that an edge between two vertices is *redundant* if there exists an alternate path between the two vertices such that the shared variables appear in every vertex in the path [Dechter and Dechter, 1987; Dechter and Pearl, 1989; Janssen *et al.*, 1989; Dechter, 2003]. Such redundant edges can be removed without modifying the set of solutions. Janssen *et al.* [1989] introduced an efficient algorithm for computing the *minimal dual graph* by removing redundant edges. Many minimal graphs may exist, but they are all guaranteed to have the same number of remaining edges. Figure 3.2 shows the dual graph of a CSP, where the edges drawn in dashed lines are redundant. Indeed, the same value for A is enforced between R_1 and R_3

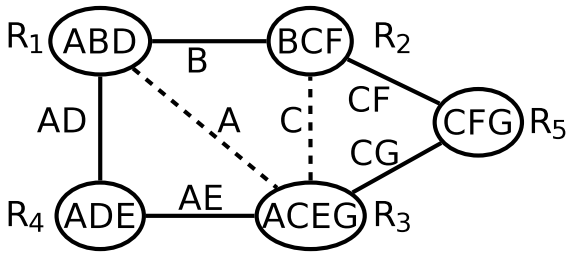

Figure 3.2: Dual graph.

through R_4, and for C between R_2 and R_3 through R_5. To enforce the R(∗,m)C property on a CSP, we must consider only combinations of relations that induce a *connected* component in the dual graph because tuples can be trivially extended to relations that do not share variables. For wR(∗,m)C, instead of using the original dual graph to generate the combinations of m relations on which to enforce the R(∗,m)C property, we propose to use the minimal dual graph obtained using the algorithm

of Janssen *et al.* [1989]. While this operation reduces the number of combinations considered (and consequently the time needed to process them and the space needed to store them), it may yield a weaker filtering of the constraints.

Definition 2 wR(∗,m)C *relative to a given minimal dual graph of a CSP* $\mathcal{P}$ *is defined as the property of* $\mathcal{P}$ *where all the combinations of* m *relations that induce connected components in the minimal dual graph verify the R(*,m)C consistency property. Note that* $m \geq 2$.

Given that, in general, more than one possible minimal dual network exists, the property obviously depends on the minimal dual graph chosen, and is always defined relative to that graph. For the sake of simplicity however, the particular minimal dual graph is not included in the notation.

Theorem 4 wR(∗,2)C *on any minimal dual graph of a CSP and* R(∗,2)C *are equivalent.*

Proof: The case where $m = 2$ corresponds to pairwise consistency and the proof is given by Janssen *et al.* [1989]. □

Theorem 5 $\forall a, b \in \mathbb{N}$ *where* $a < b \leq |\mathcal{C}|$, wR(∗,b)C *is strictly stronger than* wR(∗,a)C *on the same connected minimal dual graph of the CSP.*

Proof: See Appendix C.2.

Corollary 1 wR(∗,3)C *is strictly stronger than* R(∗,2)C *on any connected minimal dual graph of the CSP where* $|\mathcal{C}| \geq 3$.

Proof: By Theorem 4, R(∗,2)C is equivalent to wR(∗,2)C. By Theorem 5, wR(∗,3)C is stronger than wR(∗,2)C. Further, the CSP $\mathcal{P}_e$ used in the proof of Theorem 5 is R(∗,2)C but not wR(∗,3)C. □

Theorem 6 *$\forall m>2$,* R(∗,m)C *is strictly stronger than* wR(∗,m)C *on any connected minimal dual graph of the CSP.*

Proof: See Appendix C.2.

3.4 Theoretical Characterization

Theorems 3 to 6 theoretically characterize the new properties in terms of GAC, maxRPWC and relational m-consistency (denoted RmC). Those relations are illustrated Figure 3.3, where a directed edge from property p to property p' indicates that p is strictly weaker than p'.

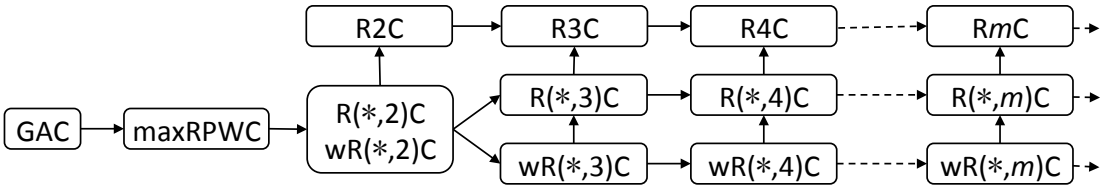

Figure 3.3: Comparing GAC, maxRPWC, R(∗,m)C, wR(∗,m)C, and RmC.

3.5 A First Algorithm for Enforcing R(∗,m)C

Expression (3.1) gives an obvious algorithm for enforcing R(∗,m)C. However, this algorithm requires computing and materializing the join of each combination of m relations, which can be prohibitive in practice. Below, we propose PerTuple, an algorithm that avoids computing and storing the intermediate joins. PerTuple uses backtrack search to identify and remove every tuple that does not verify the R(∗,m)C property. *It computes the minimal constraints in each subproblem induced by a combination of m constraints.* First, we describe initializing the queue on which

PerTuple operates, then we discuss PerTuple. After that, we describe the search for supports and the data structure we designed for this purpose.

Definition 3 *The support of a tuple τ of a relation R in a combination φ of relations, denoted $S_{\tau,\varphi}$, is a set of tuples that verifies the condition: $\forall R_i \in \varphi \setminus \{R\} \exists \tau_i \in S_{\tau,\varphi}$, $\tau_i \in R_i$ and the tuples in $S_{\tau,\varphi} \cup \{\tau\}$ agree on all shared variables.*

3.5.1 Initializing the queue

Given the dual graph (or a minimal dual graph) of a CSP, let Φ be the set of all combinations of m relations that induce connected components of the considered graph. We initialize the queue, $\mathcal{Q}$, over which our algorithm operates as follows:

- *At preprocessing*, before search, $\mathcal{Q}$ is set to all the combination-relation pairs $\langle \varphi, R \rangle$ such that $\varphi \in \Phi$ and $R \in \varphi$.
- *For lookahead*, during search, $\mathcal{Q}$ is set to all the combination-relation pairs $\langle \varphi, R \rangle$ for all relations neighboring any relation where the instantiated variable appears.

We have developed an algorithm, described in Appendix A, that computes Φ *while exploiting the topology of the considered graph.* The advantage of our algorithm is that it enumerates each connected component once and none of the non-connected components. It performs particularly well on large sparse dual graphs when m is small.

3.5.2 Processing the queue

PerTuple takes as input $\mathcal{Q}$ and Φ, see Algorithm 1. It filters the relations to enforce R($*$,m)C, and returns true if it is successful and false otherwise. When PerTuple

is executed on a single combination ϕ ($\Phi = \{\phi\}$), it computes the minimal network induced by the relations in ϕ.

Algorithm 1: PerTuple($\mathcal{Q}$,Φ).

Input: $\mathcal{Q}$ is propagation queue and Φ is the set of combinations of m constraints.
Output: *true* if the problem is R($*$,m)C, *false* otherwise

```
1  while (Q ≠ ∅) do
2      ⟨φ, R⟩ ← Pop(Q)
3      deleted ← false
4      foreach τ ∈ R do
5          support ←SearchSupport(τ, φ)
6          if support = false then
7              Delete(τ, R)
8              if R = ∅ then return false
9              deleted ← true
10     if deleted then foreach φ' ∈ (Φ \ {φ}), R ∈ φ' do
11         foreach R' ∈ (φ' \ {R}) do
12             Q ← Q ∪ {⟨φ', R'⟩}
13 return true
```

PerTuple proceeds by removing a combination-relation pair $\langle \varphi, R \rangle$ from the queue (Line 2), and searches a support in φ for each $\tau \in R$ (Line 5). A tuple that does not have a support is deleted from R (Line 7). When a relation loses its last tuple, the algorithm returns false (Line 8). If, after processing all the tuples in R, any tuples are deleted, the relations affected by the update of R are added to the queue. The affected relations are those that appear with R in a combination other than φ. Notice that a relation R' that appears in a combination with R needs to be checked only in those combinations in which it appears along with R. Therefore, when added to the queue, an affected relation R' is paired with the combination φ', other than φ, that includes both R and R' (Line 12).

3.5.3 Searching for a support

To find a support $S_{\tau,\varphi}$ for a tuple τ of a relation R in a combination φ, SEARCHSUPPORT conducts a backtrack search on the dual encoding of the CSP induced by φ. This dual CSP is denoted $\mathcal{P}_{D\varphi}$. The variables of $\mathcal{P}_{D\varphi}$ are the relations in φ. Their domains are the tuples of the relations except for the variable corresponding to R, which is assigned the tuple τ. The constraints in $\mathcal{P}_{D\varphi}$ are binary, and enforce the equality of the shared scope of the relations in φ. A solution to $\mathcal{P}_{D\varphi}$ is $S_{\tau,\varphi}$, the support set of τ in φ. The search stops at the first solution and returns $S_{\tau,\varphi}$. It returns false if no solution is found. The search process uses forward checking and dynamic variable ordering with the domain/degree heuristic.

3.5.4 The index-tree data structure

In order to effectively implement the above mentioned forward-checking, we need to locate all the tuples in a relation R_j that are consistent with a tuple τ_i of a relation R_i. For that purpose, we designed the new index-tree data structure, which we introduce below. We assume that the relations are implemented as tables of consistent tuples and that the variables are in a canonical order. Each table includes a column to indicate that the tuple is deleted (1) or not (0).

An index tree is built for each relation and each subset of its scope that is shared with another relation in the problem. Given two relations R_i and R_j and $\mathcal{X}_s$=$scope(R_i) \cap scope(R_j)$, the index tree $IT_{R_j,\mathcal{X}_s}$ returns for $\tau_i \in R_i$ all tuples $\tau_j \in R_j$ to which τ_i can be extended, that is $\pi_{\mathcal{X}_s}(\tau_i) = \pi_{\mathcal{X}_s}(\tau_j)$. An index tree $IT_{R_j,\mathcal{X}_s}$ is a rooted tree, with a dummy root, where all leaves are at height $|\mathcal{X}_s|$. The level of a node in the tree corresponds to a variable in $\mathcal{X}_s$. The nodes are labeled with values of the variables in $\mathcal{X}_s$. Each leaf node holds a list of pointers to tuples in R_j. Figure 3.4

	A	B	C	D	*deleted*
t_1	0	0	1	0	0
t_2	0	1	1	0	0
t_3	0	1	1	1	0
t_4	1	1	1	1	0

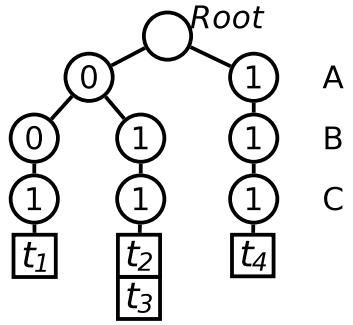

Figure 3.4: $IT_{R_j,\{A,B,C\}}$.

shows an example of an index tree for the relation R_j and $\mathcal{X}_s$=$\{A, B, C\}$.

The tree is built as follows. The tuples of R_j are sequentially inserted in the tree. For a given tuple τ_j∈R_j, we consider $\pi_{\mathcal{X}_s}(\tau_j)$. Traversing $IT_{R_j,\mathcal{X}_s}$ from the root, we match the value of a variable in $\pi_{\mathcal{X}_s}(\tau_j)$ with the label of a child of the current node in the tree. If the two values match, we move to that child node in the tree and to the value of the next variable in $\pi_{\mathcal{X}_s}(\tau_j)$. Otherwise, we add a new child node with the value of the variable in $\pi_{\mathcal{X}_s}(\tau_j)$. When the variables in $\mathcal{X}_s$ are exhausted, we insert τ_j at the end of the list at the leaf node.

When searching for the tuples in R_j that are consistent with τ_i, we traverse the tree as explained above for $\pi_{\mathcal{X}_s}(\tau_i)$. If, at a given level, no child to a tree node can be found, we conclude that no such tuple exists and return null. Otherwise, we return, from the list of pointers at the leaf, the non-deleted matching tuples.

The complexity of building the index tree is $\mathcal{O}(|\mathcal{X}_s|td)$ for time and $\mathcal{O}(|\mathcal{X}_s|t)$ for space, where t is the number of tuples in the relation and d the largest domain size of the variables in $\mathcal{X}_s$. This bound is reached when each leaf node points to a single tuple. The time complexity of a query is $\mathcal{O}(|\mathcal{X}_s|d + t)$.

3.5.5 Improving the search for support

We propose two improvements to the search for support.

When $S_{\tau,\varphi}$ is found, it is stored for the tuple-combination pair $\langle \tau, \varphi \rangle$, and reused as long as every tuple in $S_{\tau,\varphi}$ remains valid, similar to the ACS-residue algorithm [Likitvivatanavong *et al.*, 2007]. The importance of this improvement is further discussed in the complexity analysis.

Further, once $S_{\tau,\varphi}$ is found, the support of every tuple $\tau' \in S_{\tau,\varphi}$ can be directly set to be $(S_{\tau,\varphi} \cup \{\tau\}) \setminus \{\tau'\}$, thus saving SearchSupport the effort of searching for supports for all τ'. This mechanism is reminiscent of the multi-directional support of Lecoutre and Hemery [2007].

3.5.6 Improving forward checking

Another practical improvement in this work attempts to reduce the effort necessary for executing forward checking. Given that the size of relations can be large, it becomes important to check the consistency of two tuples without scanning all the relations. We have already mentioned that we use the index-tree data structure for checking the consistency of two tuples from two relations whose scopes overlap. Forward checking operates by removing from the 'future' relations those tuples that are not consistent with the current path. We call the tuples that are consistent 'valid' and those that are not 'invalid.'

For a given tuple t_i in a given relation R_i, the index-tree data structure returns all the tuples in an 'adjacent' relation that are consistent with t_i. The set of such tuples includes both valid and invalid tuples because the index-tree is not updated when forward checking invalidates tuples in future relations. Thus, the returned tuples must be scanned and only the ones considered to be valid should be considered. At some

point during the backtrack search, most of the tuples may become invalid. Hence, it may be more efficient to check the valid tuples for consistency with t_i than to check for validity of the consistent tuples returned by the index-tree. For this purpose, each relation keeps a counter of the number of 'valid' tuples and the index-tree data structure keeps a counter of the number of (valid and invalid) tuples consistent with t_i. We compare the two counters, and the smaller set is examined.

3.5.7 Complexity analysis

The time complexity of the algorithm is dominated by PerTuple, hence the initialization phase is omitted from the analysis.

Theorem 7 PerTuple *is* $\mathcal{O}(t^{m+1}e^{m+1})$.

Proof: Let t be the maximum number of tuples in a relation. It is bounded by $\mathcal{O}(d^k)$, where d is maximum domain size and k is the maximum arity of the relations. The number of constraints is e and the maximum number of combinations is $\binom{e}{m}$ and bounded by $\mathcal{O}(minimum(e^m, e^{\frac{e}{2}}))$. Below, we assume that $m<\frac{e}{2}$.

PerTuple has two nested loops. The outer loop iterates over the combination-relation pairs in $\mathcal{Q}$. The number of times that the outer loop iterates is the initial size of $\mathcal{Q}$, which is $\mathcal{O}(e^m)$, plus the number of times a combination-relation pair is added to $\mathcal{Q}$ in Line 12. A relation can participate in at most e^{m-1} combinations. Therefore, whenever a tuple is deleted $\mathcal{O}(e^{m-1})$ pairs are queued in Line 12. There are $\mathcal{O}(te)$ tuples and each tuple is deleted at most once. Thus, Line 10 is executed at most $\mathcal{O}(te)$ times, each time enqueuing $\mathcal{O}(e^{m-1})$ pairs. Therefore, the outer loop iterates at most $\mathcal{O}(te^m)$ times. The inner loop iterates over the tuples in a relation $\mathcal{O}(t)$ times. When a support for a tuple has been identified, SearchSupport costs $\mathcal{O}(m)$ to verify that every tuple in the support is still valid. When any tuple in the support

has been deleted, SearchSupport executes a backtrack search on $\mathcal{P}_{D\varphi}$. $\mathcal{P}_{D\varphi}$ has m variables of maximum domain size t, and the first variable is instantiated. Thus, the complexity of the backtrack search is $\mathcal{O}(t^{m-1})$, and that of the inner loop is $\mathcal{O}(t^m)$. Thus, PerTuple is $\mathcal{O}(t^{m+1}e^m)$. □

The time complexity of PerTuple is not worse than that of the obvious algorithm based on Expression (3.1), which is $\mathcal{O}(t^{m+1}e^{m+1})$.

When intermediate joins are not stored, the space complexity of the obvious algorithm is $\mathcal{O}(t^m)$, and constitutes a major bottleneck for its practical implementation. The space complexity of PerTuple is dominated by the space for storing the $\mathcal{O}(e^2)$ index trees, which is $\mathcal{O}(kte^2)$.

Thus, our algorithm dramatically reduces the space complexity while slightly improving the time complexity.

3.6 Related Work

The property m-wise consistency, proposed in the area of relational databases [Gyssens, 1986], requires that every tuple in a relation can be extended to a tuple in every other relation. Pairwise consistency is a special case of m-wise consistency where m=2, and is equivalent to R(∗,2)C. Janssen *et al.* [1989] proposed to enforce this consistency property by enforcing arc-consistency on the dual CSP. Importantly, they also described an algorithm for removing the redundant edges from the dual CSP to avoid revising unnecessary relation pairs. We use their redundancy removal algorithm for wR(∗,m)C. While m-wise consistency is equivalent to R(∗,m)C, to the best of our knowledge, our work is the first to propose and evaluate an algorithm for enforcing it.

Jégou [1993] proposed hyper-k-consistency, which requires the tuples in every (k-1) relations to be extendible to every k^{th} relation. Generalizing the early work on local

consistency for CSPs [Montanari, 1974; Mackworth, 1977; Jégou, 1993], Dechter and van Beek [1997] formalized *relational consistency* for non-binary CSPs in terms of *relational m-consistency* and *relational (i, m)-consistency*. Enforcing any of the above listed properties may require the addition of new constraints to the problem modifying its topology, which we avoid doing in our approach.

None of the above-listed approaches evaluates practical algorithms for enforcing the proposed properties. Next, we describe more recent approaches to relational consistency that specify and evaluate the corresponding propagation algorithms.

Stergiou and Walsh [1999] studied arc consistency on three different encodings of non-binary CSPs (i.e., the hidden variable, dual, and double encodings). Stergiou and Samaras [2005] designed specialized arc-consistency algorithms for those encodings. Their arc-consistency algorithm for the dual encoding improves performance by grouping tuples that have the same supports, but yields filtering equivalent to pairwise consistency and R(∗,2)C. While it is specialized for pairs of relations, our proposed algorithm is parameterized and applies to any number of relations. Our algorithm can benefit from the tuple grouping of Stergiou and Samaras [2005]. Further, we avoid redundant checks as proposed by Janssen *et al.* [1989], which is an improvement over the approach of [Stergiou and Samaras, 2005].

Bessiere et al. [2008] provided detailed theoretical, algorithmic, and empirical studies of domain filtering consistencies for non-binary CSPs. The consistency properties that they studied do not modify the topology of the constraint network and are restricted to combinations of two relations. Further, they are stronger than GAC (which is relational (1,1)-consistency), but are weaker than pairwise consistency followed by GAC. Our work complements and extends their approach by considering combinations of an arbitrary number of constraints and updating the constraint definitions, thus providing stronger consistency properties. In our experiments, we compare our work

against maxRPWC, which exhibits the best performance in their study.

Finally, we mention the consistency properties Conservative Path Consistency introduced by Debruyne [1999] and the stronger property Conservative Dual Consistency introduced by Lecoutre *et al.* [Lecoutre *et al.*, 2007], which do not alter the topology of the constraint graph. However, they are both restricted to binary CSPs and consider only three constraints at the same time.

3.7 Empirical Evaluations

In this section we present the empirical evaluation of wR($*$,m) on benchmark problems.

3.7.1 Experimental set-up

To evaluate the performance of our algorithm for enforcing wR($*$,m) (i.e., R($*$,m)C on the minimal dual graph), we compare it against GAC2001 [Bessiere *et al.*, 2005] and maxRPWC [Bessiere *et al.*, 2008]. All those algorithms are integrated as full lookahead strategies in a backtrack search procedure. After enforcing wR($*$,m) in the lookahead schema, we filter the domains of the uninstantiated variables by projecting the constraints on the variables. The search procedure finds the first solution of the original CSP using the domain/degree heuristic for dynamic variable ordering. During search, we timestamp the deleted tuples by the variable's instantiation. Upon backtracking, we restore all tuples that have the timestamp of the variable's instantiation.

The experiments are conducted on the benchmarks of the CSP Solver Competition[1] with a time limit of two hours per instance. We set the maximum processing time per instance to two hours for two reasons: *a*) we targeted difficult instances; and *b*) we wanted to observe the effect of stronger consistencies (i.e., backtrack-free search,

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

smaller trees) as opposed to measuring the effectiveness of our implementation. The experiments compare wR(∗,2)C, wR(∗,3)C, wR(∗,4)C, GAC, and maxRPWC.

We split the benchmark problems into three groups according to the number of nodes visited using the different consistency algorithms: The benchmarks that require many node visits with GAC but require fewer node visits with the higher levels of consistency are in the first group. The benchmarks that do not require many node visits using any of the consistency algorithms are in the second group. Lastly, the benchmarks that require many node visits using any of the consistency algorithms are in the third group.

The tables give the number of nodes visited (#Nodes), the CPU time in seconds (Time), and the maximum time (Max time) for the instances *completed* within a two-hour time limit. They also give the number of instances completed (#C), the number of instances with the fastest running time (#F), and the number of instances solved backtrack free (#BF). Time out is denoted as '-' and memory out as 'mem.' CPU time includes preprocessing. Importantly, the averages of #Nodes, Time, and Max time are computed over *only* the instances completed by *all* the compared algorithms, but algorithms that do not complete any instances are not taken into consideration. Thus, those values should be considered in light of the number of completed instances.

3.7.2 Results

The usefulness of stronger consistency is best illustrated on the unsatisfiable problems of Tables 3.1 to 3.3 and the satisfiable problems of Table 3.4. wR(∗,m)C is the fastest on most instances, and is able to solve more instances than GAC or maxRPWC[2]. In many instances, GAC takes more than 100 times the CPU time of wR(∗,m)C.

[2]Bessiere *et al.* [2008] showed that pairwise consistency (i.e., R(*,2)C) followed by GAC is strictly stronger than maxRPWC, which is strictly stronger than GAC.

In particular, many `modifiedRenault` instances are solved in a few seconds with wR($*$,m)C, but not completed in two hours by GAC. Moreover, wR($*$,m)C solves many more instances backtrack free than GAC and maxRPWC do. We emphasize that all `dag-rand` and `modifiedRenault` are solved backtrack free by wR($*$,2)C and wR($*$,4)C, respectively. Thus, wR($*$,m)C hints at the tractability of the corresponding CSP class, and constitutes another step towards empowering constraint solvers to solve problems without search, the main objective of this dissertation. Stronger consistency almost always consistently reduces the number of nodes visited, but not the CPU time. When search with a given consistency property visits relatively few nodes, enforcing a stronger property on the same instance may be overkill and wasteful. This remark holds for wR($*$,2)C and wR($*$,3)C on `dag-rand`, but not for `rand-10-20-10` where wR($*$,4)C beats all tested algorithms.

Table 3.5 for unsatisfiable problems and Tables 3.6 and 3.7 for satisfiable problems show the results of the second group of benchmarks. In these problems, all tested algorithms visit few nodes. The time for enforcing wR($*$,m)C is wasted and increases with the value of m. As for the third group in Table 3.8 for unsatisfiable problems and Tables 3.9 and 3.10 for satisfiable problems, wR($*$,m)C visits fewer nodes than both GAC and maxRPWC for most of the instances, but is not able to outperform them in terms of CPU time.

We do not report the results of R($*$,m)C for the following reasons. For $m = 2$, R($*$,2)C and wR($*$,2)C are equivalent and the latter is significantly cheaper than the former. In general, wR($*$,m)C considers significantly fewer combinations of constraints than R($*$,m)C: it scales better than and outperforms R($*$,m)C.

The goal of our experiments is to evaluate different consistency properties under similar conditions. Our solver does not implement the advanced heuristics used in the Solver Competition. Hence, we do not compare the CPU time in our experiments to

Table 3.1: Results on the unsatisfiable benchmark problems of the first group (part 1).

Algorithm	#Nodes	Time	Max time	#C	#F	#BF
aim-100 (instances: 8, vars: 100, dom: 2, rels: 173, arity: 3)						
GAC	-	-	-	0	0	0
maxRPWC	-	-	-	0	0	0
wR(∗,2)C	4,619,373.00	2,016.78	6,008.34	3	1	0
wR(∗,3)C	18,776.67	**97.36**	**282.09**	**4**	**3**	0
wR(∗,4)C	**18,685.33**	944.17	2,725.35	**4**	1	**1**
aim-200 (instances: 8, vars: 200, dom: 2, rels: 348, arity: 3)						
GAC	-	-	-	0	0	0
maxRPWC	-	-	-	0	0	0
wR(∗,2)C	-	-	-	0	0	0
wR(∗,3)C	-	-	-	0	0	0
wR(∗,4)C	**38.00**	**5.54**	**5.54**	**1**	**1**	0
aim-50 (instances: 8, vars: 50, dom: 2, rels: 84, arity: 3)						
GAC	98,475.50	**6.59**	**15.97**	8	2	0
maxRPWC	89,254.88	10.34	28.64	8	0	0
wR(∗,2)C	25,615.75	6.90	39.85	8	**3**	0
wR(∗,3)C	8,054.00	17.68	73.08	8	1	3
wR(∗,4)C	**4,019.75**	58.12	455.80	8	2	**5**
composed-25-1-2 (instances: 10, vars: 33, dom: 10, rels: 224, arity: 2)						
GAC	-	-	-	0	0	0
maxRPWC	-	-	-	0	0	0
wR(∗,2)C	-	-	-	0	0	0
wR(∗,3)C	**0.00**	**2.05**	**2.17**	**6**	6	6
wR(∗,4)C	**0.00**	14.24	14.39	**10**	4	**10**
composed-25-1-25 (instances: 10, vars: 33, dom: 10, rels: 247, arity: 2)						
GAC	-	-	-	0	0	0
maxRPWC	-	-	-	0	0	0
wR(∗,2)C	-	-	-	0	0	0
wR(∗,3)C	1.00	**2.52**	**16.89**	8	**8**	6
wR(∗,4)C	**0.00**	2.82	17.21	**10**	2	**10**
composed-25-1-40 (instances: 10, vars: 33, dom: 10, rels: 262, arity: 2)						
GAC	-	-	-	0	0	0
maxRPWC	-	-	-	0	0	0
wR(∗,2)C	-	-	-	0	0	0
wR(∗,3)C	1.56	**2.86**	**3.33**	9	**9**	6
wR(∗,4)C	**0.00**	18.13	18.42	**10**	1	**10**
composed-25-1-80 (instances: 10, vars: 33, dom: 10, rels: 302, arity: 2)						
GAC	1.00	**0.08**	**0.08**	4	2	0
maxRPWC	1.00	0.17	0.20	2	0	0
wR(∗,2)C	7.00	0.57	0.60	2	0	0
wR(∗,3)C	2.50	2.61	2.84	**10**	**8**	6
wR(∗,4)C	**0.00**	22.71	23.02	**10**	0	**10**

Table 3.2: Results on the unsatisfiable benchmark problems of the first group (part 2).

Algorithm	#Nodes	Time	Max time	#C	#F	#BF
composed-75-1-2 (instances: 10, vars: 83, dom: 10, rels: 624, arity: 2)						
GAC	-	-	-	0	0	0
maxRPWC	-	-	-	0	0	0
wR(∗,2)C	-	-	-	0	0	0
wR(∗,3)C	0.33	**6.47**	**6.68**	6	**6**	5
wR(∗,4)C	**0.00**	45.81	46.27	**10**	4	**10**
composed-75-1-25 (instances: 10, vars: 83, dom: 10, rels: 647, arity: 2)						
GAC	-	-	-	0	0	0
maxRPWC	-	-	-	0	0	0
wR(∗,2)C	-	-	-	0	0	0
wR(∗,3)C	0.33	**6.82**	**7.08**	6	**6**	5
wR(∗,4)C	**0.00**	45.84	49.09	**10**	4	**10**
composed-75-1-40 (instances: 10, vars: 83, dom: 10, rels: 662, arity: 2)						
GAC	-	-	-	0	0	0
maxRPWC	-	-	-	0	0	0
wR(∗,2)C	-	-	-	0	0	0
wR(∗,3)C	0.33	**6.82**	**7.05**	6	**6**	5
wR(∗,4)C	**0.00**	48.83	51.46	**10**	4	**10**
composed-75-1-80 (instances: 10, vars: 83, dom: 10, rels: 702, arity: 2)						
GAC	1.00	**0.15**	**0.19**	3	3	0
maxRPWC	1.00	0.27	0.35	3	0	0
wR(∗,2)C	7.00	1.26	1.28	3	0	0
wR(∗,3)C	0.67	7.69	8.09	8	**5**	5
wR(∗,4)C	**0.00**	51.85	57.70	**10**	2	**10**
dag-rand (instances: 25, vars: 23, dom: 3, rels: 16, arity: 15)						
GAC	50,570.00	5,282.70	7,127.22	20	0	0
maxRPWC	-	-	-	0	0	0
wR(∗,2)C	**0.00**	**90.15**	**105.61**	**25**	**25**	**25**
wR(∗,3)C	-	-	-	0	0	0
wR(∗,4)C	-	-	-	0	0	0
dubois (instances: 13, vars: 98, dom: 2, rels: 65, arity: 3)						
GAC	105,250,810.00	949.47	2,094.12	5	0	0
maxRPWC	105,250,810.00	1,037.58	2,296.28	5	0	0
wR(∗,2)C	7,864,312.00	**287.96**	**612.27**	**7**	**7**	0
wR(∗,3)C	7,864,312.00	818.37	1,745.33	6	0	0
wR(∗,4)C	**3,932,152.00**	1,766.67	3,797.38	4	0	0
modifiedRenault (instances: 31, vars: 111, dom: 42, rels: 130, arity: 10)						
GAC	1,171,458.43	782.32	5,259.99	9	2	0
maxRPWC	733.57	537.46	2,690.27	15	5	10
wR(∗,2)C	487.00	**5.17**	**10.96**	28	**20**	25
wR(∗,3)C	**0.00**	9.63	12.37	30	2	28
wR(∗,4)C	**0.00**	44.20	92.70	**31**	2	**31**

Table 3.3: Results on the unsatisfiable benchmark problems of the first group (part 3).

Algorithm	**#Nodes**	**Time**	**Max time**	**#C**	**#F**	**#BF**
os-taillard-4ExtConvert (instances: 10, vars: 16, dom: 256, rels: 48, arity: 2)						
GAC	47.43	**17.17**	**52.46**	9	**8**	0
maxRPWC	47.43	19.89	61.85	9	0	0
wR(∗,2)C	498.57	332.38	816.93	8	0	0
wR(∗,3)C	74.86	431.04	1,057.94	9	1	1
wR(∗,4)C	**0.00**	486.57	1,325.49	**10**	1	**9**
rand-10-20-10 (instances: 20, vars: 20, dom: 10, rels: 5, arity: 10)						
GAC	210.55	7.20	10.32	20	0	0
maxRPWC	0.55	4.45	14.36	20	0	**20**
wR(∗,2)C	**0.00**	1.27	1.39	20	0	**20**
wR(∗,3)C	**0.00**	1.17	1.24	20	0	**20**
wR(∗,4)C	**0.00**	**0.88**	**0.99**	20	**20**	**20**

that of the competition.

Table 3.4: Results on the satisfiable benchmark problems of the first group.

Algorithm	**#Nodes**	**Time**	**Max time**	**#C**	**#F**	**#BF**
aim-100 (instances: 16, vars: 100, dom: 2, rels: 307, arity: 3)						
GAC	9,459,773.00	759.65	2,891.62	15	4	1
maxRPWC	6,254,877.13	931.10	5,749.12	**16**	0	1
wR(∗,2)C	234,526.67	125.60	1,872.78	**16**	**7**	5
wR(∗,3)C	3,979.07	**19.43**	267.39	**16**	3	7
wR(∗,4)C	**559.13**	26.32	**265.34**	**16**	2	**9**
aim-200 (instances: 16, vars: 200, dom: 2, rels: 625, arity: 3)						
GAC	1,574,208.00	1,175.01	3,685.83	8	1	0
maxRPWC	1,138,576.83	2,091.46	7,194.08	8	0	0
wR(∗,2)C	28,724.00	**77.41**	**430.50**	12	**10**	4
wR(∗,3)C	4,821.33	127.08	586.33	**15**	4	8
wR(∗,4)C	**3,423.67**	954.31	4,362.43	14	1	**10**
aim-50 (instances: 16, vars: 50, dom: 2, rels: 152, arity: 3)						
GAC	15,169.13	0.93	7.17	16	**8**	1
maxRPWC	1,781.38	0.39	3.23	16	1	3
wR(∗,2)C	389.44	**0.26**	1.75	16	4	5
wR(∗,3)C	63.44	0.38	**1.33**	16	3	8
wR(∗,4)C	**55.06**	2.33	8.06	16	0	**10**
modifiedRenault (instances: 19, vars: 110, dom: 42, rels: 128, arity: 10)						
GAC	422,693.29	108.52	1,353.01	17	**14**	5
maxRPWC	1,339.47	99.12	361.15	18	0	8
wR(∗,2)C	211.53	**4.98**	**8.32**	**19**	5	7
wR(∗,3)C	110.35	13.33	16.69	**19**	0	14
wR(∗,4)C	**110.24**	81.28	106.84	**19**	0	**16**
rand-8-20-5 (instances: 20, vars: 20, dom: 5, rels: 18, arity: 8)						
GAC	5,976.00	**140.24**	**140.24**	16	1	0
maxRPWC	5,979.00	4194.12	4194.12	1	0	0
wR(∗,2)C	**535.00**	197.72	197.72	**20**	**19**	0
wR(∗,3)C	-	-	-	0	0	0
wR(∗,4)C	-	-	-	0	0	0

Table 3.5: Results on the unsatisfiable benchmark problems of the second group.

Algorithm	**#Nodes**	**Time**	**Max time**	**#C**	**#F**	**#BF**
ogdVg (instances: 49, vars: 166, dom: 26, rels: 25, arity: 20)						
GAC	**3.22**	**1.24**	**3.77**	**24**	**20**	**4**
maxRPWC	**3.22**	1.50	4.46	**24**	4	**4**
wR(∗,2)C	8.78	31.26	88.11	17	0	**4**
wR(∗,3)C	8.78	1,716.97	5,853.13	9	0	**4**
wR(∗,4)C	-	-	-	0	0	0
os-taillard-5ExtConvert (instances: 26, vars: 25, dom: 356, rels: 100, arity: 2)						
GAC	**185.00**	**304.26**	**481.45**	**4**	**4**	0
maxRPWC	**185.00**	456.75	724.23	3	0	0
wR(∗,2)C	2985.50	4,743.62	5,067.99	2	0	0
wR(∗,3)C	-	-	-	0	0	0
wR(∗,4)C	-	-	-	0	0	0
QCP-10 (instances: 5, vars: 100, dom: 10, rels: 822, arity: 2)						
GAC	491.00	**0.81**	**1.94**	**5**	**5**	0
maxRPWC	491.00	1.73	4.20	**5**	0	0
wR(∗,2)C	640.67	2.92	5.90	**5**	0	0
wR(∗,3)C	291.67	9.40	16.52	**5**	0	0
wR(∗,4)C	**249.00**	51.53	90.00	3	0	0

Table 3.6: Results on the satisfiable benchmark problems of the second group (part 1).

Algorithm	#Nodes	Time	Max time	#C	#F	#BF
allIntervalSeriesExtConvert (instances: 23, vars: 51, dom: 80, rels: 1078, arity: 3)						
GAC	**26.38**	**0.14**	**0.99**	**23**	**22**	**23**
maxRPWC	**26.38**	0.61	5.37	**23**	3	**23**
wR(∗,2)C	**26.38**	4.51	40.66	21	0	21
wR(∗,3)C	**26.38**	73.37	719.57	19	0	19
wR(∗,4)C	26.56	598.46	5,949.47	16	0	14
composed-25-10-20 (instances: 10, vars: 105, dom: 10, rels: 620, arity: 2)						
GAC	123.60	**0.20**	**0.25**	**6**	**6**	0
maxRPWC	123.60	0.46	0.51	**6**	0	0
wR(∗,2)C	154.80	1.50	1.64	**6**	0	0
wR(∗,3)C	139.80	11.46	11.86	5	0	0
wR(∗,4)C	**118.80**	75.12	80.56	**6**	1	0
ogdVg (instances: 16, vars: 37, dom: 26, rels: 12, arity: 9)						
GAC	**21.25**	**0.10**	**0.19**	**16**	4	**7**
maxRPWC	**21.25**	**0.10**	**0.19**	**16**	**12**	**7**
wR(∗,2)C	22.50	2.07	3.17	**16**	2	4
wR(∗,3)C	22.50	186.62	376.52	10	0	4
wR(∗,4)C	**21.25**	1,518.92	2,725.11	4	0	4
os-taillard-4ExtConvert (instances: 20, vars: 16, dom: 285, rels: 48, arity: 2)						
GAC	35.08	**13.51**	**138.75**	17	**14**	**13**
maxRPWC	35.08	14.86	153.68	17	1	**13**
wR(∗,2)C	253.92	193.86	2,031.80	15	0	10
wR(∗,3)C	116.75	737.45	3,238.58	**18**	4	10
wR(∗,4)C	**18.92**	2,412.51	5,724.01	**18**	1	7
os-taillard-5ExtConvert (instances: 4, vars: 25, dom: 337, rels: 100, arity: 2)						
GAC	**945.00**	**293.56**	**407.15**	**2**	**1**	0
maxRPWC	**945.00**	392.20	609.05	**2**	**1**	0
wR(∗,2)C	5,510.50	2,994.14	5,453.29	**2**	0	0
wR(∗,3)C	-	-	-	1	**1**	**1**
wR(∗,4)C	-	-	-	0	0	0
primes-10ExtConvert (instances: 12, vars: 100, dom: 28, rels: 50, arity: 5)						
GAC	97.08	0.54	2.30	12	8	12
maxRPWC	97.08	**0.52**	**2.07**	12	**9**	12
wR(∗,2)C	73.42	2.57	10.54	12	0	12
wR(∗,3)C	73.42	14.32	80.33	12	0	12
wR(∗,4)C	**65.42**	59.52	504.74	12	0	12
primes-15ExtConvert (instances: 8, vars: 100, dom: 46, rels: 50, arity: 4)						
GAC	98.13	0.10	0.28	8	5	8
maxRPWC	98.13	**0.09**	**0.26**	8	**7**	8
wR(∗,2)C	**71.38**	0.37	1.14	8	0	8
wR(∗,3)C	**71.38**	0.51	1.50	8	0	8
wR(∗,4)C	**71.38**	1.04	3.49	8	0	8

Table 3.7: Results on the satisfiable benchmark problems of the second group (part 2).

Algorithm	#Nodes	Time	Max time	#C	#F	#BF
primes-20ExtConvert (instances: 8, vars: 100, dom: 70, rels: 50, arity: 4)						
GAC	98.13	**0.31**	0.85	8	5	8
maxRPWC	98.13	0.32	**0.84**	8	**6**	8
wR(∗,2)C	**71.38**	1.34	3.89	8	0	8
wR(∗,3)C	**71.38**	2.96	8.20	8	0	8
wR(∗,4)C	**71.38**	6.74	24.83	8	0	8
QCP-10 (instances: 10, vars: 100, dom: 10, rels: 822, arity: 2)						
GAC	727.80	**0.85**	**4.74**	10	**10**	**4**
maxRPWC	727.80	1.61	9.02	10	0	**4**
wR(∗,2)C	832.90	2.77	12.51	10	0	2
wR(∗,3)C	661.20	11.26	45.42	10	0	2
wR(∗,4)C	**551.60**	53.91	214.91	10	0	2
QWH-10 (instances: 10, vars: 100, dom: 10, rels: 756, arity: 2)						
GAC	146.30	**0.21**	**0.36**	10	**10**	**3**
maxRPWC	146.30	0.27	0.49	10	0	**3**
wR(∗,2)C	153.60	1.00	1.39	10	0	2
wR(∗,3)C	148.70	3.28	5.03	10	0	2
wR(∗,4)C	**137.20**	13.03	18.08	10	0	2
renault (instances: 2, vars: 101, dom: 42, rels: 113, arity: 10)						
GAC	101.00	**1.00**	**1.01**	2	**2**	2
maxRPWC	101.00	94.66	94.74	2	0	2
wR(∗,2)C	**99.00**	3.97	4.00	2	0	2
wR(∗,3)C	**99.00**	12.98	13.04	2	0	2
wR(∗,4)C	**99.00**	84.15	87.80	2	0	2
ssa (instances: 6, vars: 2631, dom: 2, rels: 4721, arity: 6)						
GAC	**1,372.20**	**0.26**	**0.62**	5	**5**	**4**
maxRPWC	**1,372.20**	0.31	0.80	5	1	**4**
wR(∗,2)C	1,916.60	2.10	5.34	5	0	2
wR(∗,3)C	1,916.60	3.04	8.93	5	0	2
wR(∗,4)C	1,917.40	7.49	27.76	**6**	1	2

Table 3.8: Results on the unsatisfiable benchmark problems of the third group.

Algorithm	#Nodes	Time	Max time	#C	#F	#BF
pret (instances: 8, vars: 105, dom: 2, rels: 70, arity: 3)						
GAC	12,198,226.00	**103.47**	**109.05**	4	**4**	0
maxRPWC	12,198,226.00	140.19	140.42	4	0	0
wR(∗,2)C	13,583,698.00	365.85	367.81	4	0	0
wR(∗,3)C	**4,986,706.00**	429.11	429.85	4	0	0
wR(∗,4)C	**4,986,706.00**	800.82	804.26	4	0	0
rand-3-20-20 (instances: 25, vars: 20, dom: 20, rels: 58, arity: 3)						
GAC	69,956.40	**383.95**	**723.86**	**24**	**24**	0
maxRPWC	69,404.80	3,153.09	6,316.88	12	0	0
wR(∗,2)C	**36,509.90**	3,343.66	6,112.08	12	0	0
wR(∗,3)C	-	-	-	0	0	0
wR(∗,4)C	-	-	-	0	0	0
ssa (instances: 2, vars: 897, dom: 2, rels: 1081, arity: 5)						
GAC	244,086.00	**11.51**	**11.51**	**1**	**1**	0
maxRPWC	244,086.00	15.84	15.84	**1**	0	0
wR(∗,2)C	**21,406,446**	3,520.82	3,520.82	**1**	0	0
wR(∗,3)C	-	-	-	0	0	0
wR(∗,4)C	-	-	-	0	0	0
travellingSalesman-25 (instances: 7, vars: 76, dom: 1001, rels: 350, arity: 3)						
GAC	**138,985.75**	**1,741.48**	**2,208.56**	**4**	**4**	0
maxRPWC	**138,985.75**	3,760.28	4,412.96	**4**	0	0
wR(∗,2)C	186,512.75	4,492.42	5,327.00	**4**	0	0
wR(∗,3)C	-	-	-	0	0	0
wR(∗,4)C	-	-	-	0	0	0
varDimacs (instances: 5, vars: 74, dom: 2, rels: 322, arity: 10)						
GAC	66,064.00	**7.32**	**13.65**	**4**	**4**	0
maxRPWC	66,064.00	13.03	24.30	**4**	0	0
wR(∗,2)C	72,307.00	19.07	35.35	3	0	0
wR(∗,3)C	72,307.00	98.16	182.37	3	0	0
wR(∗,4)C	**21,435.00**	313.09	582.58	2	0	0

Table 3.9: Results on the satisfiable benchmark problems of the third group (part 1).

Algorithm	#Nodes	Time	Max time	#C	#F	#BF
bqwh-15-106 (instances: 100, vars: 106, dom: 7, rels: 593, arity: 2)						
GAC	5,323.21	**3.84**	**26.83**	100	**100**	0
maxRPWC	5,323.21	8.88	62.38	100	0	0
wR(∗,2)C	6,050.88	17.03	116.33	100	0	0
wR(∗,3)C	4,409.51	55.82	356.73	100	0	0
wR(∗,4)C	**1,964.50**	140.79	878.88	100	0	0
bqwh-18-141 (instances: 100, vars: 141, dom: 8, rels: 878, arity: 2)						
GAC	75,588.53	**76.24**	**532.13**	**100**	**100**	0
maxRPWC	75,588.53	159.70	1,067.85	**100**	0	0
wR(∗,2)C	85,944.91	294.30	2,068.03	99	0	0
wR(∗,3)C	57,672.90	794.33	3,841.06	93	0	0
wR(∗,4)C	**23,568.94**	2,118.25	6,992.22	82	0	0
driver (instances: 7, vars: 352, dom: 12, rels: 7201, arity: 2)						
GAC	**11,667.50**	**52.53**	**130.48**	**7**	**5**	1
maxRPWC	**11,667.50**	115.63	288.26	**7**	1	1
wR(∗,2)C	12,173.25	72.07	188.91	**7**	2	1
wR(∗,3)C	12,140.25	306.73	806.13	5	0	1
wR(∗,4)C	12,088.50	1,278.97	3,353.11	4	0	1
QCP-15 (instances: 6, vars: 225, dom: 15, rels: 2519, arity: 2)						
GAC	9,978.00	**44.37**	**85.93**	**6**	**6**	0
maxRPWC	9,978.00	85.69	167.19	**6**	0	0
wR(∗,2)C	11,493.67	76.48	148.09	6	0	0
wR(∗,3)C	10,487.67	262.00	413.35	4	0	0
wR(∗,4)C	**9,353.00**	1,531.16	2,942.51	3	0	0
QWH-15 (instances: 10, vars: 225, dom: 15, rels: 2324, arity: 2)						
GAC	22,570.83	**75.35**	**247.01**	**10**	**10**	0
maxRPWC	22,570.83	154.01	495.67	**10**	0	0
wR(∗,2)C	25,650.67	154.96	490.50	**10**	0	0
wR(∗,3)C	23,096.00	646.23	1,917.15	8	0	0
wR(∗,4)C	**9,449.67**	1,734.28	3,379.88	6	0	0
rand-3-20-20 (instances: 25, vars: 20, dom: 20, rels: 59, arity: 3)						
GAC	38,768.00	**281.78**	**281.78**	**25**	**25**	0
maxRPWC	38,636.00	2,274.60	2,274.60	20	0	0
wR(∗,2)C	20,923.00	1,238.25	1,238.25	20	0	0
wR(∗,3)C	5,890.00	2,589.74	2,589.74	6	0	0
wR(∗,4)C	**618.00**	5,318.61	5,318.61	1	0	0

Table 3.10: Results on the satisfiable benchmark problems of the third group (part 2).

Algorithm	**#Nodes**	**Time**	**Max time**	**#C**	**#F**	**#BF**
rand-3-20-20-fcd (instances: 50, vars: 20, dom: 20, rels: 58, arity: 3)						
GAC	7,852.00	**37.30**	**75.63**	**50**	**50**	0
maxRPWC	7,778.67	369.01	777.57	38	0	0
wR(∗,2)C	1,877.00	125.47	191.77	40	0	0
wR(∗,3)C	1,314.00	656.73	927.07	16	0	0
wR(∗,4)C	**560.33**	4,089.56	6,111.99	3	0	0
travellingSalesman-20 (instances: 15, vars: 61, dom: 1001, rels: 230, arity: 3)						
GAC	**3,649.20**	**18.96**	**70.18**	**15**	**15**	**1**
maxRPWC	**3,649.20**	39.41	143.36	**15**	0	**1**
wR(∗,2)C	4,617.20	59.72	197.40	**15**	0	0
wR(∗,3)C	4,541.90	303.98	1,011.68	13	0	0
wR(∗,4)C	4,484.50	1,632.82	5,474.64	10	0	0
travellingSalesman-25 (instances: 8, vars: 76, dom: 1001, rels: 350, arity: 3)						
GAC	**3,889.00**	**36.23**	**48.44**	**8**	**8**	0
maxRPWC	**3,889.00**	89.25	105.02	**8**	0	0
wR(∗,2)C	5,054.00	126.55	151.14	**8**	0	0
wR(∗,3)C	5,054.50	668.48	785.84	6	0	0
wR(∗,4)C	5,025.50	3,814.75	4,582.45	4	0	0
varDimacs (instances: 4, vars: 1141, dom: 2, rels: 1226, arity: 5)						
GAC	**1,052.75**	**0.15**	**0.22**	4	**4**	**4**
maxRPWC	**1,052.75**	0.17	0.25	4	1	**4**
wR(∗,2)C	1,053.75	0.94	2.16	4	0	2
wR(∗,3)C	1,053.50	1.55	3.16	4	0	2
wR(∗,4)C	1,053.25	4.75	8.64	4	0	3

3.7.3 Conclusions

The empirical evaluation showed that we can approach practical tractability using R($*$,m)C on many benchmarks, and achieve practical tractability when the level of the consistency is higher than the structural parameter of the corresponding constraint network. We noticed that there are three groups of problems. Using higher levels of consistency on the problems of the first group, we can approach practical tractability for solving them. Unlike the problems in the first group, we can achieve practical tractability on the problems of the second group with low consistency levels, and thus higher levels are not necessary. Finally, there are problems that the higher level consistency studied in this chapter is insufficient to approach practical tractability in solving them. These are the problems in the third group.

Our algorithm can be further improved by reducing redundant consistency checks, for example by grouping tuples [Stergiou and Samaras, 2005] or exploiting complex residual supports [Likitvivatanavong *et al.*, 2007; Lecoutre and Hemery, 2007; Lecoutre *et al.*, 2008]. Other interesting avenues for future work are to exploit the tightness of the constraints to avoid considering ineffective combinations of relations and to design techniques that automatically identify the level of consistency necessary for a given problem.

Summary

In this chapter, we studied the relational consistency property R($*$,m)C, proposed a weaker variant of it, wR($*$,m)C, and presented PerTuple, a parameterized algorithm for enforcing it. Our algorithm operates by tightening the existing constraints, without adding new ones. To demonstrate its usefulness, we evaluated it against algorithms for GAC and maxRPWC. Our experiments showed that by maintaining a stronger

consistency, the performance of search can be improved by two orders of magnitude on many benchmark problems. Several instances were solved in a backtrack-free manner, hinting at the tractability of the corresponding problem class. We were able to achieve practical tractability on problems that belonged to tractable problem classes, and for other problems, we approached practical tractability by significantly reducing the amount of backtracking.

Chapter 4

An Alternative Algorithm for Enforcing R(∗,m)C

In this chapter, we introduce AllSol, another algorithm for enforcing R(∗,m)C as an alternative to PerTuple (Algorithm 1) discussed in Chapter 3. While both algorithms compute the same result (i.e., the minimal relations of a CSP), we argue that their performances vary depending on the 'type' of problems to which they are applied. We use machine learning techniques to build a decision tree that predicts which of the two algorithms is more appropriate to apply to a given CSP instance based on a set of parameters that assess various aspects of the considered instance. Finally, we combine the two algorithms and the decision tree into a hybrid solver, and empirically evaluate the advantages of our approach. Preliminary results from this chapter appeared in technical report [Karakashian *et al.*, 2012].

4.1 Background

Montanari defined the *minimal network* of a CSP [1974]. The formal definition is as follows [Dechter, 2003].

Definition 4 *Given a CSP $\mathcal{P}_0$, let $\{\mathcal{P}_1, \ldots, \mathcal{P}_l\}$ be the set of all networks equivalent to $\mathcal{P}_0$. Then the minimal network M of $\mathcal{P}_0$ is defined by $M(\mathcal{P}_0) = \cap_{i=1}^{l} \mathcal{P}_i$.*

Informally stated, the minimal network is the network where the relations are as tight as can be; that is, each tuple in a relation can be extended to a solution to the CSP.

Gottlob argued that when a CSP has this property, a number of NP-hard queries can be answered in polynomial time, but also showed that *a)* deciding whether or not a constraint network is minimal is $\mathcal{NP}$-complete and *b)* finding a solution to a minimal network is also $\mathcal{NP}$-complete [Gottlob, 2011], thus proving earlier conjectures by Dechter and Pearl [1992].

PerTuple, (Algorithm 1) introduced in Chapter 3, is a first algorithm for enforcing R(∗,m)C. In essence, the algorithm computes the minimal network of the problem induced by *every set* of m relations of the CSP. When the input to PerTuple is restricted to the relations of a *single* combination of m relations, PerTuple computes the minimal network induced by those m relations.

In this chapter, we propose AllSol (Algorithm 2) as an alternative algorithm for enforcing the same property. Both PerTuple and AllSol use backtrack search to verify whether or not a given tuple appears in a solution to the problem, thus yielding the minimal network of a CSP. However, the former repeats a 'satisfiability' search (i.e., stopping after finding the first solution) for every tuple in every relation, in the worst case. The latter AllSol carries out a single 'solution counting' search generating a sufficient number of solutions to cover all the tuples of the minimal

network (i.e., possibly exploring the entire search space). The two search mechanisms are contrasted in Figures 4.1 and 4.2, respectively.

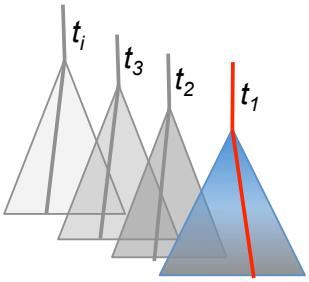

Figure 4.1: PerTuple conducts many backtrack searches, seeking one solution (satisfiability).

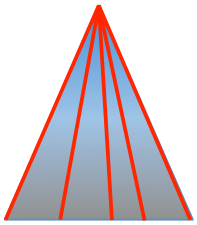

Figure 4.2: AllSol conducts a single backtrack search, possibly seeking all solutions.

The performances of both algorithms depend on whether many solutions exist and are easy to find (favoring PerTuple), or whether solutions are rare and hard to find (favoring AllSol, which traverses the search space only once).

We propose a set of parameters to predict the size of the search space and the difficulty of solving a given instance. Using machine learning techniques, we build a decision tree to select the appropriate solver to use PerTuple or AllSol. The contributions of this chapter are as follows:

1. The presentation of an alternative algorithm for enforcing R($*$,m)C.

2. The identification of CSP parameters that can be computed in polynomial time, and their use in building a decision tree that predicts the appropriate algorithm.

3. The creation of a hybrid algorithm that automatically determines whether to use PerTuple or AllSol.

4. The empirical evaluation of the hybrid algorithm for computing the minimal network of benchmark problems.

4.2 Related Work

The "algorithm selection problem" was discussed at length by Rice [1976] and has recently witnessed a surge of successful implementations under the label of "algorithm portfolio." An excellent historical review of the topic can be found by Xu *et al.* [2008]. These authors introduced SATzilla, a wildly successful portfolio algorithm for solving SAT problems. SATzilla uses 48 features, computed from 16 parameters of SAT problems, to choose between seven SAT solvers. We use 12 attributes, computed from five CSP parameters, to choose between two algorithms. Our choice of attributes is sufficient for our task, which is simpler than SATzilla's. Importantly, the time for extracting and computing the features in our case is negligible compared to the time taken by computing the minimal network using either algorithm.

4.3 AllSol

Below, we introduce AllSol, which computes the minimal network of a CSP given in its dual representation, $\mathcal{P}_D$. We then qualitatively compare AllSol and PerTuple.

4.3.1 AllSol: Solving a single counting problem

Algorithm 2 describes the operation of AllSol.

First, Algorithm 2 initializes the tuple flags to 'false'. Then it proceeds with the single backtrack search by calling BTsearchNextSol in Line 5. However, it does not stop after the first solution. It continues in the loop of Line 4 until all the solutions are found. Note that only the first call to BTsearchNextSol starts a backtrack search, and the subsequent calls only advance the backtrack search to the next solution. Every time a solution is found, all the tuples in the solution are flagged as belonging

Algorithm 2: AllSol($\mathcal{P}_D$)

Input: $\mathcal{P}_D$ the dual representation of a CSP.
Output: Minimal relations of $\mathcal{P}_D$.

```
1  foreach R_i ∈ P_D do
2      foreach τ_i ∈ R_i do  flag[τ_i] ← false
3  sol ← true
4  while sol = true do
5      sol ←BTsearchNextSol(P_D)
6      if sol ≠ false then
7          foreach τ_i ∈ sol do  flag[τ_i] ← true
8  foreach R_i ∈ P_D do
9      foreach τ_i ∈ R_i do
10         if flag[τ_i] = false then  Delete(τ_i)
```

to the minimal network in Line 7. Like PerTuple, AllSol uses forward checking. Finally, it deletes all the tuples that were not flagged in Line 10.

An important improvement allows us to interrupt search before traversing the entire space (which would be necessary in search for solution counting). After every step in the search and after executing forward checking, the domains of the future 'variables' (in fact, the relations of the original CSP) are considered. If all the 'surviving' tuples are flagged as belonging to the minimal network, as well as all the tuples in the current path, then the search resumes from that path as if it was a dead-end. At the end of search, which may or may not cover all solutions, all flagged false tuples are removed from the relations.

4.3.2 Complexity analysis

Given a CSP $\mathcal{P}_D$ in its dual representation, with e relations, t tuples per relation, and total number of tuples $T = et$. AllSol conducts a single backtrack search on the e variables of $\mathcal{P}_D$, and its worst-case time complexity is thus $\mathcal{O}(t^e)$. It is outlined in

Algorithm 2. Both AllSol and PerTuple may build more solutions than strictly needed for computing the minimal network. Indeed, we prove the following theorem in Appendix B:

Theorem 8 *Given a CSP, the problem that answers the following question is $\mathcal{NP}$-Complete: is there a set of at most k solutions such that every tuple in every relation of the minimal CSP appears in at least one solution?*

Proof sketch. See Appendix C.3.

Therefore, we will likely have to find more than the minimum number of solutions necessary for 'covering' the tuples in the minimal network.

In practice, our implementations of AllSol and PerTuple scale well with the domain sizes of the dual variable (i.e., the number of support tuples in the relations). Thanks to the index-tree structures, we could easily handle relations with 150,000 tuples.

4.3.3 Qualitative comparison of PerTuple and AllSol

Consider a network of e relations, and t tuples per relation. In order to compute the minimal network, PerTuple solves $O(et)$ times a *satisfiability* problem of size $O(t^{e-1})$. Thus, its time complexity is $O(et^e)$. In contrast, AllSol solves a *solution counting* problem of size $O(t^e)$ exactly once, and its time complexity is $O(t^e)$. Relating the worst-case time complexities of the two algorithms, their behaviors may be more clearly characterized thanks to the phase transition phenomenon observed on CSPs [Cheeseman *et al.*, 1991]. We note that AllSol and PerTuple differ in two main aspects:

1. The cost of each backtrack search, and

2. The number of times a new search is started.

AllSol conducts a single search, but searches the entire space (Figure 4.2). PerTuple conducts a search once for each tuple in the problem, but each search stops after finding the first solution (Figure 4.1). Now, back to the phase-transition. According to that macro-characterization of CSPs, we distinguish three main 'regimes:'

- *High solution density:* When a problem instance is located in the area where the existence of a solution is highly likely, solutions abound and are easy to find. In those conditions, each call to PerTuple is likely to terminate successfully and quickly. Even with repetitive calls to search, PerTuple remains quick. On the other hand, although it is sweeping only once through the search space, AllSol is likely to easily get 'overwhelmed' enumerating the large number of solutions. In that area, PerTuple is likely significantly more efficient than AllSol.

- *High nogood density:* When a problem instance is located in the area where the existence of a solution is highly unlikely, a search procedure with decent lookahead is likely to effectively prune the tree, quickly terminating the search. Even though PerTuple starts many more searches than AllSol does, both algorithms are likely to quickly traverse the same 'barren' space and their performances are comparable.

- *Low solution density:* The difference between the two algorithms arises around the area of the phase transition. An instance in that area is likely to have many 'almost' solutions [Cheeseman *et al.*, 1991]. AllSol traverses the space once. It may struggle to find the few solutions, if any, as one expects to be the case at the phase transition. However, the real misfortune is for PerTuple, because

it may have to repeat the same costly process for every tuple in each relation, which may render it totally unusable in practice.

In summary, while PerTuple is likely to be quite cheap more often than AllSol, when it encounters instances around the phase transition, it is unlikely to terminate within a set time limit even when AllSol does. The experiments reported below confirm the above interpretation.

4.4 Building a Hybrid Solver

As stated above, we expect, grossly speaking, the two algorithms to be 'complementary' in terms of their effectiveness in practice despite the fact that, there are problems too hard for either algorithm, and others easy for both. Our goal is to build a hybrid solver that adaptively chooses the 'best' algorithm to use or, at least, avoids the algorithm that does not terminate. The hybrid solver consists of:

1. The two algorithms AllSol and PerTuple,
2. A set of parameters to compute for each problem instance given an input (Section 4.4.2), and
3. A 'quick' but discriminating classifier (Section 4.4.3).

The hybrid solver computes the values of the parameters, gives them to the classifier, which then determines whether to use PerTuple or AllSol. Below, we describe the sample data, the problem parameters and features used, and the classifiers built. We then discuss the evaluation of the two resulting hybrid solvers on the benchmarks used to build the classifiers and on randomly generated problems that were not part of the training data.

4.4.1 Data used for building the classifiers

We drew the sample data from 1,616 instances from 61 benchmarks of the CSP Solver Competition.[1] Because the ultimate goal of this research endeavor is to compute the minimal network of each cluster of a tree decomposition of a CSP (see Chapters 5 and 6), we generated a tree decomposition of each problem instance, and considered each cluster in the tree decomposition as an independent problem instance (see Section 5.1). The characteristics of the instances extracted from the benchmarks and those used are shown in Table 4.1.

Table 4.1: Summary of data used.

Original Data	
Number of instances drawn from benchmarks	60,734
Number of instances solved by AllSol	53,083
Number of instances solved by PerTuple	58,444

Data Used in Study ($\lvert\delta_{\text{time}}\rvert \geq 256$ msec)				
Timeout per instance			30 minutes	
Number of instances solved by AllSol: $\mathcal{A}$			15,872	
Number of instances solved by PerTuple: $\mathcal{P}$			21,233	
Number of instances in $\mathcal{A} \setminus \mathcal{P}$			1	
Number of instances in $\mathcal{P} \setminus \mathcal{A}$			5,362	
Total number of instances used: $\mathcal{A} \cup \mathcal{P}$			21,234	
	Min	**Max**	**Avg**	**Median**
Number of variables	3	213	38.46	28
Domain size	2	238	15.12	7
Number of relations	2	2,069	218.12	153
Arity of relations	2	16	2.82	2
Number of tuples per relation	3	150,000	4253.13	25

We computed the minimal network of all 60,734 instances extracted using PerTuple and AllSol, and recorded the time taken by each algorithm. Neither algorithm consistently outperformed the other, but PerTuple was faster on more instances than AllSol was (20,400 instances versus 9,369 instances). We chose to ignore all instances

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

on which the execution of the two algorithms differed by less than 256 milliseconds, which we estimate to be, in our context, an insignificant time difference. Typically, the ignored instances are either 'easily' solved by both algorithms or solved by neither algorithm. In this section, when we say 'solved' we mean computed the minimal network within the time limit of 30 minutes. We set the time limit to 30 minutes to maintain the duration of the experiment on 65,894 instances within reasonable limits. To avoid overshadowing the differences between the two algorithms caused by the benchmark distribution, we partitioned the 21,234 remaining instances into two sets. $\mathcal{P}$ is the set of instances on which PerTuple was faster than AllSol by more than 256 milliseconds; $\mathcal{A}$ is the set on which AllSol runs faster than PerTuple by more than 256 milliseconds.

The left-hand side of Table 4.2 reports the number of instances solved from each set ($\mathcal{A}$ and $\mathcal{P}$) by each algorithm (AllSol and PerTuple). The right-hand side of the table reports the corresponding average CPU times in seconds. To compute the average, we consider only the instances solved by *both* algorithms (i.e., 5,776 instances from $\mathcal{A}$ and 10,095 instances from $\mathcal{P}$). On the instances solved by both algorithms,

Table 4.2: Number of instances solved and the corresponding average times.

#Instances in	**solved by**...			**Average CPU** (sec)	
	AllSol	PerTuple	Both	AllSol	PerTuple
$\mathcal{A}$	5,777	5,776	5,776	**1.28**	4.97
$\mathcal{P}$	10,095	15,457	10,095	109.61	**5.21**

AllSol is 74.25% faster than PerTuple on the instances in $\mathcal{A}$, while PerTuple is 95.24% faster than AllSol on the instances in $\mathcal{P}$. Incidentally, the average time for PerTuple is small (5.21 seconds) on the particular subset of instances in $\mathcal{P}$ that were solved by AllSol (10,095 instances). The average time of PerTuple on all the instances of $\mathcal{P}$ (15,457 instances) is in fact much larger (17.55 seconds). Table 4.2 shows that AllSol and PerTuple clearly outperform each other in their

respective 'niche' (here, the instance sets $\mathcal{A}$ and $\mathcal{P}$ respectively). In practice, we need to determine from the outset which algorithm to use, which motivates us to build a classifier that uses machine learning techniques.

4.4.2 Parameters and features

The topology of the constraint network (e.g., degree of a variable) and the definitions of the constraints (e.g., tightness of a relation) heavily impact the performance of the algorithms for solving CSPs (PerTuple) and counting their solutions (AllSol). We suspect that the relative performance of AllSol and PerTuple is also affected by the density of solutions in the space. Thus, we considered the following CSP parameters. Below, π, σ and $\bowtie$ denote the relational operators project, select and natural join, respectively.

1. κ is a known parameter to predict if an instance is at the phase transition [Gent *et al.*, 1996]. It is defined for CSPs as $\kappa = -\frac{\sum_{R \in \mathcal{C}} log_2(1-p_R)}{\sum_{x \in \mathcal{X}} log_2(domain(x))}$, where p_R is the tightness of the constraint.

2. `relLinkage` is an approximate measure of how likely a 'tuple at the overlap of two relations' is to appear in a solution. We propose to compute it as follows: for every two relations R_i, R_j, let $V_{ij} = scope(R_i) \cap scope(R_j)$. $\forall t \in \pi_{V_{ij}}(R_i \bowtie R_j)$:

$$\texttt{relLinkage}(t) = \prod_{\forall R_k \; scope(R_k) \supseteq V_{ij}, R_k} \frac{|\sigma_t(R_k)|}{\prod_{x \in scope(R_k) \setminus V_{ij}} |domain(x)|}.$$

3. `tupPerVvp` is the sum of all tuples in which a given variable-value pair vvp appears, $\sum_{R_i \in \mathcal{R}} |\sigma_{vvp}(R_i)|$.

4. `tupPerVvpNorm` is the value of `tupPerVvp` normalized to the size of each relation, $\sum_{R_i \in \mathcal{R}} \frac{|\sigma_{vvp}(R_i)|}{|R_i|}$.

5. `tupPerVvpNormProd` is similar to `tupPerVvpNorm` using the product instead of the sum, $\prod_{R_i \in \mathcal{R}} \frac{|\sigma_{vvp}(R_i)|}{|R_i|}$.

6. `relPerVar` is the number of relations per variable v, $|\{R_i \mid v \in scope(R_i)\}|$, which is the degree of v in the primal graph.

For a given CSP instance, each parameter yields a set of numbers, which we combine into a single value using different statistical aggregations (e.g., average and standard deviation) to obtain the following 12 features for training our classifiers:

1. κ
2. $\log_2(avg(\texttt{relLinkage}))$
3. $\log_2(stDev(\texttt{relLinkage}))$
4. $\frac{stDev(\texttt{relLinkage})}{avg(\texttt{relLinkage})}$
5. $\frac{stDev(\texttt{tupPerVvp})}{avg(\texttt{tupPerVvp})}$
6. $avg(\texttt{tupPerVvpNorm})$
7. $stDev(\texttt{tupPerVvpNorm})$
8. $stDev(\texttt{tupPerVvpNormProd})$
9. $\frac{stDev(\texttt{tupPerVvpNormProd})}{avg(\texttt{tupPerVvpNormProd})}$
10. $avg(\texttt{relPerVar})$
11. $stDev(\texttt{relPerVar})$
12. $\frac{stDev(\texttt{relPerVar})}{avg(\texttt{relPerVar})}$

We originally considered 34 combinations of CSP parameters (e.g., product of domain sizes, relations sizes, the entropy of constraint definitions) and ways to aggregate the corresponding values (e.g., sums and products, their ratios and logarithms, averages, and standard deviations). After constructing different decision trees produced by the used learning algorithms (i.e., C4.5 and Random Forest, see Section 4.4.3), the above-listed 12 features appeared constantly at the top levels of the produced trees. It is commonly acknowledged by the machine learning community that the features appearing at the top levels of decision trees are likely to be the most significant ones. Thus, we settled with this set of 12 features.

4.4.3 Building the classifiers

To build the classifier, we used 'off-the-shelf' learning algorithms, the sample instances described in Section 4.4.1, the values of the set of features listed in Section 4.4.2 on the sample data, and the CPU times for solving the sample instances with both algorithms (PerTuple and AllSol).

- *Learning algorithms.* We experimented with ten different learning algorithms from the open-source data-mining tool Weka [Hall *et al.*, 2009]. The two algorithms that yielded the best results were J48 and RF, which are Java implementations of C4.5 [Quinlan, 1993] and Random Forests [Breiman, 2001], respectively. In our experiments, we used the default parameters for each algorithm (e.g., ten trees for RF). The advantage of C4.5 is that it outputs a single decision tree which, when limited to around 20 nodes, seemed to provide a good trade-off between classification precision and 'transparency' to a human user. We tuned the C4.5 algorithm to output heavily pruned trees by reducing the pruning confidence to one percent.

- *The feature sets.* We evaluated two feature sets: the set of 12 features listed in Section 4.4.2, and a subset of it consisting of the features #1, #4, #5 and #8. The four features of the latter consistently appeared at the top three levels of the decision trees that were constructed on ten different partitions of the training set. Thus, they are likely the most significant features.

- *Classes.* We classified the data into two classes: the first class is for the instances on which PerTuple is faster than AllSol by more than 256 milliseconds, and the second class is for the instances on which AllSol is faster than PerTuple by more than 256 milliseconds.

- *Training data ($\mathcal{T}$).* At the training stage, we used data from the partitions $\mathcal{A}$ and $\mathcal{P}$. We generated the training data, denoted by $\mathcal{T}$, by sampling, for every benchmark, a maximum of 30 instances from $\mathcal{A}$ and 30 instances from $\mathcal{P}$. To select the 30, we chose the 15 instances with the largest time difference between AllSol and PerTuple, randomly selecting the rest from the remaining instances in the benchmark. We sampled instances from $\mathcal{A}$ and $\mathcal{P}$ instead of including all of them in order to balance the number of instances in each class. We balanced the number of instances so that the classifier did not bias one class over the other in its attempt to reduce the overall error rate.

We evaluated various configurations of the learning algorithms according to the transparency of the classification process and the error rate. We consider the six configurations listed in Table 4.3.

Table 4.3: Main learning algorithms and configurations tested.

Classifier	**Learning Algorithm**	**#Trees**	**Avg. #Nodes**	**Setting**	**#Features**	**Avg. Error Rate**
DT1	**C4.5**	1	18.27	**Heavy pruning**	**12**	0.21
DT2	C4.5	1	37.73	Default pruning	12	0.20
RF1	**Random Forests**	10	181.22	**Default**	**12**	0.19
DT3	C4.5	1	10.60	Heavy pruning	4	0.23
DT4	C4.5	1	23.55	Default pruning	4	0.23
RF2	Random Forests	10	207.87	Default	4	0.19

We partitioned the training set $\mathcal{T}$ described above into ten partitions, and cross-validated each configuration by testing each partition on a classifier trained on the other nine partitions. Only the decision trees produced by C4.5 with heavy pruning were deemed to be transparent enough for readability. The number of nodes and the error rates reported in Table 4.3 are the averages across all ten folds of the cross-validation.

As for the classification error-rate, we performed a paired t-test and found no statistically significant difference between the classifiers produced by C4.5, under default pruning, using the set of 12 features and the set of four features (DT2 versus DT4). Also, we observed no statistically significant difference between the classifiers produced by C4.5 using the 12 feature set with pruning and the default pruning (DT1 and DT2).

However, we discovered that changing from the 12-feature set to four-feature set increases the classification error, on the heavily pruned trees, with more than 95% confidence. Moreover, we did not find any statistically significant difference between Random Forests and C4.5 for the 12 feature set. Therefore, we chose to use the set with 12 features for the remainder of the analysis, and for generating the production classifier since it is both human readable and performs as well or better than the others classifiers.

4.4.4 Empirical evaluations

We propose two hybrid solvers: $\textsc{Solver}_{C4.5}$ and $\textsc{Solver}_{RF}$ based on each of the two classifiers DT1 and RF1 of Table 4.3.

As described above, at the training stage, in order to avoid biasing the classifier while exploiting all the data available, we partitioned $\mathcal{T}$ into ten partitions, and performed a cross-validation by testing each partition using the classifier trained on the other nine partitions. Subsequently, in an experiment separate from the cross-validation, we trained a 'production classifier' on all the instances in $\mathcal{T}$, and used it to evaluate the instances in $\mathcal{P}$ that were not included in $\mathcal{T}$. Therefore, all the instances in $\mathcal{A}$ and $\mathcal{P}$ are validated with unbiased classifiers. The decision tree of the production classifier output by C4.5 is given in Figure 4.3 and uses the following features:

1 κ

2 $\log_2(avg(\texttt{relLinkage}))$

3 $\log_2(stDev(\texttt{relLinkage}))$

7 $stDev(\texttt{tupPerVvpNorm})$

10 $avg(\texttt{relPerVar})$

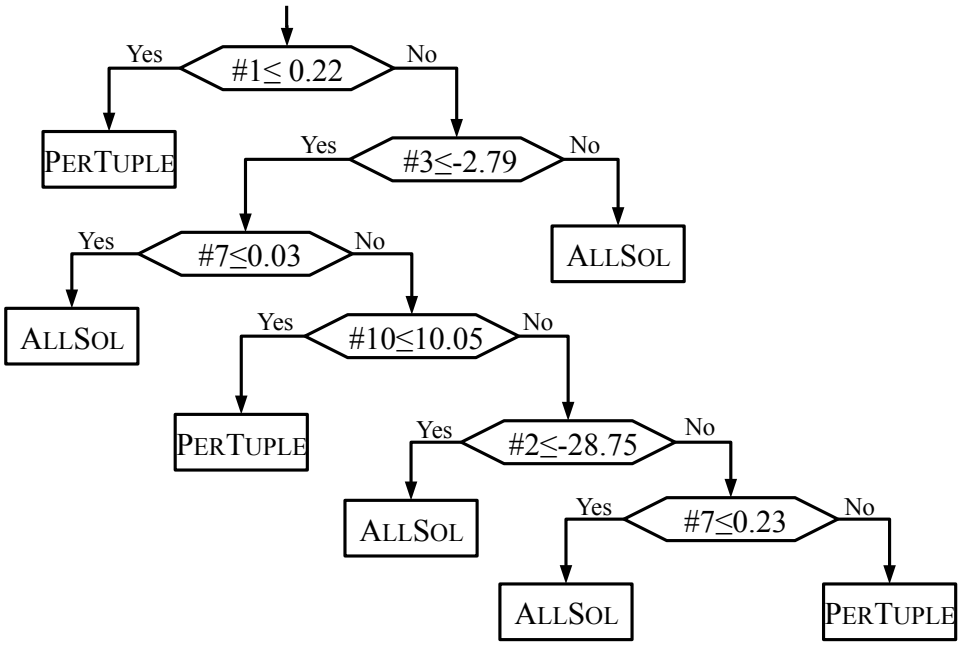

Figure 4.3: Decision tree of SOLVER$_{C4.5}$.

Table 4.4 lists, to the left, the number of instances solved by each algorithm (ALLSOL, PERTUPLE, SOLVER$_{C4.5}$ and SOLVER$_{RF}$), as well as the number of instances solved by all four algorithms. To the right, it lists the average CPU-time for executing each algorithm on those instances solved by all algorithms (and whose number is given in the center of the table). On the instances solved by all algorithms:

- In the ideal case, the (non-existent) perfect solver would choose to execute ALLSOL on 5,5776 instances averaging 1.27 seconds and PERTUPLE on 10,095 instances averaging 5.21 seconds.
- On the instances in partition $\mathcal{A}$, SOLVER$_{C4.5}$ loses 0.87 seconds by not making the ideal decision (ALLSOL), but saves 2.83 seconds by avoiding making the

Table 4.4: Comparing the performance of all four algorithms.

		#Instances solved by					Average CPU (sec)				
	Partition	AllSol	PerTuple	Solver$_{C4.5}$	Solver$_{RF}$	All solvers	#Instances	AllSol	PerTuple	Solver$_{C4.5}$	Solver$_{RF}$
1	$\mathcal{A}$	5,777	5,776	5,777	5,777	5,776	5,776	(ideal) 1.27	4.97	2.14	2.27
2	$\mathcal{P}$	10,095	15,457	15,439	14,012	10,095	10,095	109.61	(ideal) 5.21	7.72	31.53
3	$\mathcal{A} \cup \mathcal{P}$	15,872	21,233	21,216	19,789	15,871	15,871	70.18	5.12	5.69	20.88

wrong choice (PerTuple).

- On the instances in partition $\mathcal{P}$, Solver$_{C4.5}$ loses 2.51 seconds by not making the ideal decision (PerTuple), but saves 101.89 seconds by avoiding making the wrong choice (AllSol).

- On all common instances ($\mathcal{A} \cup \mathcal{P}$), Solver$_{C4.5}$ loses 2.51 seconds by not making the ideal decision (PerTuple), but saves 101.89 seconds by avoiding making the wrong choice (AllSol).

- Solver$_{C4.5}$ consistently outperforms Solver$_{RF}$.

- PerTuple remains the overall winner and the safest bet. For this reason, future chapters use PerTuple.

One might worry that the benchmark data used to build and validate our classifiers are not general enough. One may rightfully worry that the features we selected, which attempt to capture the characteristics of the structure of a CSP, and our classifiers trained on structured data, may lose their 'edge' when used on 'amorphous' instances such as randomly generated CSPs. For this reason, we tested our two hybrid solvers Solver$_{4.5}$ and Solver$_{RF}$ on three sets of random CSPs (model B) generated in a window around the phase transition. The hybrid solvers Solver$_{4.5}$ and Solver$_{RF}$ use

the production classifiers trained on the benchmark data in set $\mathcal{T}$, i.e., they were not trained on any instance from the three sets of random CSPs. This experiments also attempts to test how well the classifiers and solvers generalize to CSPs instances on which they were not trained. The problem sets' characteristics and the average times on the instances solved by both AllSol and PerTuple are shown in Table 4.5.

Table 4.5: Randomly generated CSPs.

	Set I	**Set II**	**Set III**
Number of variables	10	30	75
Domain size	10	6	5
Number of relations	100	75	120
Constraint arity	3		
Number of tuples per relation	[100,900]	[22,194]	[12,112]
Total number of instances	1000		
Number of instances solved by AllSol	1,000	731	369
Number of instances solved by PerTuple	1,000	983	397
Number of instances solved by both	1,000	731	340
On the instances solved by both PerTuple and AllSol			
Average time of AllSol in sec.	154.61	89.91	221.87
Average time of PerTuple in sec.	128.32	174.23	412.13
Average time of Solver$_{RF}$ in sec.	154.62	134.94	251.39
Average time of Solver$_{C4.5}$ in sec.	150.98	174.23	412.12

In Table 4.6, we compare the performance of our 'production' solvers (Solver$_{RF}$ and Solver$_{C4.5}$) on the benchmark data and the randomly generated. Below, we discuss the content of the table and summarize our conclusions:

- *Fatal* indicates the number of 'fatal' decisions corresponding to choosing the wrong solver (AllSol or PerTuple), that is, choosing a solver that does not complete within the time threshold over another that does.

- *Saved* indicates the number of correct decisions corresponding to choosing a solver that completes within the time threshold over another that does not. The number of instances 'saved' justifies the efforts of this research. While the

Table 4.6: Comparing the two new hybrid solvers.

	#Instances		Average savings (sec)	Classification error
	Fatal	**Saved**		
Benchmarks (21,234 instances)				
SOLVER$_{RF}$	1,445	3,918	33.54	**0.22**
SOLVER$_{C4.5}$	**18**	**5,345**	**63.92**	0.31
Set I				
SOLVER$_{RF}$	0	0	-26.29	**0.40**
SOLVER$_{C4.5}$	0	0	-19.01	0.42
Set II				
SOLVER$_{RF}$	108	144	-5.74	**0.40**
SOLVER$_{C4.5}$	**0**	**252**	-84.32	0.55
Set III				
SOLVER$_{RF}$	53	33	**132.22**	**0.17**
SOLVER$_{C4.5}$	**29**	**57**	-190.25	0.26

large number of 'saved' instances in the benchmark data can be justified by the structure of the CSPs and the fact that the classifiers were trained on similar data, *the large numbers of 'saved' data on Sets II and III justify our endeavor by demonstrating how well our system generalizes to new types of CSPs.*

- *Average savings* indicates how much time on average is saved per instance by the hybrid solver on the instances solved by both ALLSOL and PERTUPLE. The hybrid solvers yielded positive savings in some of the cases.

- *Classification error* indicates the rate of bad choices made by each hybrid solver. A bad choice occurs either when the chosen solver does not solve an instance but the alternative solver does, or when the chosen solver is slower than the alternative solver. The highest error rate is in Set II, which resulted in 84.32 seconds time loss on average.

4.4.5 Conclusions

We empirically evaluated PerTuple, AllSol and the hybrid solvers on benchmark and random problems. On most benchmark instances, PerTuple outperformed AllSoll. In the ideal situation, we expected the hybrid solvers to run the faster algorithm for a given instance. All of our classifiers achieved an error less than 23% on average. As a result, the hybrid solvers Solver$_{RF}$ and Solver$_{C4.5}$ were able correctly choose between PerTuple and AllSoll to yield savings both in the number of solved instances and in the average CPU time.

The classifiers in the hybrid solvers were not trained on the randomly generated problems. Nevertheless, they yielded savings in number of solved instances and average time. In Set III, only Solver$_{C4.5}$ achieved savings in terms of the number of solved instances, and thus outperformed Solver$_{RF}$.

Our preliminary investigations confirmed that more sophisticated techniques for building the classifier are worth investigating [Geschwender *et al.*, 2013]. However, that research effort is beyond the scope of this thesis.

Again, in the current state of affairs, PerTuple remains the overall winner and the safest bet and is used in the remaining chapters of this dissertation.

Summary

In this chapter, we presented AllSol, an alternative algorithm for enforcing R($*$,m)C. We also identified various CSP parameters and used them to build a hybrid solver that chooses either PerTuple or AllSol given a problem instance. We evaluated our solvers on benchmark and randomly generated problems for computing the minimal network of each problem.

Chapter 5

Localized Consistency & Structure-Guided Propagation

In this chapter, we investigate ways to exploit the structure of a tree decomposition of the constraint network of a CSP in the context of higher level consistencies. In particular, we propose to

1. Restrict the application of R($*$,m)C to the clusters of a tree decomposition, and
2. Guide constraint propagation along the structure of the tree decomposition.

After quickly reviewing how we generate a tree decomposition of a CSP, we discuss localization of R($*$,m)C to the clusters of a tree decomposition and theoretically characterize the resulting consistency properties in terms of the previous ones discussed in this thesis. Then we discuss structure-based constraint propagation and propose three strategies for managing the propagation queue of the localized consistency. (Our strategies are not restricted to localized consistency but applicable to any constraint propagation algorithm, yielding qualitatively similar results.) Finally, we conduct

extensive empirical evaluations to demonstrate the effectiveness of our approach. The contributions of this chapter are as follows:

1. Introduction of a cluster-based relational consistency property cl-R($*$,m)C.
2. Proposal of three queue-management strategies and the algorithms for implementing them on a generic local consistency property.
3. Empirical evaluation of the above two contributions on benchmark problems.

Results from this chapter have been published [Karakashian *et al.*, 2013].

5.1 Generating a Tree Decomposition

Many techniques for generating a tree decomposition of a CSP exist [Dechter and Pearl, 1989; Jeavons *et al.*, 1994; Gottlob *et al.*, 1999]. We use an adaption for non-binary CSPs of the tree-clustering technique [Dechter and Pearl, 1989]:

1. We triangulate the primal graph of the CSP using the min-fill heuristic [Kjærulff, 1990]. Figure 5.1 shows a sample CSP and Figure 5.2 shows a triangulated primal graph of the example in Figure 5.1. The dotted edges (B,H) and (A,I) in Figure 5.2 are fill-in edges generated by the triangulation algorithm. The ten maximal cliques of the triangulated graph are highlighted with 'blobs.'
2. We identify the maximal cliques in the resulting chordal graph using the MaxCliques algorithm [Golumbic, 1980], and use the identified maximal cliques to form the clusters of the tree decomposition as shown in Figure 5.3 for the example in Figure 5.1 .
3. We build the tree by connecting the clusters using the JoinTree algorithm [Dechter, 2003]. While any cluster can be chosen as the root of the tree, we choose the

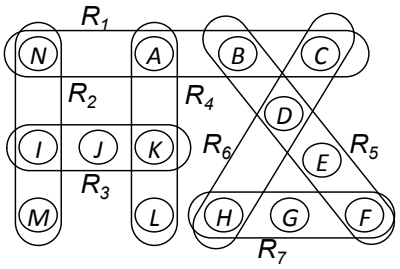

Figure 5.1: The hypergraph of a CSP.

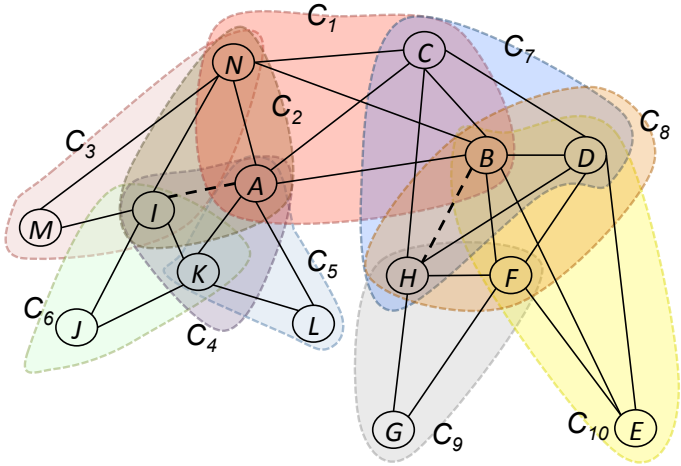

Figure 5.2: Triangulated primal graph of the example in Figure 5.1 and the corresponding maximal cliques.

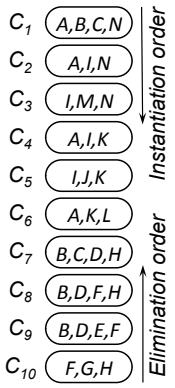

Figure 5.3: Maximal cliques.

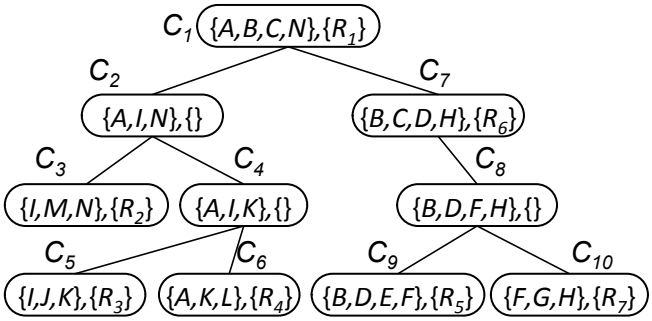

Figure 5.4: Tree decomposition.

cluster that minimizes the longest chain from the root to a leaf. Figure 5.4 shows the tree after connecting the maximal cliques of Figure 5.3. It illustrates also the so-called *elimination ordering* (i.e., bottom up) and *instantiation ordering* (i.e., top down) as used in this thesis.

4. We determine the variables and constraints of each cluster as follows: *a*) the variables of a cluster cl, $\chi(cl)$, are the variables in the maximal clique that yields the cluster; *b*) the clique in the primal graph formed by the vertices of

a given constraint R of the CSP is, by construction, a subset of at least one maximal clique in the triangulated primal graph. Thus, *scope*(R) is a subset of the variables of at least one cluster; and c) The constraints of a cluster cl, $\psi(cl)$, are the constraints R_i, such that *scope*(R_i) $\subseteq \chi(cl)$.

Figure 5.4 shows a tree decomposition produced by this process for the example of Figure 5.1. Note that some clusters may end up with no constraints (e.g., C_2, C_4 and C_8), and are ignored during processing.

A *separator* of two adjacent clusters is the set of variables that are associated with both clusters. A given tree decomposition is characterized by its *treewidth*, which is the maximum number of variables in a cluster.

5.2 Localizing Consistency to Clusters

We denote by cl-R($*$,m)C the consistency property corresponding localizing R($*$,m)C to the clusters of a tree decomposition. Here, we discuss the benefits of cl-R($*$,m)C, explain its design, and compare it to the properties discussed in Chapter 3.

When we introduced the relational consistency property R($*$,m)C in Chapter 3, we did not discuss the choice of the value of m. Obviously, increasing the value of m increases the level of consistency enforced. However, the number of combinations of relations to consider increases exponentially with m: It is $\mathcal{O}(\binom{e}{m}) = \mathcal{O}(e^m)$ where e is the number of constraints in the problem. By localizing R($*$,m)C to the clusters of a tree decomposition, we reduce e to the number of constraints in a cluster cl (i.e., $|\psi(cl)|$), and, consequently, decrease the total number of combinations that we have to store and handle for a given problem instance. In the extreme case, when $m = |\psi(cl)|$, three goals are achieved:

1. We have to handle only *one* combination of constraints per cluster.

2. The value of m is directly determined by each cluster and 'adaptively' varies along the tree decomposition. And,

3. The approach constitutes an appealing approximation of the famous tractability condition of CSPs that relates the *level of consistency* of a CSP to a structural parameter of its constraint network, such as the *treewidth* [Freuder, 1982; Dechter, 2003].

Next, we describe how we transfer information between clusters.

5.2.1 Information transfer between clusters

The information transferred from a cluster to its parent (or its child) transits via the constraints common to the two adjacent clusters and the domains of the variables in the separator. Any constraint whose scope is a subset of the variables in two clusters is added to both of them (e.g., relation R_4 in Figure 5.5). Thus, when it is

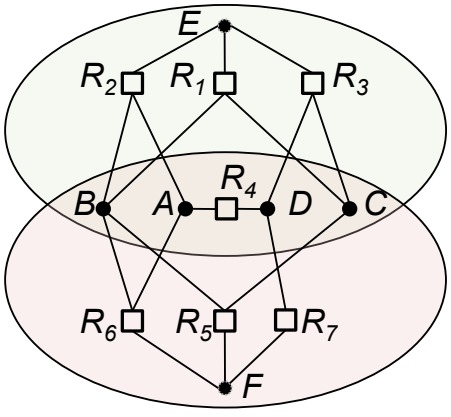

Figure 5.5: Two adjacent clusters with $\{A, B, C, D\}$ and R_4 in the separator.

filtered in one cluster, the information is automatically passed to the other cluster. Information is also transferred between clusters through the domains of the variables.

The constraints in a cluster are projected onto the domains of the variables in the separator, and constraints in the neighboring cluster are filtered with the domains of the variables in the separator. In Chapter 6, we investigate more aggressive strategies for improving information transfer between clusters.

5.2.2 Characterizing cl-R(∗,m)C

Localization prevents us from considering combinations of constraints across clusters. As a result, the consistency enforced is weakened. Below, we characterize cl-R(∗,m)C in terms of the consistency properties discussed in previous chapters.

Theorem 9 *R(∗,m)C is strictly stronger than cl-R(∗,m)C.*

Proof: See Appendix C.4.

Theorem 10 *cl-R(∗,m)C and maxRPWC are not comparable for* $m \geq 2$.

Proof: See Appendix C.4.

Theorem 11 $\forall a, b \in \mathbb{N}$ *where* $a < b \leq |\psi(cl)|$, *cl-R(∗,a)C is strictly weaker than cl-R(∗,b)C.*

Proof: Straightforward. □

Figure 5.6 summarizes the above results and integrates them with those of Figure 3.3 in Chapter 3 (shown in grey for differentiation):

5.3 Structure-Guided Propagation

Algorithms for enforcing consistency typically maintain a queue of the variables (or constraints) that need to be revised for propagation. However, the ordering of those

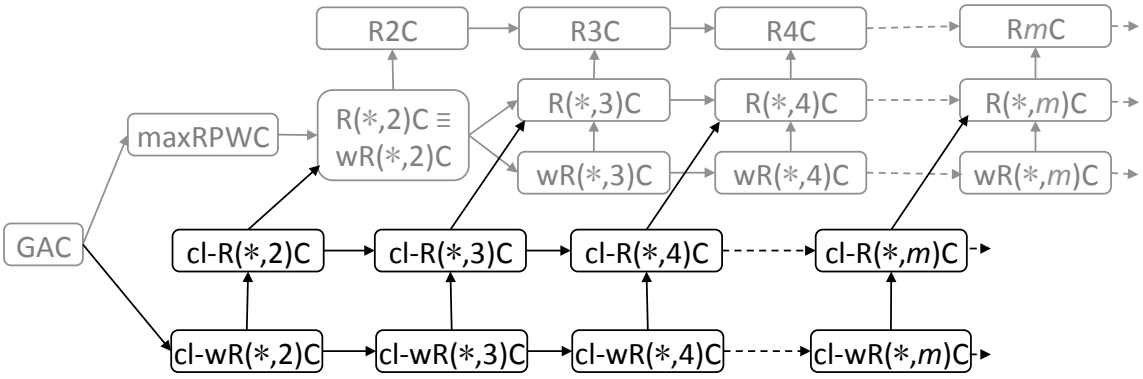

Figure 5.6: Characterizing cl-R(∗,m)C in terms of GAC, maxRPWC, and R(∗,m)C.

elements in the queue is usually random. Information specific to a problem, such as its structure, is typically neglected. One avenue to improve the performance of a propagation algorithm is to 'direct' or 'guide' propagation along a 'structural ordering' of the constraint network. The only technique that follows this rationale of which we are aware is *directional consistency* (e.g., directional i-consistency and adaptive consistency) [Dechter and Pearl, 1987], which may require adding new constraints to the problem and may thus increase the time and space cost. In this section, we exploit the problem's structure

1. Without weakening the consistency level enforced, and

2. Without adding new constraints to the problem.

We propose three ordering strategies (i.e., Static, Priority, and Dynamic) that follow the structure of a tree decomposition of the CSP. We show that structure-based orderings exhibit qualitatively equivalent performances among themselves, but clearly outperform the random ordering (Random, which ignores the structure of the constraint network but otherwise results in the same consistency level).

Although our approach is applicable to any constraint propagation algorithm, consistency algorithms that are readily restricted to the clusters of a tree decomposition

(such as the ones discussed in Section 5.2) are particularly well suited to exploiting the tree structure.

5.3.1 Related Work

Wallace and Freuder investigated various ordering heuristics for the propagation queue of arc consistency showing a reduction of the number of constraint checks [1992]. Unlike the strategies studied in this paper, which exploit the structure of the network, their heuristics considered the properties of individual domains and constraints. Laburhe [2000] and Schulte and Stuckey [2004] ordered propagation queues by prioritizing the constraints based on the time complexity of their processing. Thus, the queue ordering is based on the cost and a predefined set of rules, and not on the structure of the problem as proposed in this paper. Schulte and Stuckey [2008] used the semantics of the constraints/propagators in re-ordering the queue, but did not exploit the structure of the problem. Lagerkvist and Schulte [2009] also studied the propagation order of constraints, but required the user to specify the ordering. Francis and Stuckey [2007] investigated a propagation ordering on problems with articulation points, which is less general than a tree decomposition.

Freuder linked the width of the constraint network to the consistency level necessary in a relation that guarantees a backtrack-free search [1982]. This approach was extended by Dechter and Pearl [1987] to adaptive consistency where propagation proceeds along a fixed variable ordering while generating new constraints. This work is perhaps the closest in spirit to the one presented here. However, the approaches differ in that adaptive consistency may require adding new constraints to the problem, which can be prohibitive in terms of both time and space. Planken *et al.* [2008] used DPC on a perfect elimination ordering of some triangulation of the constraint graph of

a binary CSP in order to propagate Partial Path Consistency (PPC) [Bliek and Sam-Haroud, 1999], which requires adding constraints, thus modifying the constraint graph. Their work is restricted to the Simple Temporal problem [Dechter *et al.*, 1991]. The work presented in this paper exploits the information gained from triangulation/tree decomposition but does not alter the topology of the constraint network or add any constraints to the problem.

More recently, Jégou and Terrioux [2010] proposed the consistency property w-SC, which enforces inverse consistency (by domain filtering) of a relaxed CSP obtained by removing constraints in order to guarantee a tree decomposition of bounded width w. The structure of the tree decomposition is used to relax the problem and not to guide the propagation.

5.3.2 Structure of the propagation queue

Although we restrict our discussion to the localized consistency properties introduced in Section 5.2, we claim that the queue-management strategies proposed here are *generic and applicable to any consistency algorithm* by a simple adaptation of the propagation queue of the consistency algorithm.

In order to introduce the structure provided by the tree decomposition $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ into the operation of a consistency algorithm, we use the propagation queue of the consistency algorithm. Let $\mathcal{Q}_{cl}$ be the queue of the algorithm for enforcing cl-R($*$,m)C. $\mathcal{Q}_{cl}$ is a set of sets of relation-combination pairs as given in Expression (5.1), where $\Phi(cl_i)$ is the set of all m combinations of connected constraints in the dual graph induced by the constraints in $\psi(cl_i)$:

$$\mathcal{Q}_{cl} = \{\{\langle \varphi, R\rangle \mid R \in \psi(cl_i) \wedge \varphi \in \Phi(cl_i)\} \mid cl_i \in \mathcal{V}\} \tag{5.1}$$

Note that the consistency algorithms studied here implement a support-type data structure for tuples; thus, revisions are executed only for tuples that lost a support [Bessiere *et al.*, 2005]. All queue-management strategies reach the same fixed point, but differ in the order in which the clusters are visited and in the number of times that consistency is enforced on them.

Further, the order of the relations in each element of $\mathcal{Q}_{cl}$ is arbitrary. Ordering the relations by increasing size yielded no difference in the performance, likely because support structures were implemented for tuples.

5.3.3 Queue-management strategies

We propose the following queue-management strategies for cl-wR($*$,m)C to order the clusters of a tree decomposition, and revise the relations associated to the clusters in that order.

1. RANDOM: The elements of $\mathcal{Q}_{cl}$ are processed in first in first out order, without following the tree structure of any specific criterion. The ordering is thus arbitrary.

2. STATIC: The order of the elements of $\mathcal{Q}_{cl}$ corresponds to the ordering given by the MAXCLIQUES algorithm [Golumbic, 1980]. The clusters are processed back and forth in that order until quiescence. At preprocessing, we start from bottom up, following the chain of maximal cliques in the direction of the perfect elimination ordering (PEO). During search, we start at the shallowest cluster in the MAXCLIQUES ordering where a relation on the instantiated variable appears and proceed down in the opposite direction to the of the PEO. We repeat until quiescence.

3. PRIORITY: The propagation algorithm sweeps through the tree, starting from the leaves, and visiting each cluster once. Then it sweeps back starting from one of the clusters towards the leaves. We keep a fringe of 'open clusters,' which constitutes a frontier of last visited clusters. We select from the fringe the cluster that has witnessed the most significant filtering as determined by a heuristic. Importantly, each cluster is processed exactly once at each sweep or *iteration*.

4. DYNAMIC: The clusters are traversed in a similar way to the PRIORITY ordering except that the clusters can be processed more than once during an iteration depending on the amount of 'propagation activity' (i.e., filtering) witnessed by the relations in the cluster. Clusters that witness significant filtering activities may be processed more than once during an iteration.

When the tree decomposition is a Berge-acyclic graph [Fagin, 1983], the STATIC and PRIORITY strategies require only two passes before reaching quiescence.

The algorithms that implement the last two strategies (i.e., DYNAMIC and PRIORITY) are described in detail in Sections 5.3.4, 5.3.4.3, and 5.3.4.4. The motivation for these two strategies is to discover inconsistency as quickly as possible by 'tracing' some noteworthy activity of constraint propagation. Both of these strategies were expected to be particularly useful on unsolvable instances. However, the experimental results showed all three structure-based strategies to be equivalent (see Section 5.4).

5.3.4 Implementing PRIORITY and DYNAMIC

The algorithms implementing PRIORITY and DYNAMIC proceed by iteration, where each iteration consists of one sweep through the tree decomposition. During an iteration, a given consistency property is enforced on each cluster individually. The application of a given consistency algorithm to a given cluster *cl* is accomplished

by a call to EnforceConsistency(cl). (This call is made in Algorithms 4 and 5 for Priority, and in Algorithms 6 and 7 for Dynamic.) We use a data structure called *fringe* where clusters are stored immediately after being processed; that is, after calling EnforceConsistency on them. While the two strategies differ in which clusters they add to the fringe, they use the same criterion to select from the fringe the cluster whose neighbors are candidates for processing. The rationale is to choose the cluster in the fringe that is likely to trigger the most propagation activity.

Below, we discuss the data structures used in our algorithms, the criterion for choosing the cluster from the fringe, the pseudocode of Priority and Dynamic, and finally we provide the algorithms for managing the propagation queue at preprocessing (Algorithm 4 for Priority and Algorithm 6 for Dynamic) and during search (Algorithm 5 for Priority; and Algorithm 7 for Dynamic).

5.3.4.1 Functions & accessors used in pseudocode

Below, we introduce the notations used in the pseudo-code. Typically, the names of variables and attributes are italicized, and small caps are used for the names of functions and methods.

- EnforceConsistency(cl) calls a local consistency algorithm and locally enforces on the cluster cl. It returns a tuple ($consistent, change$), where *consistent* is a Boolean indicating whether or not the cluster is found to be consistent, and *change* is the list of relations that lost tuples as a result of enforcing consistency.
- *Iteration*(cl) is an integer counter, an attribute of a cluster cl.
- *Neighbors*(cl) is the list of clusters adjacent to cl in the tree decomposition.

- *Degree*(cl) is degree of cl in the decomposition tree, which is the cardinality of *Neighbors*(cl).
- *Clusters*(R) is the set of clusters where a relation R appears (i.e., $\{cl|R \in \psi(cl)\}$).
- *Revisit*(cl) is a Boolean flag on a cluster indicating that at least one tuple in some relation in the cluster was deleted as a result of processing another cluster where the relation also appears.
- NbrCurrTuples(cl) is a function that computes the number of tuples alive in the cluster cl.
- $NbrInitTuples(cl)$ is the number of tuples initially in the cluster cl, obtained as the sum of tuples in the relations of the cluster.
- $NbrDelTuples1(cl)$, $NbrDelTuples2(cl)$ are two attributes of the cluster cl storing the number of tuples deleted from the relations at some point in time.

5.3.4.2 Selection from *fringe*

Cluster selection is accomplished using the function RemoveMax in Algorithm 3.

Each tuple in a relation is marked as either alive or deleted. The two attributes $NbrDelTuples1(cl)$ and $NbrDelTuples2(cl)$ store the total number of deleted tuples from the relations in cl at two different points in time. $NbrDelTuples1(cl)$ stores the number of deleted tuples right before the last time consistency was enforced on the cluster (and the cluster was then added to the fringe). $NbrDelTuples2(cl)$ stores the number of tuples deleted from cl since the beginning of *any* processing, which includes the number of tuples deleted after the cluster was added to the fringe. The number of tuples lost since the last time consistency was enforced on the cluster is obtained by making the difference between $NbrDelTuples2(cl)$ and $NbrDelTuples1(cl)$. This

Algorithm 3: RemoveMax($fringe$)

Input: $fringe$
Output: Cluster with the highest ratio of deleted tuples.

```
1  maxScore ← 0
2  maxCluster ← First(fringe)
3  foreach cl ∈ fringe do
4      nbrtuples ← NbrCurrTuples(cl)
5      NbrDelTuples2(cl) ← NbrInitTuples(cl) − nbrtuples
6      score ← (NbrDelTuples2(cl) − NbrDelTuples1(cl)) / nbrtuples
7      if score ≥ maxScore then
8          maxScore ← score
9          maxCluster ← cl
10 return maxCluster
```

number equals to the number of tuples deleted during the last processing of cl plus the number of tuples deleted after cl was processed, as a result of filtering relations in cl while processing other clusters where those relations also appear. The ratio of the number of those deleted tuples to the total number of tuples remaining in the cluster's relations is computed and used to assess the degree of activity in the cluster. The cluster with the largest ratio is assumed to be the one that witnessed the largest amount of 'activity.' Naturally, other criteria can be used in place of this scheme.

5.3.4.3 Algorithm for Priority

A given iteration sweeps through the clusters from the leaves up the branches of the tree (or in reverse order from a given cluster towards the leaves of the tree). When a cluster is selected and removed from the fringe, EnforceConsistency is called on each of its neighbors that have not been processed during the current iteration. Thus, at any given sweep, each cluster is processed exactly once. The algorithm halts when it detects no change throughout an entire iteration. There is a slight difference in how the algorithm is applied during pre-processing (Algorithm 4) and for full-lookahead

during search (Algorithm 5). The pre-processing stage is described first, followed by the full-lookahead stage.

Algorithm 4 proceeds by initializing the fringe with the clusters of degree one (leaf clusters) in the loop on Line 5. Each cluster is processed before it is added

Algorithm 4: PRIORITY-PREPROCESSING($Clusters$)

```
   Input: Clusters the list of clusters of a tree decomposition of the CSP
   Output: true if the problem is consistent, false otherwise
 1 foreach cl ∈ Clusters do
 2     Iteration(cl) ← 0, NbrDelTuples1(cl) ← 0, NbrDelTuples2(cl) ← 0
 3 iteration ← 1
 4 fringe ← ∅
 5 foreach cl ∈ Clusters and Degree(cl) = 1 do
 6     (consistent, ∅) ← EnforceConsistency(cl)
 7     if consistent = false then return consistent
 8     Iteration(cl) ← iteration
 9     fringe ← fringe ∪ {cl}
10 newChange ← true
11 while newChange do
12     newChange ← false
13     roots ← ∅
14     while fringe ≠ ∅ do
15         cl ← RemoveMax(fringe)
16         foreach cl_i ∈Neighbors(cl) and Iteration(cl_i) < iteration do
17             NbrDelTuples1(cl_i) ← NbrDelTuples2(cl_i)
18             (consistent, change) ← EnforceConsistency(cl_i)
19             if consistent = false then return consistent
20             newChange ← newChange or change
21             Iteration(cl_i) ← iteration
22             fringe ← fringe ∪ {cl_i}
23             if (∀cl_j ∈ Neighbors(cl_i) Iteration(cl_j) = iteration) then
24                 roots ← roots ∪ {cl_i}
25     cl ←RemoveMax(roots)
26     iteration ← iteration + 1
27     Iteration(cl) ← iteration
28     (consistent, change) ← EnforceConsistency(cl)
29     if consistent = false then return consistent
30     fringe ← {cl}
31 return true
```

to the fringe. Afterwards, in the loop on Line 14, the cluster *cl* with the highest level of 'activity' in the latest iteration is removed from the fringe. RemoveMax (Algorithm 3) selects and removes a cluster from the fringe. The clusters cl_i adjacent to *cl* in the tree that were not processed during the current iteration are processed and added to the fringe. We continue selecting and removing clusters from the fringe. The iteration ends when the fringe is empty. A set of clusters is designated as *pseudo-root* at the end of the iteration. These clusters are the ones that were processed after all their neighbors. The pseudo-root that has the highest 'activity' value is selected, processed, and added to the fringe if it is consistent on Line 28. The process loops until quiescence. Whenever a relation becomes empty, EnforceConsistency returns *false* and the algorithm ends, signaling inconsistency. When no relation needs processing, the algorithm ends returning *true*.

The lookahead algorithm (Algorithm 5) is similar to the one for pre-processing (Algorithm 4), but instead of initializing the fringe with the clusters of degree one, it initializes the fringe with the shallowest cluster containing the variable instantiated by search.

Algorithm 5: PRIORITY-SEARCH($cluster, Clusters$)

Input: $cluster$ a cluster that has the instantiated variable, and $Clusters$ the list of clusters of a tree decomposition of the CSP

Output: $true$ if the problem is consistent, $false$ otherwise

```
1  foreach cl ∈ Clusters do Iteration(cl) ← 0
2  iteration ← 1
3  (consistent, ∅) ← EnforceConsistency(cluster)
4  if consistent = false then return consistent
5  Iteration(cluster) ← iteration
6  fringe ← {cluster}
7  newChange ← true
8  while newChange do
9      newChange ← false
10     roots ← ∅
11     while fringe ≠ ∅ do
12         cl ← RemoveMax(fringe)
13         foreach cl_i ∈ Neighbors(cl) and Iteration(cl_i) < iteration do
14             NbrDelTuples1(cl_i) ← NbrDelTuples2(cl_i)
15             (consistent, change) ← EnforceConsistency(cl_i)
16             if consistent = false then return consistent
17             newChange ← newChange or change
18             Iteration(cl_i) ← iteration
19             fringe ← fringe ∪ {cl_i}
20             if (∀cl_j ∈ Neighbors(cl_i) Iteration(cl_j) = iteration) then
21                 roots ← roots ∪ {cl_i}
22     cl ←RemoveMax(roots)
23     iteration ← iteration + 1
24     Iteration(cl) ← iteration
25     (consistent, change) ← EnforceConsistency(cl)
26     if consistent = false then return consistent
27     fringe ← {cl}
28 return true
```

5.3.4.4 Algorithm for DYNAMIC

DYNAMIC differs from PRIORITY in that clusters already processed during the current iteration can be added to the fringe. Hence, a cluster may be processed multiple times during a given iteration. DYNAMIC raises the following challenges:

1. *Not terminating.* Whenever a cluster is removed from the fringe, all of its neighbors are candidates for processing. If all are processed, then all would have to be added to the fringe. In this situation, the fringe will never be empty, and the algorithm will not halt.

2. *Early termination.* Instead of processing all the neighbors and adding them back to the fringe, we process only those clusters that lost a tuple since the last processing and add them to the fringe. However, if, in some neighboring clusters, no tuples are deleted, then they will not be processed and added to the fringe. In this case, the algorithm may stop prematurely before checking all the clusters.

Both problems are solved by adding to the fringe the clusters that either lost tuples or were never processed during the current iteration.

Algorithms 6 and 7 outline the dynamic queue management for preprocessing and maintaining the property during search, respectively. They are the same as Algorithms 4 and 5, except for the condition for adding clusters back to the fringe. The condition checks, in addition to the iteration counter of the cluster, the *Revisit* flag on Line 18 of Algorithm 6. The *Revisit* flag for a cluster is set on Line 23 if a relation in the cluster loses a tuple during the recent processing of a cluster.

Algorithm 6: DYNAMIC-PREPROCESSING($Clusters$)

Input: $Clusters$ the list of clusters of a tree decomposition of the CSP
Output: $true$ if the problem is consistent, $false$ otherwise

```
1  foreach cl ∈ Clusters do
2      Iteration(cl) ← 0, NbrDelTuples1(cl) ← 0, NbrDelTuples2(cl) ← 0
3      Revisit(cl) ← true
4  iteration ← 1
5  fringe ← ∅
6  foreach cl ∈ Clusters and Degree(cl) = 1 do
7      (consistent, ∅) ← EnforceConsistency(cl)
8      if consistent = false then return consistent
9      Iteration(cl) ← iteration
10     fringe ← fringe ∪ {cl}
11 newChange ← true
12 while newChange do
13     newChange ← false
14     roots ← ∅
15     while fringe ≠ ∅ do
16         cl ← RemoveMax(fringe)
17         foreach cl_i ∈ Neighbors(cl) do
18             if Iteration(cl_i) < iteration or Revisit(cl_i) = true then
19                 NbrDelTuples1(cl_i) ← NbrDelTuples2(cl_i)
20                 (consistent, change) ← EnforceConsistency(cl_i)
21                 if consistent = false then return consistent
22                 newChange ← newChange or change
23                 foreach cl_j ∈ Clusters(R_i), R_i ∈ change do  Revisit(cl_j) = true
24                 Revisit(cl_i) = false
25                 Iteration(cl_i) ← iteration
26                 fringe ← fringe ∪ {cl_i}
27                 if (∀cl_j ∈ Neighbors(cl_i) Iteration(cl_j) = iteration) then
28                     roots ← roots ∪ {cl_i}
29     cl ←RemoveMax(roots)
30     iteration ← iteration + 1
31     Iteration(cl) ← iteration
32     (consistent, change) ← EnforceConsistency(cl)
33     if consistent = false then return consistent
34     fringe ← {cl}
35 return true
```

Algorithm 7: DYNAMIC-SEARCH($cluster, Clusters$)

Input: $cluster$ a cluster that has the instantiated variable and $Clusters$ the list of clusters of a tree decomposition of the CSP
Output: $true$ if the problem consistent, $false$ otherwise

```
1  foreach cl ∈ Clusters do Iteration(cl) ← 0, Revisit(cl) ← true
2  iteration ← 1
3  (consistent, ∅) ← EnforceConsistency(cluster)
4  if consistent = false then return consistent
5  Iteration(cluster) ← iteration
6  fringe ← {cluster}
7  newChange ← true
8  while newChange do
9      newChange ← false
10     roots ← ∅
11     while fringe ≠ ∅ do
12         cl ← RemoveMax(fringe)
13         foreach cl_i ∈ Neighbors(cl) do
14             if Iteration(cl_i) < iteration or Revisit(cl_i) = true then
15                 NbrDelTuples1(cl_i) ← NbrDelTuples2(cl_i)
16                 (consistent, change) ← EnforceConsistency(cl_i)
17                 if consistent = false then return consistent
18                 newChange ← newChange or change
19                 foreach cl_j ∈ Clusters(R_i), R_i ∈ change do Revisit(cl_j) ← true
20                 Revisit(cl_i) ← false
21                 Iteration(cl_i) ← iteration
22                 fringe ← fringe ∪ {cl_i}
23                 if (∀cl_j ∈ Neighbors(cl_i) Iteration(cl_j) = iteration) then
24                     roots ← roots ∪ {cl_i}
25     cl ← RemoveMax(roots)
26     iteration ← iteration + 1
27     Iteration(cl) ← iteration
28     (consistent, change) ← EnforceConsistency(cl)
29     if consistent = false then return consistent
30     fringe ← {cl}
31 return true
```

5.3.5 Correctness of the algorithms

The algorithms that we presented are guaranteed to stop either by discovering the inconsistency in the problem, or by enforcing the consistency property.

Algorithms 4 and 5 add a cluster to the fringe exactly once during an iteration. A new iteration is started only if a tuple is deleted in a relation. Given that there are a finite number of tuples, there will be a finite number of iterations. Thus, the algorithms must eventually halt. Moreover, the algorithms stop after all the clusters are processed and no tuple is deleted. Therefore, the consistency property is enforced by both algorithms.

Algorithms 6 and 7 are more complicated with respect to the fringe management schemes because the clusters may be added to the fringe multiple times during an iteration. However, a cluster is guaranteed to be added to the fringe once, and for each subsequent addition, it must have a relation that lost a tuple since it was last added to the fringe. Therefore, Algorithms 6 and 7 will eventually halt, because there are a finite number of tuples in relations. Because the last iteration is completed by processing all the clusters without deleting any further tuples, the consistency property is correctly enforced.

5.4 Empirical Evaluations

Below, we empirically evaluate the localization of R($*$,m)C in Section 5.4.2, and the queue-management algorithms in Section 5.4.3.

5.4.1 Experimental set-up

The impact of localizing and managing the propagation queues is evaluated for finding the first solution of a CSP using backtrack search. The consistency properties listed in Table 5.1, as well as GAC [Bessiere *et al.*, 2005] and maxRPWC [Bessiere *et al.*, 2008], are compared by enforcing them as full lookahead strategies in the backtrack search using the domain/degree heuristic for dynamic variable ordering.

Table 5.1: Tested consistencies.

Type	$m = 2, 3, 4$	$m = \|\psi(cl_i)\|$
global	wR($*$,m)C	
local	cl-wR($*$,m)C	cl-R($*$,$\|\psi(cl_i)\|$)C

The experiments were conducted on benchmark problems from the CSP Solver Competition,[1] with a total of 679 difficult CSP instances.[2] The processing time for each instance was limited to two hours.[3] The results are split into satisfiable and unsatisfiable problems.

5.4.2 Evaluating the localization

In Table 5.2, the results of localization of R($*$,m)C are reported in terms of:

- Completed: the number of tested instances that search solved within the allocated time.
- BT-free: the number of tested instances that search solved in a backtrack-free manner.
- Min(#NV): the number of tested instances where search visited the least number of nodes.

[1] `www.cril.univ-artois.fr/CPAI09`

[2] The benchmarks tested are: aim-(50, 100, 200), composed-(25-10-20, 25-1-2, 25-1-25, 25-1-40, 25-1-80, 75-1-2, 75-1-25, 75-1-40, 75-1-80), dag-rand, dubois, graphColoring-(hosExtConvert, mug, register-mulsol, register-zeroin, sgb-book, sgb-games, sgb-miles, sgb-queen), hanoi, modifiedRenault, QCP-15, rand-(10-20-10, 8-20-5), rlfap(GraphsMod, Scens11, ScensMod), ssa, and tightness0.9. Their characteristics are provided in Appendix E.

[3] We carried out our experiments on a large computer cluster with a heavy and variable load. Although the cluster's hardware is homogeneous, the load varies and affects the precision of the clock time. For this reason, we measured the time in seconds computed from the instruction count instead of the clock time, after normalizing the instruction cost and the CPU speed across all runs. We compared the results in instruction counts and CPU time. Although they were qualitatively similar, the former was more reproducible and precise.

- Fastest: the number of tested instances that were solved fastest by the corresponding algorithm (within a precision of 256 msec).

For the algorithms enforcing the localized properties (i.e., cl-wR(∗,2)C, cl-wR(∗,3)C, cl-wR(∗,4)C, and cl-R(∗,$|\psi(cl_i)|$)C), we used the STATIC queue-management strategy.

Table 5.2: Aggregate results comparing R(∗,m)C and cl-wR(∗,m)C.

		Domain based		wR(∗,2)C		wR(∗,3)C		wR(∗,4)C		
	#Instances	GAC	maxRPWC	global	local	global	local	global	local	cl-R(∗,$\|\psi(cl_i)\|$)C
Completed	UNSAT	167	142	170	167	191	232	190	225	**285**
	479	34.9%	29.6%	35.5%	34.9%	39.9%	48.4%	39.7%	47.0%	**59.5%**
	SAT	174	159	**179**	178	147	164	132	151	152
	200	87.0%	79.5%	**89.5%**	89.0%	73.5%	82.0%	66.0%	75.5%	76.0%
BT-Free	UNSAT	0	30	70	39	97	104	141	104	**187**
	479	0.0%	6.3%	14.6%	8.1%	20.3%	21.7%	29.4%	21.7%	**39.0%**
	SAT	44	49	55	37	65	30	**68**	32	39
	200	22.0%	24.5%	27.5%	18.5%	32.5%	15.0%	**34.0%**	16.0%	19.5%
Min(#NV)	UNSAT	19	39	74	43	105	116	154	134	**231**
	479	4.0%	8.1%	15.4%	9.0%	21.9%	24.2%	32.2%	28.0%	**48.2%**
	SAT	47	51	66	37	72	40	**87**	66	**87**
	200	23.5%	25.5%	33.0%	18.5%	36.0%	20.0%	**43.5%**	33.0%	**43.5%**
Fastest	UNSAT	72	14	20	35	18	106	17	35	**184**
	479	15.0%	2.9%	4.2%	7.3%	3.8%	22.1%	3.5%	7.3%	**38.4%**
	SAT	**122**	31	45	47	26	32	8	26	34
	200	**61.0%**	15.5%	22.5%	23.5%	13.0%	16.0%	4.0%	13.0%	17.0%

As expected, the localization weakens the level of consistency enforced but is quicker to process. For $m = 2, 3, 4$, wR(∗,m)C globally outperforms cl-wR(∗,m)C in terms of 'fastest' and 'completed.'

cl-R(∗,|$\psi(cl_i)$|)C), a localized strategy, is clearly the overall winner. The highlighted cells in Table 5.2 indicate that cl-R(∗,|$\psi(cl_i)$|)C) outperforms all tested consistency levels in the UNSAT category on all four reported criteria. However, on SAT instances, it was not the best on time related performance (on the criteria 'completed' and 'fastest'). Also, wR(∗,4)C solved more instances backtrack-free than cl-R(∗,|$\psi(cl_i)$|)C.

Regarding the time performance, we strongly suspect that the culprit is the implementation of the algorithm, which can benefit from various optimizations such as grouping tuples [Stergiou and Samaras, 2005]. A faster implementation would overcome this unique limitation of the algorithm. Regarding the criterion BT-Free, the performance of cl-R(∗,m)C is significantly improved by bolstering, as discussed in Chapter 6.

Figures 5.7 and 5.8 compare, pairwise, the running time of the algorithms on individual instances. Note the logarithmic scale. The diagonal line indicates equal

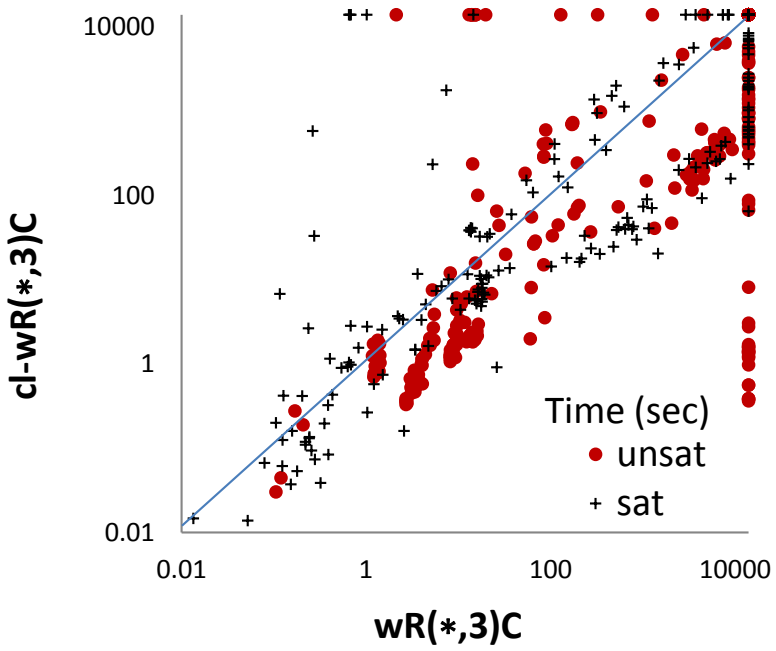

Figure 5.7: Comparing local to global for wR(∗,3)C.

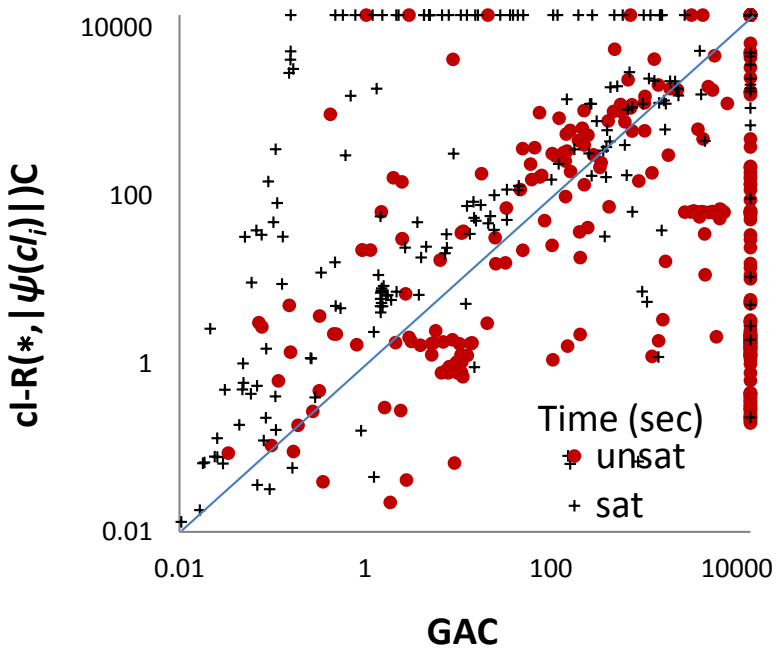

Figure 5.8: Comparing cl-R(∗,|$\psi(cl)$|)C to GAC.

performance of both algorithms. The points above the diagonal indicate that the corresponding instances are solved faster by the algorithm on the horizontal axis, and vice versa. The points along the top (right) edge indicate that the corresponding instances timed out for the algorithm on the vertical (horizontal) axes.

Figure 5.7 compares the running time of wR(∗,3)C to cl-wR(∗,3)C. It shows that localization (cl-wR(∗,3)C) outperforms the global version of the algorithm (wR(∗,3)C) by roughly an order of magnitude (points below the diagonal).

Figure 5.8 compares the performance of GAC, which is the fastest on SAT instances, to cl-R(∗,|$\psi(cl_i)$|)C, which is the fastest algorithm on UNSAT instances. The large number of points clustered around the diagonal confirm that consistencies significantly stronger than GAC are worthwhile even when the running time is considered.

The results strongly suggest that localization by tree decomposition is a crucial facilitator to increasing the consistency level while keeping the algorithm fast and

practical.

5.4.3 Evaluating queue-management strategies

Table 5.3 shows the empirical results for the localized consistencies (i.e., cl-wR(∗,2)C, cl-wR(∗,3)C, cl-wR(∗,4)C, and cl-R(∗,$|\psi(cl)|$)C) for each of the four queue-management strategies (RANDOM, STATIC, PRIORITY, and DYNAMIC). The number of instances completed, the number of instances on which the algorithm was fastest, and the average CPU time are reported. The CPU results are averaged over the instances completed by the various queue-management strategies for a given value of m. Therefore, the numbers should not be compared across different values of m.

We also give the number of instances on which the average time was computed. First, we consider $m = 2, 3, 4$. The number of completions for RANDOM and the structured orderings are similar. However, the average CPU time improves for the structured orderings, with the exception of $m = 2$ on UNSAT instances, and they are able to solve more instances fastest. The structured strategies are better than RANDOM. However, among the strategies that follow the tree structure (i.e., STATIC, PRIORITY, and DYNAMIC), there is not a clear winner.

Now, we consider $m = |\psi(cl)|$. The number of completions for the structured orderings significantly increases for UNSAT, while remaining similar for SAT. The structured orderings outperform STATIC in average CPU time, and they are able to solve more instances the fastest. Among the 'structured' strategies (i.e., STATIC, PRIORITY, and DYNAMIC), STATIC is the fastest.

Table 5.3: Comparison of the queue-management strategies.

		cl-wR(∗,2)C				cl-wR(∗,3)C				cl-wR(∗,4)C				cl-R(∗,$\|\psi(cl)\|$)C			
	#Instances	Random	Static	Priority	Dynamic	Random	Static	Priority	Dynamic	Random	Static	Priority	Dynamic	Random	Static	Priority	Dynamic
Completed	UNSAT	168	167	168	**171**	233	232	233	**234**	222	**225**	224	**225**	261	**285**	282	282
	479	35.1%	34.9%	35.1%	**35.7%**	48.6%	48.4%	48.6%	**48.9%**	46.3%	**47.0%**	46.8%	**47.0%**	54.5%	**59.5%**	58.9%	58.9%
	SAT	178	178	178	178	164	164	**165**	163	149	**151**	**151**	**151**	**154**	152	151	151
	200	89.0%	89.0%	89.0%	89.0%	82.0%	82.0%	**82.5%**	81.5%	74.5%	**75.5%**	**75.5%**	**75.5%**	**77.0%**	76.0%	75.5%	75.5%
Fastest	UNSAT	74	77	118	**120**	66	**157**	117	114	89	**159**	116	108	151	**220**	161	155
	479	15.4%	16.1%	24.6%	**25.1%**	13.8%	**32.8%**	24.4%	23.8%	18.6%	**33.2%**	24.2%	22.5%	31.5%	**45.9%**	33.6%	32.4%
	SAT	54	63	**152**	129	51	88	**111**	108	38	84	76	**92**	50	**88**	74	84
	200	27.0%	31.5%	**76.0%**	64.5%	25.5%	44.0%	**55.5%**	54.0%	19.0%	42.0%	38.0%	**46.0%**	25.0%	**44.0%**	37.0%	42.0%
Time (sec)	UNSAT	Nbr instances 167				Nbr instances 232				Nbr instances 220				Nbr instances 254			
	479	**413.3**	433.3	414.4	417.0	402.9	383.9	**366.3**	369.3	443.0	**410.6**	411.5	415.8	397.4	**318.9**	341.3	344.1
	SAT	Nbr instances 178				Nbr instances 162				Nbr instances 149				Nbr instances 150			
	200	500.2	490.5	**453.7**	454.7	622.9	601.6	599.2	**598.3**	324.2	313.8	**309.5**	**309.5**	571.1	**542.5**	557.7	546.8

Figures 5.9 and 5.10 compare the running time for two pairs of queue-management strategies, showing a fine-grained analysis of the experiment. Figure 5.9 compares the random and static orderings. Here, one can easily see that STATIC solves more instances than RANDOM (see the large number of points along the right edge). Notice that a large number of these points are solved *orders of magnitude* faster than RANDOM.

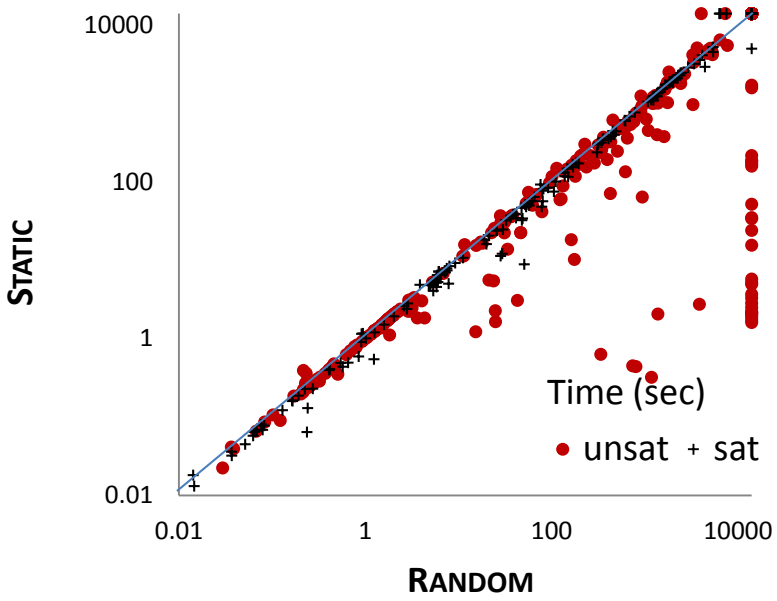

Figure 5.9: Comparing RANDOM and STATIC for cl-R($*$,$|\psi(cl)|$)C.

The different structured orderings showed no significant difference among themselves. Figure 5.10 compares the dynamic and static orderings. The majority of the points lie near the line, indicating that there is no clear winner between the two structured orderings.

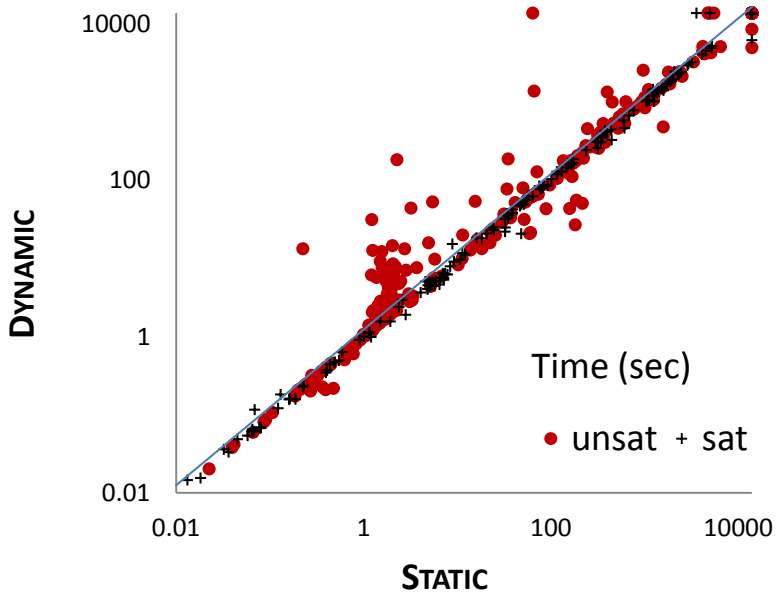

Figure 5.10: Comparing DYNAMIC and STATIC for cl-R($*$,$|\psi(cl)|$)C.

5.5 Conclusions

The experimental results clearly demonstrated that localization by tree decomposition is a crucial facilitator to increasing the consistency level while keeping the algorithm fast and practical. Moreover, with localization, we were able to exploit the problem structure to better manage the propagation queue of the consistency algorithm. The random ordering of the queue is never a good idea, and exploiting the structure of the problem is greatly beneficial. Interestingly, a simple static ordering that follows the linear order of the maximal cliques performs as well as more sophisticated strategies that attempt to 'follow' the propagation activity. Finally, and contrary to our hopes and expectations, PRIORITY and DYNAMIC do not discover inconsistency any earlier than STATIC.

Summary

In this chapter, we presented techniques to improve the performance of algorithms for higher-level consistency by localizing their application to the clusters of a tree decomposition and directing propagation along the tree decomposition. We proposed various strategies for managing the propagation queue of a consistency algorithm. We established that exploiting the structure of the problem in the management of the propagation queue of a consistency algorithm is beneficial and should not be ignored.

Chapter 6

Bolstering Propagation at Separators

In Chapter 5, we proposed to apply R($*$,m)C locally to the clusters of a tree decomposition of a CSP, improving performance by reducing the number of considered constraint combinations but also reducing the consistency level enforced. In this chapter, we propose to enhance propagation effectiveness between clusters by bolstering propagation at separators of a tree decomposition via the addition of redundant constraints on the variables of the separators. Results from this chapter have been published [Karakashian *et al.*, 2013].

6.1 Introduction

The tractability condition of CSPs that relates the *level of consistency* of a CSP to the treewidth of a tree decomposition requires perfect 'communication' between clusters [Freuder, 1982; Dechter, 2003]. A perfect communication between clusters requires generating a *unique* constraint over the separator's variables, but materializing

such a constraint is prohibitive in terms of space [Fattah and Dechter, 1996; Kask *et al.*, 2005]. In this chapter, we approximate this requirement to achieve practical tractability by *bolstering* constraint propagation along the tree via the addition of redundant constraints at the separators between clusters.

We present three schemes for bolstering propagation in the localized consistency property. We characterize the resulting consistency properties by comparing them, theoretically and empirically, to the original and localized R($*$,m)C, GAC, and maxR-PWC, and establish the benefits of our approach for solving difficult problems. The contributions of this chapter are as follows:

1. New relational consistency properties resulting from bolstering the propagation.
2. A theoretical characterization of those new properties.
3. An empirical evaluation of our approach establishing its benefits on difficult benchmarks, solving many problems in a backtrack-free manner and, thus, approaching 'practical tractability.'

6.2 Bolstering Propagation at Separators

We introduce the simple example of two adjacent clusters shown in Figure 6.1 to illustrate our approach. The variables in this example are $A, B, \ldots, F$ and the original constraints are $R_1, R_2, \ldots, R_7$. When a consistency algorithm is applied locally to a cluster, the effects of filtering relations in one cluster are propagated, or transferred, to a neighboring cluster only through the domains of the variables and those constraints common to both clusters (i.e., constraint R_4 in Figure 6.1). Thus, localization may compromise the effectiveness of constraint propagation across the entire problem. Here, we explore ways to remedy this situation by adding redundant constraints at the

separators to boost the transfer of information between clusters. Below, we introduce three schemes to this end, explain how to build them, and discuss their implementation.

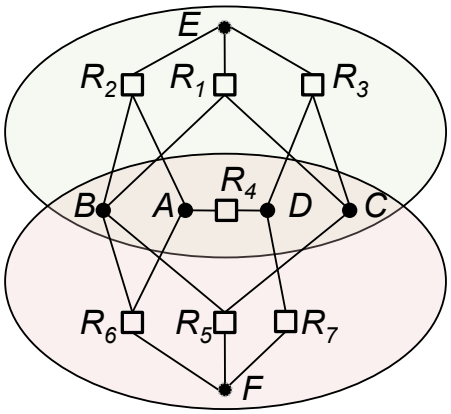

Figure 6.1: Two adjacent clusters.

6.2.1 Three bolstering schemes

According to the Cluster-Tree Elimination algorithm [Dechter, 2003; Kask *et al.*, 2005], we can solve the CSP in a backtrack-free manner after adding a unique constraint over the variables of each separator and enforcing R($*$,$|\psi(cl)|$)C (i.e., computing the minimal network induced by each cluster) in a two-pass process from the leaves of the tree to its root and back. Optimally, a single constraint over all the separator's variables would be added to every separator as shown as R_{sep} in Figure 6.2. Unfortunately, the size of such a relation grows exponentially with the number of variables in the separator, which is prohibitive in practice. Thus, trading space for time becomes necessary [Fattah and Dechter, 1996]. Instead of generating one unique constraint per separator, three schemes of increasing complexity and 'completeness' are considered:

1. Adding projections of all existing constraints.

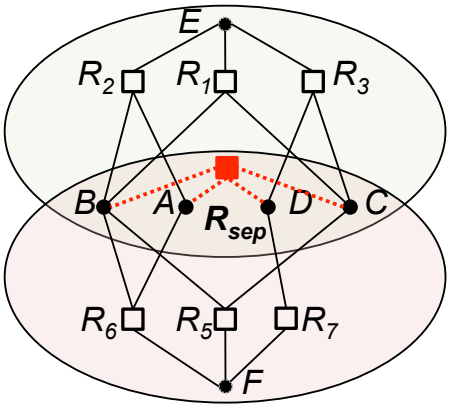

Figure 6.2: Unique constraint over the separator's variables.

2. Adding new binary constraints.

3. Adding new 'clique' constraints.

Below, we describe how the constraints are added to the separators for each of the above three schemes. (In all cases, the constraints in the clusters are normalized.) Then we describe how the relations of the binary and clique constraints are generated.

6.2.1.1 Adding constraint projections

We add to each cluster the projection of all the constraints outside the cluster onto the variables inside the cluster, then we normalize the constraints in the cluster. That is, whenever the scope of a constraint is a subset of another constraint, we merge the two constraints (see Section 6.2.2). In the example, this process results in the new constraint R'_3 added to the lower cluster as shown in Figure 6.3.

6.2.1.2 Adding binary constraints

In addition to the projected relations, we add to the separator all non-existing binary constraints that result from triangulating the subgraph induced by the separator's

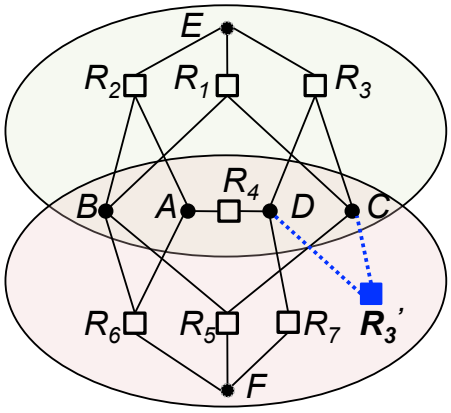

Figure 6.3: Constraint projections.

variables on the primal graph of the CSP. The subgraph induced by the separator on the primal graph of the CSP before triangulation is shown in Figure 6.4, and after triangulation in Figure 6.5, where BD is a fill-in edge resulting from triangulation. This process results in the addition of the constraint R_a ($scope(R_a)$=$\{B, D\}$) in the above example as shown in Figure 6.6.

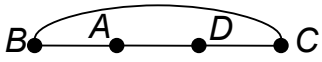

Figure 6.4: Induced primal-graph.

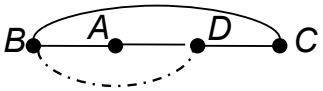

Figure 6.5: Triangulated induced primal-graph.

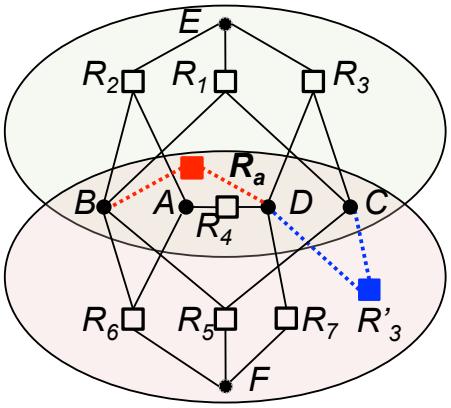

Figure 6.6: Binary constraints.

The process is applied as follows. First, the subgraph induced by the separator's variables on the primal graph is extracted. Second, the induced subgraph is triangulated using the min-fill heuristic [Kjærulff, 1990]. Third, a binary constraint is generated for each fill-in edge generated by the min-fill heuristic. For example, the relation R_a is added to the separator in Figure 6.6 because after triangulating the primal graph induced by the variables in the separator, the fill-in edge BD is added in Figure 6.5.

6.2.1.3 Adding clique constraints

In addition to those projected relations, we add to the separators all non-existent non-binary constraints whose scopes are the maximal cliques of the triangulated subgraph induced by the separator's variables on the primal graph of the CSP. For this reason, we refer to those constraints as *clique constraints*. The relations R_x and R_y are added in the example as shown in Figure 6.7.

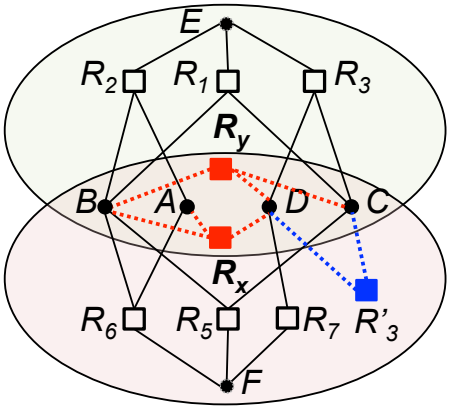

Figure 6.7: Clique constraints.

The scopes of the constraints are generated as follows. As for binary constraints, the subgraph induced by the separator's variables on the primal graph is first extracted. Second, the induced subgraph is triangulated using the min-fill heuristic [Kjærulff, 1990]. Third, the maximal cliques are identified in the resulting chordal graph using

the MaxCliques algorithm [Golumbic, 1980]. Fourth, for each maximal clique, a constraint over the variables in the maximal clique is generated. For example, in Figure 6.7, we add the constraints R_x and R_y whose scopes are $\{A, B, D\}$ and $\{B, C, D\}$, respectively, to the separator.

6.2.1.4 Generating the relations of the binary and clique constraints

In order to generate the relations of the binary and clique constraints added to the separators, each cluster C_i is visited in the order of the clusters given by the elimination ordering shown in Figure 5.3.

At each cluster, the selected consistency property is enforced to filter the existing constraints (those whose relations are already defined). Then we generate the relations of the constraints added at the separator between the cluster and its parent. The relation of a binary or a clique constraint R_x in the separator of clusters C_i and C_j is generated as follows:

1. Every constraint in $\psi(C_i)$ is projected on the $scope(R_x)$ and stored in the set of relations $\mathcal{R} = \{R_j | R_i \in \psi(C_i), R_j = \pi_{scope(R_x)} R_i\}$

2. The relations in $\mathcal{R}$ are joined together to yield R_x: $R_x = \bowtie_{R_j \in \mathcal{R}} R_j$.

The join operation is implemented using a variation of AllSol (Algorithm 2). It performs a backtrack search for all solutions on the dual CSP induced by the relations in $\mathcal{R}$ using forward checking, and outputs a tuple in R_x for every solution it finds. However, unlike AllSol, it does not filter the input relations.

For example, the clique constraints R_x and R_y shown in Figure 6.8 are generated using the constraints in the cluster: R_1, R_2, R_3 and R_4. The constraints R_x and R_y are generated independently as follows:

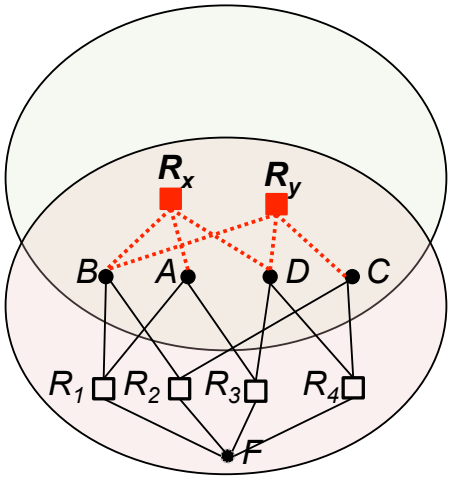

Figure 6.8: Separator constraint example.

1. The set of projected relations for R_x are prepared: $\mathcal{R} = \{R_{AB}, R_B, R_{AD}, R_D\}$ where:

$$R_{AB} = \pi_{\{A,B,D\}} R_1 \qquad R_B = \pi_{\{A,B,D\}} R_2$$

$$R_{AD} = \pi_{\{A,B,D\}} R_3 \qquad R_D = \pi_{\{A,B,D\}} R_4$$

2. Then, the AllSol algorithm is used to find all solutions to the dual CSP with the dual variables in $\mathcal{R}$. Each solution is a tuple in R_x, and the process is equivalent to $R_x = R_{AB} \bowtie R_B \bowtie R_{AD} \bowtie R_D$.

3. Afterwards, the process is repeated to generate R_y. The set of projected relations for R_y are prepared: $\mathcal{R} = \{R_B, R_{BC}, R_{BD}, R_{CD}\}$ where:

$$R_B = \pi_{\{B,C,D\}} R_1 \qquad R_{BC} = \pi_{\{B,C,D\}} R_2$$

$$R_{BD} = \pi_{\{B,C,D\}} R_3 \qquad R_{CD} = \pi_{\{B,C,D\}} R_4$$

4. Then, the AllSol algorithm is used to find all solutions to the dual CSP with the dual variables in $\mathcal{R}$. Each solution is a tuple in R_y, and is equivalent to $R_y = R_B \bowtie R_{BC} \bowtie R_{BD} \bowtie R_{CD}$.

The complexity for generating a constraint R_x is $\mathcal{O}(|\psi(C_i)| \cdot d^{|scope(R_x)|})$, where d

is the maximum domain size of the variables in the $scope(R_x)$. The factor $|\psi(C_i)|$ is due to the number of 'original' constraints that we may have to project on the scope of R_x. This procedure can generate a constraint that is tighter than the constraint obtained by taking the cross product of the domains of the variables in the $scope(R_x)$. However, it does not necessarily generate the tightest constraint.

6.2.2 Transferring information between clusters

The information transferred from a cluster to its parent (or its child) transits via the domains of the separator's variables and the added redundant constraints. In the above three bolstering schemes, the constraints are normalized to save space and processing effort. Due to this normalization, a mechanism is needed to ensure the fullest transfer of information between constraints of overlapping scopes in neighboring clusters. Assume that cluster cl_i is being processed after its neighbor cl_j was processed. For every relation R_j in cl_j, consider s the set of variables in the scope of R_j that are also in the separator between cl_i and cl_j, $s = scope(R_j) \cap \chi(cl_i) \cap \chi(cl_j)$. s must be a subset of some constraint R_i of cl_i (by construction of the projected constraints). Before processing cl_i, R_i must be filtered given R_j in a process akin to directional R(∗,2)C consistency. In the example of Figure 6.3, R_6, R_5, and R'_3 are used to filter R_2, R_1 and R_3, respectively.

6.3 Resulting Consistency Properties

In Chapter 5, the consistency property corresponding to the localized version of R(∗,m)C was denoted by cl-R(∗,m)C. Here, cl+proj-R(∗,m)C, cl+bin-R(∗,m)C, and cl+clq-R(∗,m)C denote the properties resulting from combining localization and the addition of projected constraints, binary constraints, and clique constraints,

respectively. Intuitively speaking, localization weakens R(∗,m)C because localization ignores combinations across clusters. In contrast, adding constraints increases the level of consistency. In Figure 6.9, the new properties are compared to GAC, maxRPWC, R(∗,m)C, and cl-R(∗,m)C for $m = 2, 3, 4, |\psi(cl_i)|$. In this figure, the property at the source of an arrow is strictly weaker than the one at which the arrow points. It is interesting to note that R(∗,2)C, cl+proj-R(∗,2)C, and cl+bin-R(∗,2)C are equivalent, as are R(∗,3)C and cl+proj-R(∗,3)C.

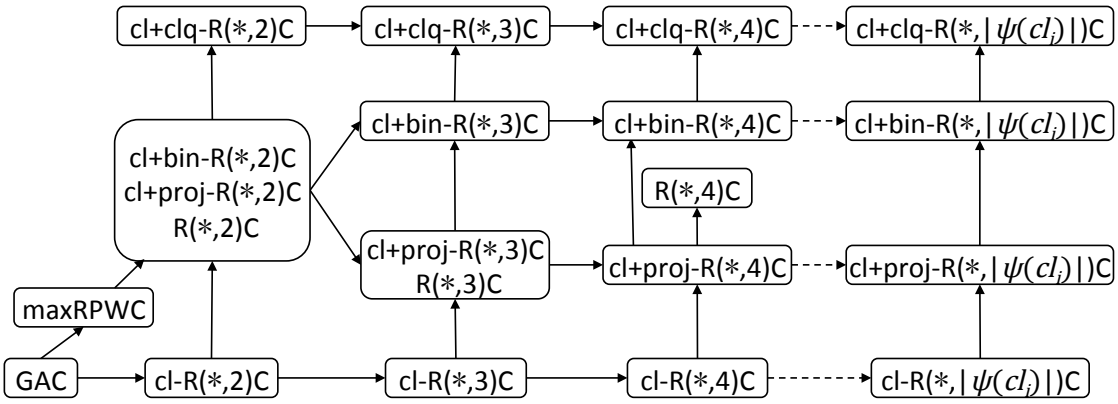

Figure 6.9: Comparing consistency properties.

When one considers the dual graph of a CSP, some edges in the dual graph may be redundant and could be removed without changing the set of solutions to the problem [Dechter and Dechter, 1987; Dechter and Pearl, 1989; Janssen *et al.*, 1989; Dechter, 2003]. In Chapter 3.3, we proposed to remove redundant edges using the algorithm of Janssen *et al.*. This operation reduces the number of constraint combinations that R(∗,m)C must consider, and results in significant cost savings in time and space. The enforced consistency is strictly weaker than R(∗,m)C and denoted wR(∗,m)C. Given the advantageous cost of wR(∗,m)C, in our empirical evaluations in Section 6.5, the properties shown in Figure 6.9 were not implemented. Instead, their weakened versions were tested (i.e., cl-wR(∗,m)C, cl+proj-wR(∗,m)C,

cl+bin-wR(∗,m)C, and cl+clq-wR(∗,m)C), obtained after removal of redundant edges in the dual graph. The redundant edges are removed for $m = 2, 3, 4$. However, for $m = |\psi(cl_i)|$, no edges are removed, because only a single combination per cluster is considered. While the relationships shown in Figure 6.9 do not necessarily hold for the weakened versions of the consistency properties, in the experiments, they applied for the weakened properties as well. Moreover, the redundant constraints helped regain some of the consistency strength lost due to removing the redundant edges from the dual graph.

Theorem 12 *R(∗,2)C and cl+proj-R(∗,2)C are equivalent.*

Proof: See Appendix C.5.

When the redundant edges are removed, cl+proj-wR(∗,2)C and wR(∗,2)C are also equivalent because wR(∗,2)C and R(∗,2)C are equivalent as shown in Chapter 3.

Theorem 13 *cl+proj-R(∗,2)C and cl+bin-R(∗,2)C are equivalent.*

Proof: See Appendix C.5.

Theorem 14 *cl+bin-R(∗,m)C is strictly stronger than cl+proj-R(∗,m)C for* $m \geq 2$.

Proof: See Appendix C.5.

Theorem 15 *R(∗,3)C and cl+proj-R(∗,3)C are equivalent.*

Proof: See Appendix C.5.

Theorem 16 *R(∗,m)C is strictly stronger than cl+proj-R(∗,m)C for* $m > 3$.

Proof: See Appendix C.5.

Theorem 17 *cl+clq-R(∗,m)C is strictly stronger than cl+bin-R(∗,m)C.*

Proof: See Appendix C.5.

6.4 Related Work

The algorithm for cl-R($*$,m)C with bolstering is an implementation in the spirit of the Cluster-Tree Elimination algorithm [Kask *et al.*, 2005]. Because unique 'global' constraints are not added at the separators, neither converging nor solving the CSP is guaranteed in two passes. Thus, it is used as a full-lookahead schema in backtrack search. Nonetheless, experiments show that it yielded backtrack-free search on a large number of instances.

Fattah and Dechter [1996] study space-time tradeoffs of tree clustering by increasing the cluster sizes to reduce the separators' sizes. In the approach presented in Section 6.2, the space requirement is reduced by replacing the unique 'global' constraints at the separators with constraints of smaller scopes.

Based on the bucket elimination method [Dechter, 1996; 1999], the mini-bucket elimination (MBE) algorithm generates relations from the mini-buckets (which are partitions of the relations in a cluster) and then projects them on the separators [Rollon and Dechter, 2010]. This approach differs from the bolstering schemes, which only generates relations on the separators. Moreover, the sizes of the mini-buckets and those of the generated constraints are bounded by a fixed parameter z chosen by trial and error, while the sizes in the clique bolstering are automatically determined by the structure of the constraint graph at the separators.

The consistency property w-SC enforces inverse consistency (by domain filtering) of a relaxed CSP obtained by removing constraints in order to guarantee a tree decomposition of bounded width w [Jégou and Terrioux, 2010]. We use the decomposition to process the consistency locally and do not relax the CSP.

Algorithms for higher-order consistency by domain filtering that remain weaker than R($*$,m)C have been proposed [Bessiere *et al.*, 2008] and more recently improved

upon [Paparrizou and Stergiou, 2012]. We compare our results to maxRPWC in the experiments.

Stergiou and Samaras [2005] improved the performance of an arc-consistency algorithm for the dual CSP (i.e., R(∗,2)C) by grouping tuples that have the same supports. While we target stronger consistencies, their technique could be used to improve the performance of our algorithms.

6.5 Empirical Evaluations

In this section, an empirical evaluation of the three bolstering schemes is presented.

6.5.1 Experimental set-up

We compare the advantages of enforcing the properties listed in Table 6.1 to those of enforcing GAC [Bessiere *et al.*, 2005] and maxRPWC [Bessiere *et al.*, 2008]. All consistencies are enforced as full lookahead strategies in a backtrack search using the domain/degree heuristic for dynamic variable ordering. The benchmarks are selected from the CSP Solver Competition.[1] Because we target problems that require higher consistency levels than provided by GAC, we selected 32 benchmarks that are not easily solved by GAC, but compared against GAC as a baseline for evaluation. These benchmarks are listed in Appendix E with their characteristics.

To evaluate the impact of bolstering propagation two contexts are distinguished: solvable and unsolvable CSPs. Indeed, difficult, unsolvable CSPs are expected to be more challenging for GAC and to require higher-level consistency. In the selected benchmarks, 479 instances were unsatisfiable and 200 satisfiable. We set the maximum

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

Table 6.1: Tested consistencies.

Type	$m = 2, 3, 4$	$m = \|\psi(cl_i)\|$
global	wR(∗,m)C	
local	cl-wR(∗,m)C	cl-R(∗,$\|\psi(cl_i)\|$)C
projection	cl+proj-wR(∗,m)C	cl+proj-R(∗,$\|\psi(cl_i)\|$)C
binary	cl+bin-wR(∗,m)C	cl+bin-R(∗,$\|\psi(cl_i)\|$)C
clique	cl+clq-wR(∗,m)C	cl+clq-R(∗,$\|\psi(cl_i)\|$)C

processing time per instance to two hours. In Table 6.2, the results of bolstering are reported for each consistency algorithm in terms of:

- Completed: the number of tested instances that search solved within the allocated time.
- BT-free: the number of tested instances that search solved in a backtrack-free manner.
- Min(#NV): the number of tested instances where search visited the least number of nodes.
- Fastest: the number of tested instances that were solved the quickest by the corresponding algorithm (within a precision of 256 msec).

6.5.2 Aggregate results

First, we discuss $m = |\psi(cl_i)|$, which is a localized strategy and also the strongest consistency property. It is clearly the overall winner. The highlighted cells in Table 6.2 indicate that $m = |\psi(cl_i)|$ outperforms all tested consistency levels in both SAT/UNSAT categories and on all four reported criteria with only two exceptions. Both exceptions are on SAT instances, and are related to time performance (on the criteria 'completed' and 'fastest'). This result strongly supports two aforementioned claims: *a*) higher-level consistencies are useful for approaching tractability in practice;

and *b*) localization by tree decomposition is a crucial facilitator to increasing the consistency level. (Indeed, very high-level consistencies are not possible without localization because of the number of constraint combinations that need to be stored and manipulated.) The two exceptions are related to the implementation of the algorithm as discussed in Section 5.4.

Second, a little bolstering is great, but too much may be detrimental. For a given m value, we see that, grossly speaking, 'projection' yields the best results (versus global, localization, binary, and clique). Two reasons may explain why heavier bolstering (i.e., binary and clique) are not the winners that were expected: *a*) the heavier the bolstering, the more expensive the processing (indeed, the completion rate of clique degrades); and *b*) in most of the tested instances clusters seem to overlap heavily making the generation of redundant constraints overkill. One may want to decide locally based on the overlap of the clusters which level of bolstering to apply.

Finally, localization and projection always outperformed 'global' for all considered criteria, particularly when $m \geq 3$. For $m = 2$, localization and projection are equivalent to the global strategy confirming the theory of Section 6.3. Consequently, the additional processing is wasted.

Table 6.2: Aggregate results of the bolstering schemes.

		Domain based		wR(∗,2)C					wR(∗,3)C					wR(∗,4)C					R(∗,|ψ(cl_i)|)C			
	#Instances	GAC	maxRPWC	global	local	projection	binary	clique	global	local	projection	binary	clique	global	local	projection	binary	clique	local	projection	binary	clique
Completed	UNSAT	167	142	170	167	172	169	162	191	232	237	232	218	190	225	230	226	223	285	**286**	282	271
	479	34.9%	29.6%	35.5%	34.9%	35.9%	35.3%	33.8%	39.9%	48.4%	49.5%	48.4%	45.5%	39.7%	47.0%	48.0%	47.2%	46.6%	59.5%	**59.7%**	58.9%	56.6%
	SAT	174	159	**179**	178	176	169	104	147	164	155	149	111	132	151	153	147	112	152	138	124	113
	200	87.0%	79.5%	**89.5%**	89.0%	88.0%	84.5%	52.0%	73.5%	82.0%	77.5%	74.5%	55.5%	66.0%	75.5%	76.5%	73.5%	56.0%	76.0%	69.0%	62.0%	56.5%
BT-free	UNSAT	0	30	70	39	70	70	74	97	104	139	139	132	141	104	142	142	149	187	**223**	**223**	213
	479	0.0%	6.3%	14.6%	8.1%	14.6%	14.6%	15.4%	20.3%	21.7%	29.0%	29.0%	27.6%	29.4%	21.7%	29.6%	29.6%	31.1%	39.0%	**46.6%**	**46.6%**	44.5%
	SAT	44	49	55	37	53	52	38	65	30	65	63	53	68	32	75	67	55	39	**77**	71	58
	200	22.0%	24.5%	27.5%	18.5%	26.5%	26.0%	19.0%	32.5%	15.0%	32.5%	31.5%	26.5%	34.0%	16.0%	37.5%	33.5%	27.5%	19.5%	**38.5%**	35.5%	29.0%
Min(#NV)	UNSAT	17	37	73	43	72	72	77	103	115	147	147	144	150	127	159	159	167	220	**249**	248	239
	479	3.5%	7.7%	15.2%	9.0%	15.0%	15.0%	16.1%	21.5%	24.0%	30.7%	30.7%	30.1%	31.3%	26.5%	33.2%	33.2%	34.9%	45.9%	**52.0%**	51.8%	49.9%
	SAT	47	51	64	37	62	61	39	69	38	76	70	61	78	63	108	94	73	83	**111**	100	79
	200	23.5%	25.5%	32.0%	18.5%	31.0%	30.5%	19.5%	34.5%	19.0%	38.0%	35.0%	30.5%	39.0%	31.5%	54.0%	47.0%	36.5%	41.5%	**55.5%**	50.0%	39.5%
Fastest	UNSAT	72	14	13	35	5	1	1	15	106	58	13	15	12	35	3	0	0	**176**	108	42	37
	479	15.0%	2.9%	2.7%	7.3%	1.0%	0.2%	0.2%	3.1%	22.1%	12.1%	2.7%	3.1%	2.5%	7.3%	0.6%	0.0%	0.0%	**36.7%**	22.5%	8.8%	7.7%
	SAT	**121**	31	45	47	23	14	12	26	30	27	13	11	7	26	14	9	10	34	18	13	12
	200	**60.5%**	15.5%	22.5%	23.5%	11.5%	7.0%	6.0%	13.0%	15.0%	13.5%	6.5%	5.5%	3.5%	13.0%	7.0%	4.5%	5.0%	17.0%	9.0%	6.5%	6.0%

6.5.3 A finer view

Figures 6.10 and 6.11 compare, pairwise, the running time of the algorithms on individual instances for $m = 3$ with increasing sophistication (i.e., localization, projection, then clique). Figure 6.12 compares the performance of GAC, which is the fastest on SAT instances, to cl-R(∗,$|\psi(cl_i)|$)C, which solved the largest number of instances backtrack-free. Note the logarithmic scale. The diagonal line plotted indicates equal performance of both algorithms. The points above the diagonal indicate that the corresponding instances are solved faster by the algorithm on the horizontal axis, and vice versa. The points along the top (right) edge indicate that the corresponding instances timed out for the algorithm on the vertical (horizontal) axes.

Figure 6.10 evaluates the cost of bolstering by projection (cl+proj-wR(∗,3)C). Most of the points are tightly clustered above the diagonal, reflecting the additional cost of processing the projected constraints. The cost of bolstering is compensated by a group of problem instances that are solved orders of magnitude faster with bolstering than without it (see points on the right edge).

Figure 6.11 compares projection and clique bolstering (cl+proj-R(∗,3)C versus cl+clq-wR(∗,3)C), and illustrates how bolstering can be overkill. However, the points below the diagonal suggest that results can be improved when 'clique bolstering' is selectively applied to avoid situations where adjacent clusters significantly overlap.

Again, the best results were obtained with $m = |\psi(cl_i)|$. In Figure 6.12, cl-R(∗,$|\psi(cl_i)|$)C is compared to GAC. The large number of points on the right edge correspond to the instances that were solved with the strong consistency when GAC was insufficient to solve them. Despite the difference in the cost of each application of cl-R(∗,$|\psi(cl_i)|$)C compared to GAC, the overall cost of the backtrack search with full-lookahead is in general comparable even for problems that do not necessarily

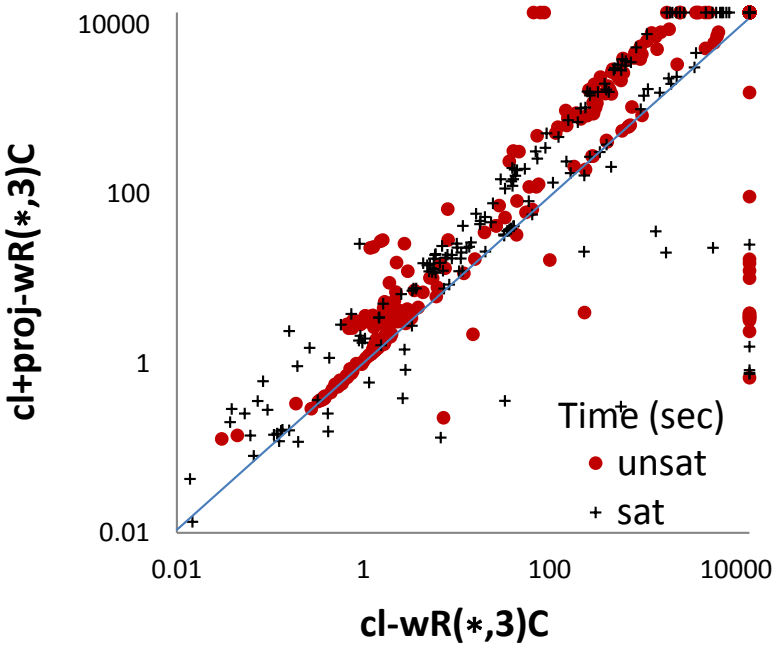

Figure 6.10: Projection.

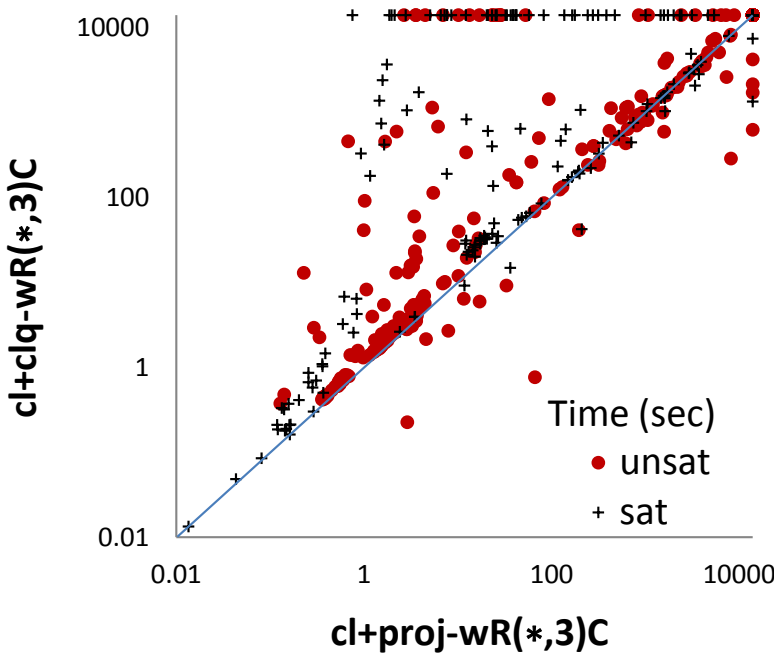

Figure 6.11: Clique.

require higher consistency levels. This fact can be seen from the large number of points clustered near the diagonal.

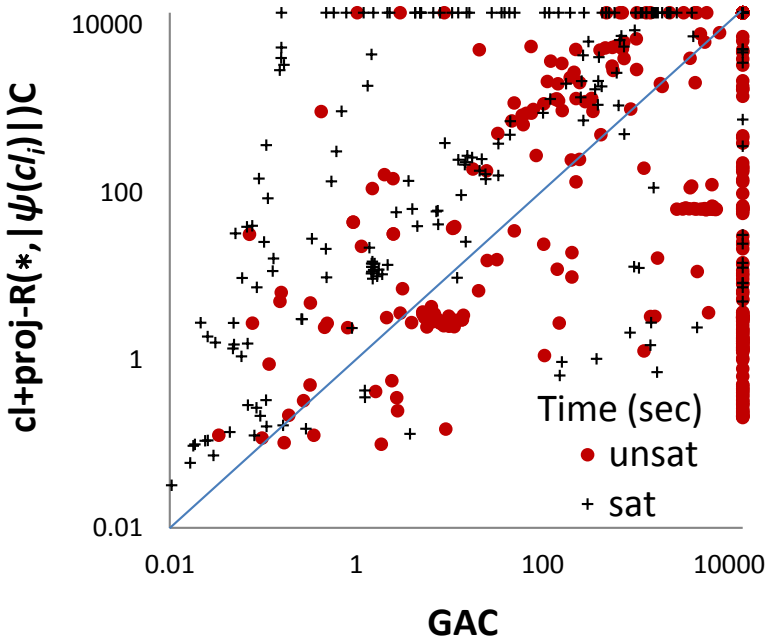

Figure 6.12: Compared to GAC.

6.5.4 Performance as a function of the treewidth

An interesting perspective on the results of Table 6.2 compares the performance of search endowed with each of consistency levels as a function of the value of the treewidth of the tree decomposition used in the experiments. The consistency levels compared are: GAC, cl+proj-R(∗,2)C, cl+proj-R(∗,3)C, cl-R(∗,$|\psi(cl_i)|$)C and cl+proj-R(∗,$|\psi(cl_i)|$)C. Those results are shown in four charts:

- Figures 6.13 and 6.14 show the number of instances completed for unsatisfiable and satisfiable instances, respectively. The horizontal axis represents the treewidth and the vertical axis represents the cumulative count of the completed instances within a given treewidth value.

- Figures 6.15 and 6.16 show the number of instances solved backtrack-free for unsatisfiable and satisfiable instances, respectively. The horizontal axis represents

the treewidth and the vertical axis represents the cumulative count of the instances solved in a backtrack-free manner within a given treewidth value.

Figure 6.13 shows that the compared algorithms have a similar performance for instances with treewidth less than 15. As the value of the treewidth increases, the difference in the performance of the consistency algorithms becomes increasingly more significant. The performance of cl-R($*$,$|\psi(cl_i)|$)C and cl+proj-R($*$,$|\psi(cl_i)|$)C is comparable. The algorithms are effective on instances with a treewidth value up to 60. For larger values, the corresponding curves become flat. GAC and cl+proj-R($*$,2)C are effective only on problems with a treewidth value up to 24. The performance of cl+proj-R($*$,3)C occupies a middle ground between the two above pairs. In conclusion, on unsatisfiable instances, higher consistency levels are more effective than lower consistency levels as the treewidth value increases.

For satisfiable instances (Figure 6.14), all algorithms seem to be effective throughout the considered treewidth range. The difference in performance does not become noticeable until treewidth values larger than 17 where GAC and cl+proj-R($*$,2)C seem slightly slightly more effective than the algorithms enforcing higher consistency levels. However, the difference between the tested algorithms is not as large as for unsatisfiable instances (Figure 6.13).

Figure 6.15 shows that GAC is unable to detect the inconsistency of the unsatisfiable instances. cl+proj-R($*$,2)C is effective for instances with treewidth up to 23. The higher consistencies are effective on problems with a larger treewidth value.

Finally, Figure 6.15 shows the higher the consistency levels clearly dominate the lower ones and that the difference increases with increasing treewidth values.

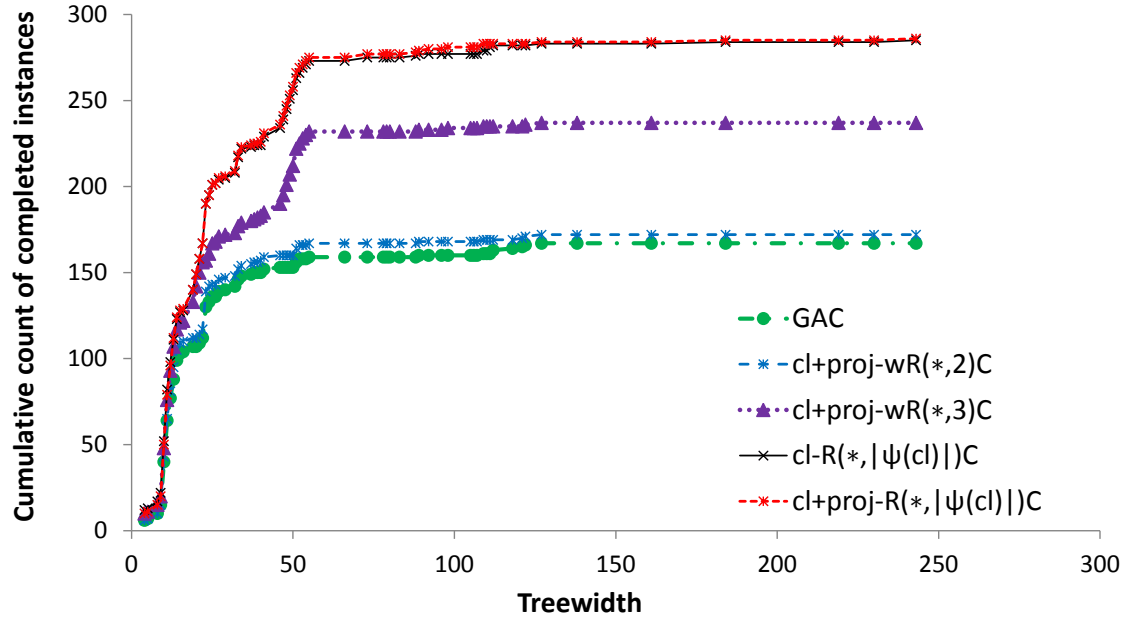

Figure 6.13: UNSAT instances: Cumulative count of completed instances within a treewidth value.

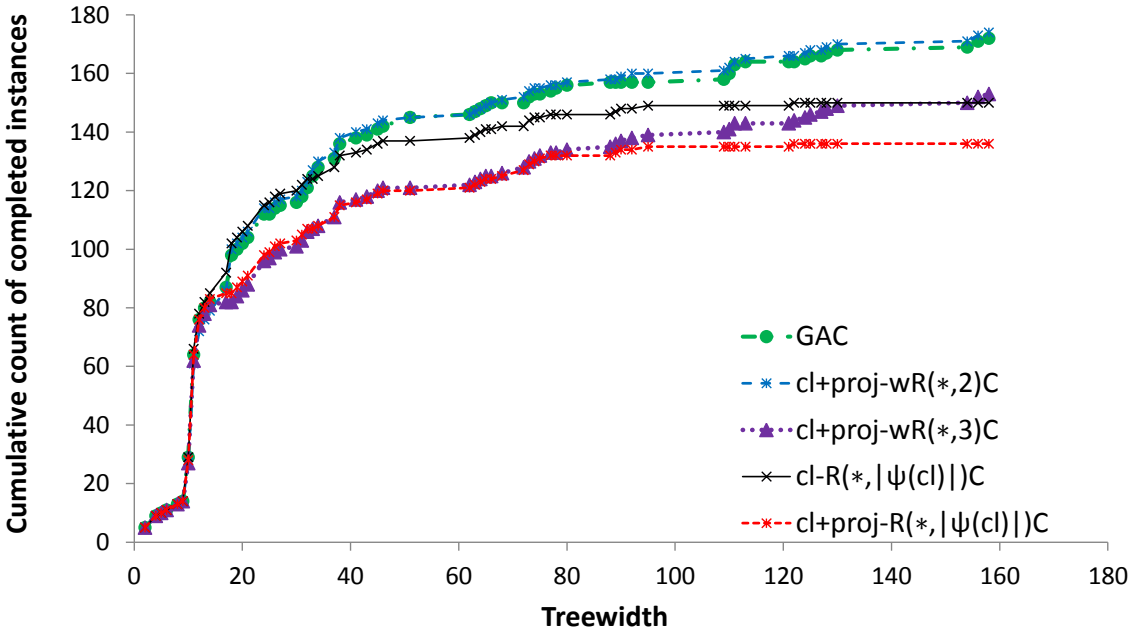

Figure 6.14: SAT instances: Cumulative count of completed instances within a treewidth value.

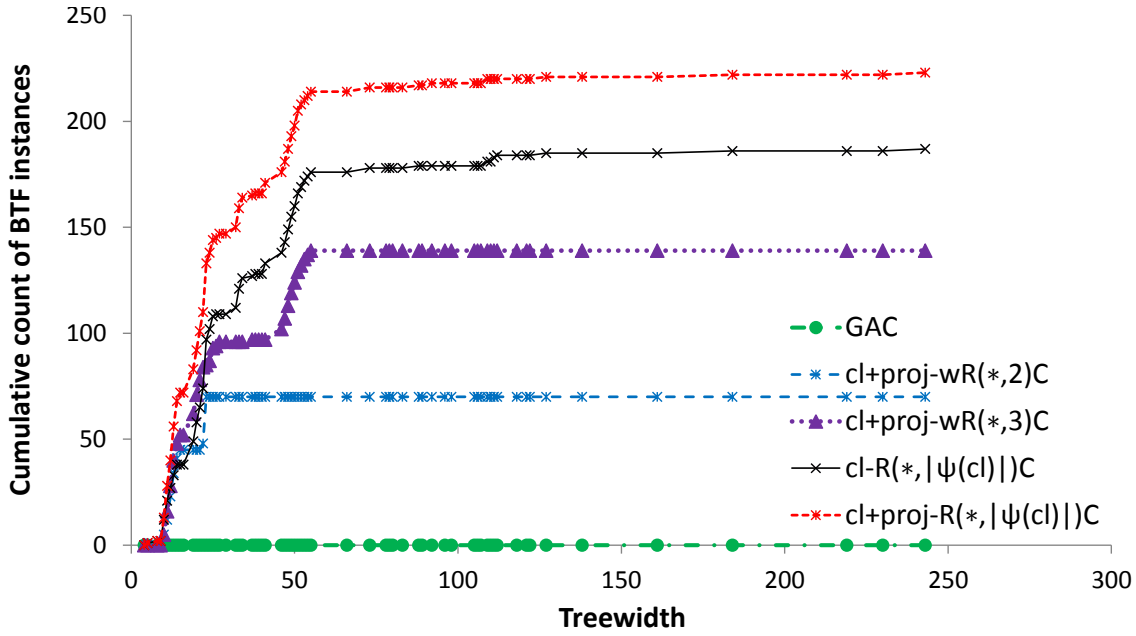

Figure 6.15: UNSAT instances: Cumulative count of number of instances solved backtrack-free within a treewidth value.

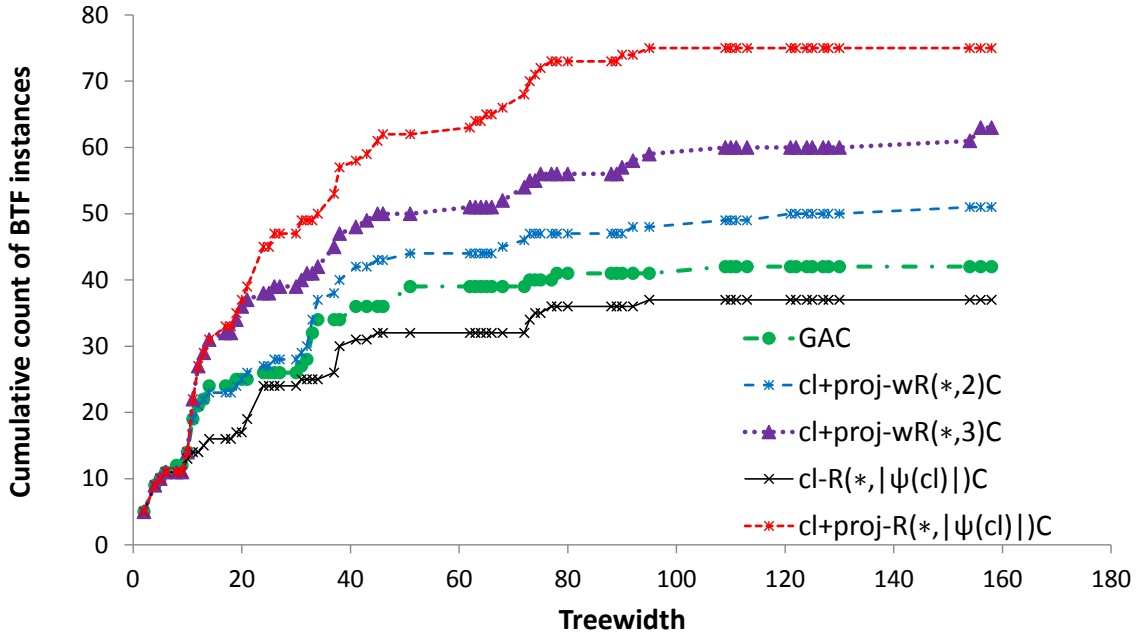

Figure 6.16: SAT instances: Cumulative count of number of instances solved backtrack-free within a treewidth value.

6.5.5 Merging decomposed tree clusters

About 90% of the variables in a cluster were also in the separator for the tree decompositions in most of the tested problems. The large overlap among the clusters is not favorable for our bolstering technique because many of the original constraints are repeated in the neighboring clusters and leave little opportunity for the bolstering to improve the propagation.

We merged clusters with high overlap into a larger cluster to obtain a tree decomposition with smaller overlaps among the clusters as described by Fattah and Dechter [1996]. The resulting tree decompositions had fewer and larger clusters, as a result of which cl-R($*$,m)C behaved more like R($*$,m)C and did not yield any significant improvements.

6.6 Conclusions

The results show that higher-level consistency properties are useful for approaching tractability in practice, and localization by tree decomposition and bolstering facilitated an increase in the consistency level.

Localization and projection always outperformed 'global' for all considered criteria, particularly when $m \geq 3$. However, too much bolstering was detrimental. For a given m value, projection yielded the best results, followed by binary and clique bolstering.

Summary

In this chapter, we presented techniques for bolstering the propagation at the separators with redundant constraints. The empirical results demonstrated orders of magnitude time savings over GAC and R($*$,m)C on difficult CSPs.

Chapter 7

Witness-Based Algorithm for Finding All of a CSP

In this chapter we propose an improvement to the backtrack search with tree decomposition (BTD) proposed by Jégou and Terrioux [2003]. Our technique improves BTD by avoiding the enumerating solutions in a subtree that cannot be extended to a global solution to the CSP.

7.1 Background

Backtrack search with tree decomposition (BTD) is a technique used for solving CSPs [Jégou and Terrioux, 2003] and for counting the number of solutions to a CSP [Favier *et al.*, 2009]. It applies backtrack search on some tree decomposition of the CSP following the ordering of the variables in the clusters of the tree decomposition. Moreover, BTD generates and stores, as search proceeds, partial solutions that succeed (i.e., goods) or fail (i.e., nogoods) in order to prevent the search process from re-exploring known partial solutions. Indeed, these goods and nogoods allow BTD to avoid visiting the

subtrees rooted at the corresponding separator when the same partial assignments of the variables in a separator are encountered again.

We use the example in Figure 7.1 to illustrate this situation. The variable ordering forces the variables in a cluster C_p to be instantiated before the variables in the subtrees rooted at C_p. Thus, given an assignment a_p to the variables in C_p, every

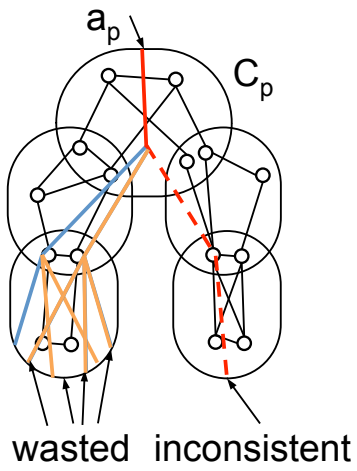

Figure 7.1: Illustrating wasteful enumeration of partial solutions.

consistent assignment to the variables in one subtree rooted at C_p can be extended to every assignment to the variables in the other subtrees. This property is crucial for counting the number of solutions because it allows us to independently count the number of solutions in each subtree rooted at C_p, and then multiply the counts to get the total number of solutions that extend a_p. However, if a_p cannot be extended to a solution in a subtree rooted at C_p, then a_p cannot be extended to any solution to the problem. Therefore, the solution counts in the other subtrees were not needed. In this case, the effort made to count the number of solutions was wasted. The cost of finding one solution is less than counting the number of solutions; thus we propose a new algorithm called WitnessBTD that first guarantees the existence of a *witness* solution to the problem, then counts the number of solutions, which can be extended

to the witness solution, in all the branches of the tree. Thus, we do not waste time counting the solutions in one branch of the tree if another branch has no solutions. The contributions of this chapter as as follows:

1. The introduction of a witness-based solution counting algorithm.
2. Theoretical analysis of the algorithm.
3. The empirical evaluation of the proposed algorithm.

7.2 WitnessBTD for Solution Counting

We now describe the algorithm for witness-based solution counting. It is similar to BTD except that, for a partial assignment to the variables in a cluster, it first finds the witness solution in each subtree before proceeding with counting the solutions in the subtrees.

WitnessBTD, the witness-based solution counting procedure, is given in Algorithm 8. The algorithm presented here is recursive, however, our implementation is iterative. In the iterative implementation, when a witness is found in a tree branch, the state of the search is preserved, so that when that branch is revisited to count all the solutions, the effort for finding the first solution is not repeated. The pseudocode of the iterative algorithm is give in Appendix D.

7.2.1 Notation used in pseudocode

Below, we summarize the notation used in Algorithm 8. We italicize the name of variables and attributes and we use small upper case letters for the names of functions and methods. The list of attributes is as follows:

- $\mathcal{A}$: Set of variables assignments (i.e., a partial solution)
- $\chi(C)$: Set of variables in the cluster C
- $Children(C)$: The children of cluster C
- $countSol$: State of counting solutions
- $consistent$: Indicates if the problem is consistent or should backtrack
- $curCl$: The current cluster being processed
- $curDom(A)$: The current domain of the variable A
- $curVariable$: The current variable being instantiated
- $satisfy$: Satisfiability state when searching for the witness solution
- $solCount$: The count of solutions for the assignment $\mathcal{A}$
- $state(C)$: The state of the search (solution counting or satisfiability check)
- V_{curCl}: Uninstantiated variables in cluster $curCl$

List of functions:

- GetGoodSolCount(C): Returns the stored solution count in goods for the current assignment of the separator
- HasGoodSolCount(C): Indicates if the solution count is stored for the current assignment of the separator
- Instantiate(A, v): Instantiates variable A with the value v in the current domain

- IsGood(C): Indicates if the current assignment of the separator is a known good
- IsNoGood(C): Indicates if the current assignment of the separator is a known nogood
- Propagate(A): Propagates the consistency algorithm given the instantiation of the variable A
- RecordGood(C): Records the good at cluster C with the optional solution count if available
- RecordNoGood(C): Records the nogood at cluster C

7.2.2 Recursive specification of WitnessBTD

Algorithm 8 takes as parameters the set of instantiated variables $\mathcal{A}$, the current cluster *curCl*, set of uninstantiated variables in the cluster V_C, and the state of the algorithm *state*, which could be in *satisfy* to perform a satisfiability search for a witness solution, or *countSol* to count the number of solutions. The algorithm is initially called with the empty assignment set, the root cluster, $\chi(C)$, and *countSol*. It recursively calls itself every time a new variable is instantiated, or when a child cluster is picked to be the current cluster. The block starting at Line 2 is visited to instantiate an uninstantiated variable in the current cluster. The current cluster is initially the root of the tree.

After instantiating a variable in Line 5, the consistency property is propagated in Line 6. If the consistency property is verified, the recursive call is made in Line 8 to instantiate the next variable. The recursive call in Line 8 returns zero if no solution is found, returns one if a solution is found and the state is *satisfy*, and otherwise returns

Algorithm 8: A recursive specification of WitnessBTD($\emptyset$,root,χ(root),countSol)

Input: $\mathcal{A}, curCl, V_{curCl}, state$, where $\mathcal{A}$ is the set of instantiated variables, $curCl$ is the current cluster, V_{curCl} is the set of uninstantiated variables in the cluster.

Output: Number of solutions in the problem.

```
1  solCount ← 0
2  if V_curCl then
3      curVariable ← A, A ∈ V_curCl
4      foreach v ∈ curDom(curVariable) do
5          Instantiate(curVariable, v)
6          consistent ←Propagate(curVariable)
7          if consistent then
8              count ←WitnessBTD (A ∪ {curVariable ← v},
               curCl, V_curCl \ {curVariable}, state)
9              if count > 0 then
10                 consistent ← true
11                 solCount ← solCount + count
12         if consistent AND state = satisfy then break
13 else
14     solCount ← 1
15     foreach C_i ∈ Children(curCl) do
16         if IsGood(C_i) then consistent ← true
17         else if IsNoGood(C_i) then consistent ← false
18         else
19             consistent ←WitnessBTD(A, C_i, χ(C_i) \ χ(C_curCl), satisfy) > 0
20             if consistent then RecordGood(C_i)
21             if consistent = false then RecordNoGood(C_i)
22         if consistent = false then return 0
23     if state = countSol then
24         foreach C_i ∈Children(C_i) do
25             if HasGoodSolCount(C_i) then count ←GetGoodSolCount(C_i)
26             else
27                 cont ←WitnessBTD(A, C_i, χ(C_i) \ χ(C_curCl), countSol);
                   RecordGood(C_i)
28             solCount ← solCount × count
29 return solCount
```

the number of solutions in the subtree rooted at the current cluster that extend the current assignment $\mathcal{A}$. The solution count for the partial assignment ending at the current variable is added in Line 11. Finally, if a solution is found and only a witness

solution is searched, the next value of the current variable is not instantiated, and the loop breaks in Line 12. However, if no solution is found or if counting the number of solutions, the search for solutions with the other values of the variable continues through the loop in Line 4.

When all the variables are instantiated in a cluster, the search continues to the variables in the children of the current cluster. The algorithm first checks if a witness solution is found in the loop in Line 15. For each child, it first checks for a good or nogood; if one is not found, it initiates a search for a single solution in Line 19. The recursive call returns zero in Line 22 if the current assignment cannot be extended to the variables in the subtree of the current cluster. The solution count is performed only if a witness is found, and the state is *countSol*.

The state is *countSol* when every solution found in the subtree rooted at the current cluster can be extended to a witness solution in the rest of the problem. Note that the state can be *countSol* while a witness solution does not exist in the subtree. This happens when the assignment in the current cluster changes to find the next solution. The witness solution found for the rest of the problem will be valid in this case, but it would have a different extension in the current subtree. For this reason the witness is searched inside the condition of Line 19, even if the state is *countSol*. Finally, if the solution count for the current assignment is not already computed and stored as good at the separator, checked in Line 25, the search for all solutions is called in Line 27.

7.3 Theoretical Analysis of the Algorithm

The number of solutions to a CSP is counted by finding the solutions. The tree decomposition allows us to find, at every tree node, all solutions in every branch rooted at that node independently, and then multiply the counts of solutions in each

branch to get the total count of solutions. WitnessBTD never searches for all solutions in a branch if the solutions in the branch cannot be continued to another branch, and consequently the number of solutions in that branch will be multiplied by zero. During the search for a witness solution, the implementation of the algorithm should preserve the state of the search in a branch so that the time spent for finding the first solution is not repeated when the same branch is revisited to count the rest of the solutions. We preserve the state of the search in our implementation. However, the effort for finding the first consistent assignment in a branch of the tree cannot always be saved, and may be repeated. We next explain this case and discuss at what cost it can be avoided.

Consider the case where the variables in cluster C_1 are instantiated in the ordering $\langle A, B, C, D\rangle$ as shown in Figure 7.2. Let the cause of the conflict be the value of B. As the search progresses, some other consistent value for B triggers a new search in C_1, thus the state of the search will be lost in C_2. Later, another assignment with the same values for C and D for which the solution count in C_2 was not performed, can be extended to the whole problem. In this case, the first solution searched in C_2 will be repeated, in order to count the rest of the solutions. When this situation occurs, the witness-based BTD may visit more nodes than the regular BTD. We show in the experiments in Section 7.4 that this situation in not likely to cause more node visits than the number of node visits saved. This situation can be avoided by ordering the variable instantiation according to their appearance in the subtrees. However, forcing the variables ordering interferes with the operation of the variable ordering heuristic, and is likely to be a bad trade-off. Another method to avoid this situation is to store the state of the search in each subtree that reaches a consistent instantiation but is not continued to find all solutions. This method is prohibitive because of its exponential memory requirement.

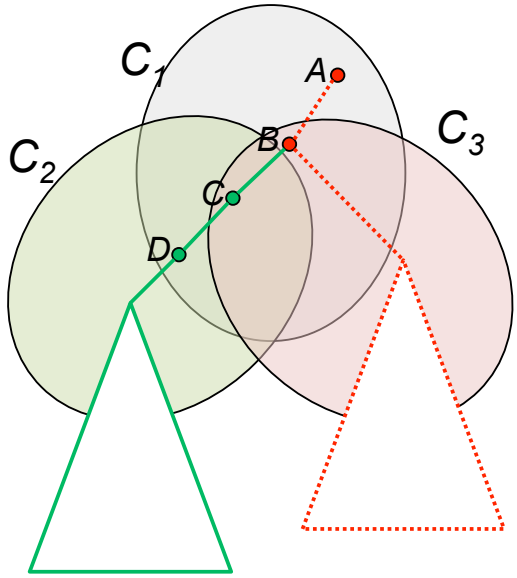

Figure 7.2: Case of repeated search.

7.4 Empirical Evaluations

All the experiments reported in this section are concerned with using the BTD for counting the number of solutions of a CSP [Favier *et al.*, 2009]. Below, we compare:

1. The performance of BTD with GAC against that of WitnessBTD with GAC.
2. The performance of the BTD with GAC, global R(∗,m)C, cl-R(∗,m)C, and cl-R(∗,m)C with bolstering.
3. The performance of the WitnessBTD with GAC, global R(∗,m)C, cl-R(∗,m)C, and cl-R(∗,m)C with bolstering.

7.4.1 Comparing WitnessBTD to BTD (with GAC)

The goal of this experiment is to assess the benefit of the witness-based strategy. We compare WitnessBTD with BTD as described by Favier *et al.* [2009].

7.4.1.1 Experimental set-up

We integrate with GAC2001 [Bessiere *et al.*, 2005] as full lookahead strategy in WitnessBTD and BTD and use the domain/degree heuristic for dynamic variable ordering inside clusters.

The experiments are conducted on the benchmarks of the CSP Solver Competition[1] with a time limit of two hours per instance. We divided the instances into unsatisfiable and satisfiable groups. We tested on 1647 unsatisfiable and 1320 satisfiable instances (see Table 7.1). Of these instances, BTD completes 740 unsatisfiable and 997 satisfiable instances, and WitnessBTD completed 743 unsatisfiable and 997 satisfiable instances. Both algorithms completed on 735 unsatisfiable and 994 satisfiable instances.

Table 7.1: Number of benchmark problems completed by each and both algorithms.

	BTD	WitnessBTD	Both
UNSAT (1,647)	740	**743**	735
SAT (1,320)	997	**999**	994
Total (2,967)	1,737	**1,742**	1,729

7.4.1.2 Results

The difference in the number of instances completed by each algorithm is clearly insignificant, and is due to the random time fluctuations between runs. Thus, we only consider the instances completed by both algorithms. Both algorithms visited the same number of nodes on 646 unsatisfiable and 893 satisfiable instances. Also, the average time difference between the two algorithms on these instances was less than 0.1%. Thus, we focus our analysis on the remaining instances where there is a difference in the number of nodes visited between the two algorithms.

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

BTD and WitnessBTD had different numbers of nodes visited on 89 unsatisfiable and 101 satisfiable instances. BTD visited fewer nodes on 53 satisfiable instances. WitnessBTD visited fewer nodes on 89 unsatisfiable and 49 satisfiable instances. These counts are summarized in Table 7.2. Note that WitnessBTD did not visit any more nodes than BTD on unsatisfiable instances. This result is consistent with our theoretical analysis in Section 7.3.

Table 7.2: Number of instances with fewer #NV.

	BTD	WitnessBTD
UNSAT (89)	0	**89**
SAT (101)	**53**	49
Total (190)	53	**138**

The average number of nodes visited on these instances by each algorithm is given in Table 7.3. WitnessBTD on average visited half the number of nodes visited by BTD on unsatisfiable instances. However, on satisfiable instances, the difference was insignificant. WitnessBTD visited fewer nodes, but this difference did not exceed one percent that of BTD on average. Yet, this small percentage amounts to an average saving of more than 50,000 node visits per instance.

Table 7.3: Average number of nodes visited.

	BTD	WitnessBTD
UNSAT	1,437,909.79	**734,983.25**
SAT	4,785,737.57	**4,735,136.28**

The difference in the number of nodes visited is reflected in the total time taken by each algorithm to complete on each instance. Table 7.4 shows that 22 unsatisfiable instances are solved faster using BTD compared to 80 instances solved faster using WitnessBTD. However, as we expect from the results of nodes visited, 71 satisfiable instances were solved faster using BTD and 38 instances solved faster using WitnessBTD. Fortunately, the magnitude of the time difference is to the advantage of

WitnessBTD. Although more satisfiable instances are solved faster using BTD, the average times given in Table 7.5 show that WitnessBTD is on average faster than BTD by 15% on unsatisfiable instances. However, WitnessBTD is insignificantly faster than BTD on satisfiable instances. We next analyze these results in more detail.

Table 7.4: Number of instances completed faster.

	BTD	WitnessBTD
UNSAT (89)	22	**80**
SAT (101)	**71**	38
Total (190)	93	**118**

Table 7.5: Average time in seconds.

	BTD	WitnessBTD
UNSAT	145.55	**123.86**
SAT	1,151.35	**1,148.46**

Figure 7.3 compares the running time of BTD to WitnessBTD. The time is in seconds, with the time of BTD on the x-axis and WitnessBTD on the y-axis. The points represent the 89 unsatisfiable and 101 satisfiable instances on which the two algorithms had different numbers of nodes visited. Most of the points are on the diagonal line, meaning both algorithms had the same time on those instances. A number of points are below the diagonal, meaning that WitnessBTD was faster. No points are observed above the diagonal, which implies that WitnessBTD is never slower that BTD.

7.4.2 Comparing R(∗,m)C to GAC for finding all solutions

We now study the advantages of R(∗,m)C (Chapter 3) with localization (Chapter 5) and bolstering (Chapter 6) for each of the BTD and WitnessBTD.

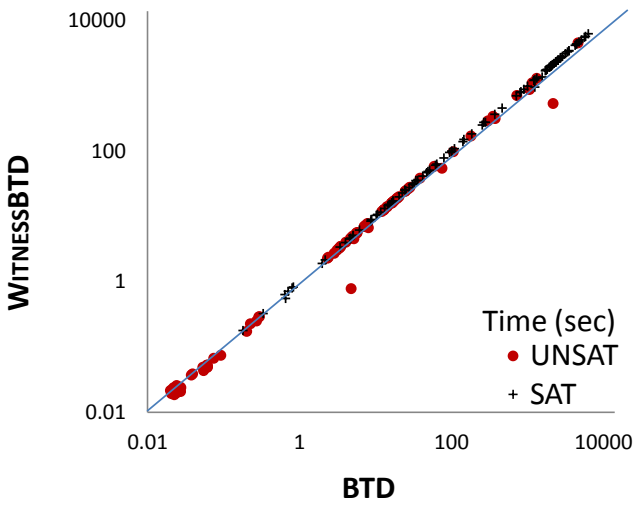

Figure 7.3: WitnessBTD and BTD time comparison.

7.4.2.1 Experimental set-up

We compare the advantages of enforcing the properties listed in Table 7.6 BTD and WitnessBTD. The consistency properties are enforced as full lookahead strategies in BTD and WitnessBTD, with domain/degree heuristic for dynamic variable ordering within the clusters. The benchmarks are selected from the CSP Solver Competition.[2] and are listed in Appendix E with their characteristics.

Table 7.6: Tested consistencies.

Type	$m = 2, 3, 4$	$m = \|\psi(cl_i)\|$
global	GAC	
global	wR(∗,m)C	
local	cl-wR(∗,m)C	cl-R(∗,$\|\psi(cl_i)\|$)C
projection	cl+proj-wR(∗,m)C	cl+proj-R(∗,$\|\psi(cl_i)\|$)C
binary	cl+bin-wR(∗,m)C	cl+bin-R(∗,$\|\psi(cl_i)\|$)C
clique	cl+clq-wR(∗,m)C	cl+clq-R(∗,$\|\psi(cl_i)\|$)C

In the selected benchmarks, 479 instances are unsatisfiable and 200 satisfiable. The maximum processing time per instance is set to two hours. The results are reported

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

for BTD in Table 7.7 and for WitnessBTD in Table 7.8. In both tables, the results are for each consistency algorithm, in terms of:

- Completed: the number of tested instances where the search counted the number of solutions within the allocated time.
- BT-free: the number of tested instances where the search counted the solutions in a backtrack-free manner.
- Min(#NV): the number of tested instances where the search visited the least number of nodes.
- Fastest: the number of tested instances that were solved fastest by the corresponding algorithm (within a precision of 256 msec).

7.4.2.2 Results

The results in Tables 7.7 and 7.8 show that higher levels of consistency with localization and bolstering outperform GAC when used with BTD and WitnessBTD for finding all solutions to the CSP. The results for the BTD (Table 7.7) and for WitnessBTD (Table 7.8) are quantitatively similar. Indeed, the benefits of WitnessBTD are significantly reduced because the high level of consistency eliminates the need of ensuring the global consistency of a partial solution (i.e., finding a witness).

On the criteria 'completed' and 'fastest', we distinguish between the SAT and UNSAT instances. On SAT instances, the localized properties with $m \geq 3$ and with and without bolstering outperform GAC (global) and wR($*$,m)C (global). cl-R($*$,$|\psi(cl_i)|$)C is the overall winner. On SAT instances, the localized properties with $m \geq 3$ and without bolstering complete on similar number of problems as GAC. However, GAC is the fastest (although the difference can be recovered by improving

our implementations). On the criteria Min(#NV) and BT-free, cl-R($*$,$|\psi(cl_i)|$)C with projection and binary bolstering is the winner on both SAT and UNSAT instances.

Table 7.7: Comparing consistency properties using BTD.

			wR(∗,2)C					wR(∗,3)C					wR(∗,4)C					R(∗,$\|\psi(cl_i)\|$)C			
	#Instances	GAC	global	local	projection	binary	clique	global	local	projection	binary	clique	global	local	projection	binary	clique	local	projection	binary	clique
Completed	UNSAT	202	190	199	190	190	175	193	248	238	236	221	193	240	236	234	226	**302**	291	286	277
	479	42.2%	39.7%	41.5%	39.7%	39.7%	36.5%	40.3%	51.8%	49.7%	49.3%	46.1%	40.3%	50.1%	49.3%	48.9%	47.2%	**63.0%**	60.8%	59.7%	57.8%
	SAT	**112**	92	110	92	89	71	87	111	100	96	81	81	109	96	92	80	108	90	89	78
	200	**56.0%**	46.0%	55.0%	46.0%	44.5%	35.5%	43.5%	55.5%	50.0%	48.0%	40.5%	40.5%	54.5%	48.0%	46.0%	40.0%	54.0%	45.0%	44.5%	39.0%
BT-free	UNSAT	0	70	39	70	70	74	97	104	139	139	131	140	103	141	141	148	186	**222**	**222**	213
	479	0.0%	14.6%	8.1%	14.6%	14.6%	15.4%	20.3%	21.7%	29.0%	29.0%	27.3%	29.2%	21.5%	29.4%	29.4%	30.9%	38.8%	**46.3%**	**46.3%**	44.5%
	SAT	15	25	16	25	25	29	42	17	47	47	45	52	21	60	55	50	25	**61**	**61**	52
	200	7.5%	12.5%	8.0%	12.5%	12.5%	14.5%	21.0%	8.5%	23.5%	23.5%	22.5%	26.0%	10.5%	30.0%	27.5%	25.0%	12.5%	**30.5%**	**30.5%**	26.0%
Min(#NV)	UNSAT	4	71	40	71	71	74	100	110	145	145	136	161	130	166	166	170	234	**264**	263	244
	479	0.8%	14.8%	8.4%	14.8%	14.8%	15.4%	20.9%	23.0%	30.3%	30.3%	28.4%	33.6%	27.1%	34.7%	34.7%	35.5%	48.9%	**55.1%**	54.9%	50.9%
	SAT	19	27	16	25	25	28	43	23	52	49	49	57	43	71	65	60	55	**74**	73	64
	200	9.5%	13.5%	8.0%	12.5%	12.5%	14.0%	21.5%	11.5%	26.0%	24.5%	24.5%	28.5%	21.5%	35.5%	32.5%	30.0%	27.5%	**37.0%**	36.5%	32.0%
Fastest	UNSAT	100	28	44	19	14	14	26	117	73	23	25	17	45	17	12	12	**187**	128	58	52
	479	20.9%	5.8%	9.2%	4.0%	2.9%	2.9%	5.4%	24.4%	15.2%	4.8%	5.2%	3.5%	9.4%	3.5%	2.5%	2.5%	**39.0%**	26.7%	12.1%	10.9%
	SAT	**73**	40	22	16	11	8	20	20	18	9	9	5	15	10	5	6	26	15	9	9
	200	**36.5%**	20.0%	11.0%	8.0%	5.5%	4.0%	10.0%	10.0%	9.0%	4.5%	4.5%	2.5%	7.5%	5.0%	2.5%	3.0%	13.0%	7.5%	4.5%	4.5%

Table 7.8: Comparing consistency properties using WitnessBTD.

			wR(∗,2)C					wR(∗,3)C					wR(∗,4)C					R(∗,|ψ(cl_i)|)C			
	#Instances	GAC	global	local	projection	binary	clique	global	local	projection	binary	clique	global	local	projection	binary	clique	local	projection	binary	clique
Completed	UNSAT	200	191	199	191	190	175	192	248	237	236	220	196	241	234	234	225	**302**	290	286	277
	479	41.8%	39.9%	41.5%	39.9%	39.7%	36.5%	40.1%	51.8%	49.5%	49.3%	45.9%	40.9%	50.3%	48.9%	48.9%	47.0%	**63.0%**	60.5%	59.7%	57.8%
	SAT	111	94	109	96	91	73	85	**112**	100	96	81	81	109	96	92	79	110	90	88	78
	200	55.5%	47.0%	54.5%	48.0%	45.5%	36.5%	42.5%	**56.0%**	50.0%	48.0%	40.5%	40.5%	54.5%	48.0%	46.0%	39.5%	55.0%	45.0%	44.0%	39.0%
BT-free	UNSAT	0	70	39	70	70	74	97	104	139	139	131	140	103	141	141	148	186	221	**222**	212
	479	0.0%	14.6%	8.1%	14.6%	14.6%	15.4%	20.3%	21.7%	29.0%	29.0%	27.3%	29.2%	21.5%	29.4%	29.4%	30.9%	38.8%	46.1%	**46.3%**	44.3%
	SAT	15	25	16	25	25	29	42	17	47	47	45	52	21	60	55	50	25	**61**	**61**	52
	200	7.5%	12.5%	8.0%	12.5%	12.5%	14.5%	21.0%	8.5%	23.5%	23.5%	22.5%	26.0%	10.5%	30.0%	27.5%	25.0%	12.5%	**30.5%**	**30.5%**	26.0%
Min(#NV)	UNSAT	2	72	41	72	72	74	101	111	145	145	136	163	132	166	166	169	235	263	**264**	244
	479	0.4%	15.0%	8.6%	15.0%	15.0%	15.4%	21.1%	23.2%	30.3%	30.3%	28.4%	34.0%	27.6%	34.7%	34.7%	35.3%	49.1%	54.9%	**55.1%**	50.9%
	SAT	19	26	16	24	25	28	42	23	52	49	49	57	43	71	65	61	57	**74**	73	65
	200	9.5%	13.0%	8.0%	12.0%	12.5%	14.0%	21.0%	11.5%	26.0%	24.5%	24.5%	28.5%	21.5%	35.5%	32.5%	30.5%	28.5%	**37.0%**	36.5%	32.5%
Fastest	UNSAT	100	28	44	21	15	15	26	120	71	23	23	23	46	21	15	15	**189**	126	58	52
	479	20.9%	5.8%	9.2%	4.4%	3.1%	3.1%	5.4%	25.1%	14.8%	4.8%	4.8%	4.8%	9.6%	4.4%	3.1%	3.1%	**39.5%**	26.3%	12.1%	10.9%
	SAT	**73**	39	23	15	11	8	20	20	18	9	9	5	15	10	5	6	27	15	9	9
	200	**36.5%**	19.5%	11.5%	7.5%	5.5%	4.0%	10.0%	10.0%	9.0%	4.5%	4.5%	2.5%	7.5%	5.0%	2.5%	3.0%	13.5%	7.5%	4.5%	4.5%

7.5 Conclusions

The experimental results showed that the performance of WitnessBTD in terms of total time and nodes visited is very similar to that of BTD for most instances. However, in about ten percent of the completed instances, there was a difference in the number of nodes visited, and we focused our analysis on these instances. Our analysis showed that WitnessBTD is more effective on unsatisfiable instances than on satisfiable instances, and yielded on average 49% reduction in number of nodes visited on unsatisfiable instances compared to BTD. Consequently, WitnessBTD was faster than BTD by 15% on the unsatisfiable instances. The results also showed that although the benefits of WitnessBTD were not significant for satisfiable instances, there was no significant overhead in using WitnessBTD.

Therefore, WitnessBTD can be safely applied with the potential of significantly reducing the number of nodes visited without incurring significant increase in the cost. The benefit of using WitnessBTD can be more valuable in situations where the constraint checks are expensive, and thus the reduction in the number of nodes visited can be even more significant.

The results on difficult CSPs showed that cl-R($*$,m)C outperforms GAC when used with BTD and WitnessBTD. Therefore, the advantages of higher consistency levels demonstrated in Chapters 5 and 6 also apply when these properties are used as full lookahead strategies in BTD and WitnessBTD.

Summary

In this chapter, we proposed a new algorithm WitnessBTD to improve the performance of BTD. WitnessBTD avoids the counting of solutions in a subtree that

cannot be extended to a global solution to a CSP. In our results, the behavior of WitnessBTD measured in number of nodes visited differed from BTD in only 10% of the computed CSPs. In those instances, WitnessBTD was most effective on unsatisfiable instances. It yielded on average 49% reduction is number of nodes visited, and consequently was faster than BTD by 15%. Although the benefits of WitnessBTD were not significant for satisfiable instances, there was no significant overhead in using WitnessBTD. Therefore, it can be safely used because it does not have a significant overhead, and can be faster on certain instances. Moreover, we experimentally showed that cl-R($*$,m)C outperforms GAC on difficult CSP when used with BTD and WitnessBTD.

Chapter 8

Conclusion

Below, we reflect on our approach and draw directions for future research.

8.1 Conclusions

The research presented in this thesis addresses the question of achieving practical tractability for solving CSPs. CSPs are in general $\mathcal{N}P$-complete, and are usually solved with search. In order to reduce the size of the search space, usually backtrack search is interleaved with constraint propagation. Linking the level of consistency satisfied by a CSP to the width of its constraint graph provides a sufficient condition for a backtrack-free search. Although this condition is appealing in theory, its usefulness in practice is limited because of its prohibitive space requirement.

We introduced a new parameterized relational consistency property, R($*$,m)C, and two algorithms for implementing it. We identified problem parameters and used them to construct a decision tree for dynamically selecting the appropriate algorithm for enforcing R($*$,m)C. Further, we adapted R($*$,m)C to a tree decomposition of the CSP by localizing the application of the algorithm to the clusters of the tree decomposition.

We proposed strategies for managing the propagation queue of a consistency algorithm in order to guide propagation along the structure of a tree decomposition of the problem. The strength of the consistency property, when localized to the clusters of a tree decomposition, depends on the messages communicated between the clusters. We proposed schemes for bolstering the separators of the clusters to further strengthen the enforced consistency. In addition, we proposed an improvement to the BTD algorithm for solution counting.

We characterized the proposed techniques and empirically evaluated their impact on difficult problems. Our results showed that, on difficult benchmark problems tested, the most effective technique is the one that enforces a minimal network on the clusters of a tree decomposition while bolstering propagation at the separators using projections of all existing constraints (i.e., cl+proj-R($*$,$|\psi(cl)|$)C). Indeed, using cl+proj-R($*$,$|\psi(cl)|$)C we were able to solve 424 out of 679 difficult instances, solving 300 of them without backtracking.

In this thesis, we explored new frontiers in higher level consistency, and established that achieving tractability in practice is both feasible and cost effective. We established new ways for advantageously exploiting the problem's structure in many aspects of CSP solving: consistency algorithms, ordering heuristics, and in solution counting.

8.2 Directions for Future Research

Below we identify a number of directions for future work.

1. *Extension to non-table constraints*: In this dissertation, we focused on constraints defined in extension. The consistency properties that we proposed here can also be enforced when the constraints are defined in intension. However, the enforced property may be weaker, because only values will be filtered from the

domains of the variables. Further research to study the generation of partial table-constraints to strengthen the enforced property when used with intensional constraints is an interesting direction for future work.

2. *Alternative criteria for propagation-queue management*: We showed the benefits of managing the propagation queue of consistency algorithms along a tree embedding of the CSP. We believe it may be worthwhile to investigate alternative criteria for selecting the cluster to remove from the fringe similarly to the plethora of variable ordering heuristics explored for backtrack search.

3. *Automating the selection of consistency algorithms*: Our hybrid solver that uses a decision tree to select one of the two algorithms for computing a minimal network (i.e., PerTuple and AllSol) performs well when applied to a *single* problem instance. However, when used, in sequence, on each of the clusters on a tree decomposition, the overall performance degraded, more because of incorrect selections than because of the overhead of computing the parameters. We need to investigate a more robust decision trees that performs competitively in the context of the tree decomposition.

   Another avenue is to design strategies that use the decision tree in the following context: If we can predict that a cluster is too expensive to solve, we can delay processing it, process other clusters, and then propagate, to the difficult cluster, the effects of filtering its neighbors. The propagation may likely simplify the difficult problem to the extent that it can be solved by one of our algorithms.

4. *Automatic selection of consistency property*: Finally, our research uncovers both the opportunity and the need to dynamically and locally select the appropriate consistency levels to enforce on a problem depending on the characteristics of each cluster and its difficulty. This observation opens the door to a 'fine

grain/cluster level' portfolio-based methods for consistency selection [Xu *et al.*, 2008]. The hybrid solver presented in this dissertation (which selects between PerTuple and AllSol) is a step in this direction. The approach can be extended to include the choice of the appropriate consistency property to enforce [Stergiou, 2009]. Coordination of such 'context-sensitive' methods with the queue-management strategies will allow us to target a cluster with the 'right' property at the 'right' time.

5. *Modify the structure of a tree decomposition*: We experimentally showed that bolstering can be an effective method to approach practical tractability. We also noticed that, sometimes, too much bolstering may be detrimental. We suspect that the large amount of overlap between the clusters is critical for choosing the appropriate level of bolstering. One direction to overcame this situation is to pre-process a CSP and decide whether clusters need to be merged or not, similar to the work by Fattah and Dechter [1996], and locally choose the right bolstering scheme based on the amount of the overlap at each separator.

6. *Characterize the performance of our techniques on randomly generated problems*: We focused our evaluations of difficult benchmark problems. Characterizing the effectiveness of our approach in terms of the structural properties of the constraint network (e.g., treewidth in relation to the size of the separators) and properties and types of constraints (e.g., arity and tightness) would increase our understanding of those techniques and of their applicability. To this end, it may be useful to write a generator of random structured problems with controllable parameters [Hogg and Dyer, 1996; Jégou and Terrioux, 2003].

Although the opportunities for further investigations may seem endless, this thesis has successfully and positively answered our original concern about implementing

tractability in practice: It provided a methodology and an arsenal of techniques to erode the difficulty of solving CSPs in practice.

Bibliography

[Arnborg *et al.*, 1987] Stefan A. Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of Finding Embeddings in a K-Tree. *SIAM Journal on Algebraic Discrete Methods*, 8:277–284, April 1987.

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

[Arndt, 2010] Jörg Arndt. *Matters Computational: Ideas, Algorithms, Source Code*, chapter Compositions. Academic Press, London, UK, 2010.

[Beeri *et al.*, 1983] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the Desirability of Acyclic Database Schemes. *Journal of the ACM*, 30(3):479–513, 1983.

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

[Bessiere *et al.*, 2008] Christian Bessiere, Kostas Stergiou, and Toby Walsh. Domain Filtering Consistencies for Non-Binary Constraints. *Artificial Intelligence*, 172(6-7):800–822, 2008.

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

[Bollobás *et al.*, 2003] Béla Bollobás, Christian Borgs, Jennifer Chayes, and Oliver Riordan. Directed Scale-Free Graphs. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 132–139. Society for Industrial and Applied Mathematics, 2003.

[Breiman, 2001] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

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

[Cohen *et al.*, 2008] David A. Cohen, Peter Jeavons, and Marc Gyssens. A unified theory of structural tractability for constraint satisfaction problems. *Journal of Computer and System Sciences*, 74(5):721–743, 2008.

[Debruyne and Bessiere, 1997] Romuald Debruyne and Christian Bessiere. Some Practical Filtering Techniques for the Constraint Satisfaction Problem. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI 1997)*, pages 418–423, 1997.

[Debruyne, 1999] Romuald Debruyne. A Strong Local Consistency for Constraint Satisfaction. In *Proceedings of the 11th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 1999)*, pages 202–209, 1999.

[Dechter and Dechter, 1987] Avi Dechter and Rina Dechter. Removing Redundancies in Constraint Networks. In *Proceedings of the sixth AAAI Conference on Artificial Intelligence (AAAI 1987)*, pages 105–109, 1987.

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

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

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

[Dechter and Rish, 1994] Rina Dechter and Irina Rish. Directional Resolution: The Davis-Putnam Procedure, Revisited. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*, pages 134–145, 1994.

[Dechter and Rish, 2003] Rina Dechter and Irina Rish. Mini-Buckets: A General Scheme for Bounded Inference. *Journal of the ACM*, 50(2):107–153, 2003.

[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 *et al.*, 1991] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal Constraint Networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.

[Dechter *et al.*, 2001] Rina Dechter, Kalev Kask, and Javier Larrosa. A General Scheme for Multiple Lower Bound Computation in Constraint Optimization. In *Proceedings of the Seventh International Conference on Principle and Practice of Constraint Programming (CP 2001)*, volume 2239 of *LNCS*, pages 346–360, 2001.

[Dechter, 1996] Rina Dechter. Bucket Elimination: A Unifying Framework for Probabilistic Inference Algorithms. In *Proceedings of the 12th Conference on Uncertainty in AI (UAI 1996)*, pages 211–219, 1996.

[Dechter, 1997] Rina Dechter. Mini-Buckets: A General Scheme of Generating Approximations in Automated Reasoning. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI 1997)*, pages 1297–1302, 1997.

[Dechter, 1999] Rina Dechter. Bucket Elimination: A Unifying Framework for Reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.

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

[Dermaku *et al.*, 2008] Artan Dermaku, Tobias Ganzow, Georg Gottlob, Ben Mcmahan, Nysret Musliu, and Marko Samer. Heuristic Methods for Hypertree Decomposition. In *Proceedings of the 7th Mexican International Conference on Artificial Intelligence (MICAI 2008)*, pages 1–11, 2008.

[Fagin, 1983] Ronald Fagin. Degrees of Acyclicity for Hypergraphs and Relational Database Schemes. *Journal of the ACM*, 30(3):514–550, 1983.

[Fattah and Dechter, 1996] Yousri El Fattah and Rina Dechter. An Evaluation of Structural Parameters for Probabilistic Reasoning: Results on Benchmark Circuits. In *Proceedings of the 12th Conference on Uncertainty in AI (UAI 1996)*, pages 244–251, 1996.

[Favier *et al.*, 2009] Aurélie Favier, Simon de Givry, and Philippe Jégou. Exploiting Problem Structure for Solution Counting. In *Proceedings of the 15th International Conference on Principle and Practice of Constraint Programming (CP 2009)*, volume 5732, pages 335–343, 2009.

[Fikes, 1970] Richard E. Fikes. REF-ARF: A System for Solving Problems Stated as Procedures. *Artificial Intelligence*, 1:27–120, 1970.

[Francis and Stuckey, 2007] Kathryn Francis and Peter J. Stuckey. Constraint Propagation for Loose Constraint Graphs. In *Proceedings of the 122nd ACM Symposium on Applied Computing (ACM SAC 2007)*, pages 334–335, 2007.

[Freuder and Wallace, 1992] Eugene C. Freuder and Richard J. Wallace. Partial Constraint Satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

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

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

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

[Garey and Johnson, 1979] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[Gent *et al.*, 1996] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, and Toby Walsh. The Constrainedness of Search. In *Proceedings of the 13th AAAI Conference on Artificial Intelligence (AAAI 1996)*, pages 246–252, 1996.

[Geschwender *et al.*, 2013] Daniel Geschwender, Shant Karakashian, Robert Woodward, Berthe Y. Choueiry, and Stephen D. Scott. Selecting the Appropriate Consistency Algorithm for CSPs Using Machine Learning Techniques. In *Pre-PhD Student Abstract and Poster Program, Proceedings of the 27th Conference on Artificial Intelligence (AAAI 2013)*, pages 1–2, 2013.

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

[Gottlob and Samer, 2009] Georg Gottlob and Marko Samer. A Backtracking-Based Algorithm for Hypertree Decomposition. *ACM Journal of Experimental Algorithmics*, 13:1–19, 2009.

[Gottlob and Scarcello, 2001] Georg Gottlob and Francesco Scarcello. Hypertree decompositions: A survey. In *Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science (MFCS 2001)*, pages 37–57, 2001.

[Gottlob and Szeider, 2008] Georg Gottlob and Stefan Szeider. Fixed-Parameter Algorithms For Artificial Intelligence, Constraint Satisfaction and Database Problems. *Compututer Journal*, 51(3):303–325, 2008.

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

[Gottlob *et al.*, 2000] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence*, 124(2):243–282, 2000.

[Gottlob *et al.*, 2002] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions and Tractable Queries. *Journal of Computer and System Sciences*, 64(3):579–627, 2002.

[Gottlob, 2011] Georg Gottlob. On Minimal Constraint Networks. In *Proceedings of the 17th International Conference on Principle and Practice of Constraint Programming (CP 2011)*, volume 6876 of *LNCS*, pages 325–339, 2011.

[Gyssens *et al.*, 1994] Marc Gyssens, Peter G. Jeavons, and David A. Cohen. Decomposing Constraint Satisfaction Problems Using Database Techniques. *Artificial Intelligence*, 66(1):57–89, 1994.

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

[Hall *et al.*, 2009] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explorations Newsletter*, 11:10–18, November 2009.

[Haralick and Elliott, 1980] Robert M. Haralick and Gordon L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14(3):263–313, 1980.

[Hogg and Dyer, 1996] Tadd Hogg and Martin E. Dyer. Refining the Phase Transitions in Combinatorial Search. *Artificial Intelligence*, 81 (1-2):127–154, 1996.

[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 *Proceedings of the IEEE Workshop on Tools for Artificial Intelligence*, pages 420–427, 1989.

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

[Jégou and Terrioux, 2003] Philippe Jégou and Cyril Terrioux. Hybrid Backtracking Bounded by Tree-Decomposition of Constraint Networks. *Artificial Intelligence*, 146(1):43–75, 2003.

[Jégou and Terrioux, 2010] Philippe Jégou and Cyril Terrioux. A New Filtering Based on Decomposition of Constraint Sub-Networks. In *Proceedings of the 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2010)*, pages 263–270, 2010.

[Jégou and Vilarem, 1993] Philippe Jégou and Marie-Catherine Vilarem. On Some Partial Line Graphs of a Hypergraph and the Associated Matroid. *Discrete Mathematics*, 111(1-3):333–344, 1993.

[Jégou, 1993] Philippe Jégou. On the Consistency of General Constraint-Satisfaction Problems. In *Proceedings of the 11th AAAI Conference on Artificial Intelligence (AAAI 1993)*, pages 114–119, 1993.

[Karakashian and Choueiry, 2010] Shant Karakashian and Berthe Y. Choueiry. Tree-Based Algorithms for Computing k-Combinations and k-Compositions. Technical Report TR-UNL-CSE-2010-0009, Constraint Systems Laboratory, University of Nebraska-Lincoln, Lincoln, NE, 2010.

[Karakashian *et al.*, 2010a] Shant Karakashian, Robert Woodward, Christopher Reeson, Berthe Y. Choueiry, and Christian Bessiere. A First Practical Algorithm for High Levels of Relational Consistency. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 10)*, pages 101–107, 2010.

[Karakashian *et al.*, 2010b] Shant Karakashian, Robert J. Woodward, Berthe Y. Choueiry, and Christian Bessiere. Relational Consistency by Constraint Filtering. In *Proceedings of the 25th ACM Symposium On Applied Computing (ACM SAC 2010)*, pages 2073–2074, 2010.

[Karakashian *et al.*, 2012] Shant Karakashian, Robert J. Woodward, Berthe Y. Choueiry, and Stephen D. Scott. Algorithms for the Minimal Network of a CSP and a Classifier for Choosing Between Them. Technical Report TR-UNL-CSE-2012-0007, Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE, 2012.

[Karakashian *et al.*, 2013] Shant Karakashian, Robert Woodward, and Berthe Y. Choueiry. Improving the Performance of Consistency Algorithms by Localizing and Bolstering Propagation in a Tree Decomposition. In *Proceedings of the 27th Conference on Artificial Intelligence (AAAI 2013)*, pages 1–8 (to appear), 2013.

[Kask *et al.*, 2005] Kalev Kask, Rina Dechter, Javier Larrosa, and Avi Dechter. Unifying Tree Decompositions for Reasoning in Graphical Models. *Artificial Intelligence*, 166(1-2):165–193, 2005.

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

[Laburhe, 2000] François Laburhe. CHOCO: Implementing a CP Kernel. In *CP Workshop on Techniques for Implementing Constraint Programming Systems (TRICS 2000)*, pages 71–85, 2000.

[Lagerkvist and Schulte, 2009] Mikael Z. Lagerkvist and Christian Schulte. Propagator Groups. In *Proceedings of the* 15th *International Conference on Principle*

and Practice of Constraint Programming (CP 2009), volume 5732 of *LNCS*, pages 524–538. Springer, 2009.

[Lecoutre and Hemery, 2007] Christophe Lecoutre and Fred Hemery. A Study of Residual Support in Arc Consistency. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 125–130, 2007.

[Lecoutre *et al.*, 2003] Christophe Lecoutre, Frédéric Boussemart, and Fred Hemery. Exploiting Multidirectionality in Coarse-Grained Arc Consistency Algorithms. In *Proceedings of the Ninth International Conference on Principle and Practice of Constraint Programming (CP 2003)*, volume 2833 of *LNCS*, pages 480–494. Springer, 2003.

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

[Lecoutre *et al.*, 2008] Christophe Lecoutre, Chavalit Likitvivatanavong, Scott G. Shannon, Roland H.C. Yap, and Yuanlin Zhang. Maintaining Arc Consistency with Multiple Residues. *Constraint Programming Letters*, 2:3–19, 2008.

[Lecoutre, 2009] Christophe Lecoutre. *Constraint Networks: Techniques and Algorithms*. ISTE Ltd & Wiley Press, 2009.

[Likitvivatanavong *et al.*, 2007] Chavalit Likitvivatanavong, Yuanlin Zhang, Scott Shannon, James Bowen, and Eugene C. Freuder. Arc Consistency During Search. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 137–142, 2007.

[Mackworth, 1977] Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[Marinescu and Dechter, 2007] Radu Marinescu and Rina Dechter. Best-First AND/OR Search for Graphical Models. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI 2007)*, pages 1171–1176, 2007.

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

[Paparrizou and Stergiou, 2012] Anastasia Paparrizou and Kostas Stergiou. An Efficient Higher-Order Consistency Algorithm for Table Constraints. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI 2012)*, 2012.

[Planken *et al.*, 2008] Léon Planken, Mathijs de Weerdt, and Roman van der Krogt. P3C: A New Algorithm for the Simple Temporal Problem. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS 2008)*, pages 256–263, 2008.

[Quinlan, 1993] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

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

[Robertson and Seymour, 1986] N. Robertson and P.D. Seymour. Graph Minors II: Algorithmic Aspects of Tree-Width. *Journal of Algorithms*, 7:309–322, 1986.

[Rollon and Dechter, 2010] Emma Rollon and Rina Dechter. New Mini-Bucket Partitioning Heuristics for Bounding the Probability of Evidence. In *Proceedings of*

the 24th AAAI Conference on Artificial Intelligence (AAAI 2010), pages 1199–1204, 2010.

[Rossi *et al.*, 2006] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming.* Elsevier, 2006.

[Ruskey, 2010] Frank Ruskey. Combinatorial Generation. Unpublished manuscript from Citeseer, 2010.

[Schulte and Stuckey, 2004] Christian Schulte and Peter J. Stuckey. Speeding up Constraint Propagation. In *Proceedings of the 10th International Conference on Principle and Practice of Constraint Programming (CP 2004)*, volume 3258 of *LNCS*, pages 619–633, 2004.

[Schulte and Stuckey, 2008] Christian Schulte and Peter J. Stuckey. Efficient Constraint Propagation Engines. *Transactions on Programming Languages and Systems*, 31(1), 2008.

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

[Stergiou and Walsh, 1999] Kostas Stergiou and Toby Walsh. Encodings of Non-Binary Constraint Satisfaction Problems. In *Proceedings of the 16th AAAI Conference on Artificial Intelligence (AAAI 1999)*, pages 163–168, 1999.

[Stergiou, 2009] Kostas Stergiou. Heuristics for Dynamically Adapting Propagation in Constraint Satisfaction Problems. *AI Communications*, 22(3):125–141, 2009.

[Sutherland, 1963] Ivan E. Sutherland. SKETCHPAD: A Man-Machine Graphical Communications System. Technical Report 296, Lincoln Laboratory, MIT, Cambridge, MA, 1963.

[Tsang, 1993] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993. Out of print, available `href="http://cswww.essex.ac.uk/CSP/papers/Tsang-Fcs1993.pdf"`.

[Wallace and Freuder, 1992] Richard J. Wallace and Eugene C. Freuder. Ordering Heuristics for Arc Consistency Algorithms. In *Proceedings of the Ninth Canadian Conference on Artificial Intelligence*, pages 163–169, 1992.

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

[Wilf, 1989] Herbert S. Wilf. *Combinatorial Algorithms: An Update*. SIAM CBMS-NSF 55, 1989.

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

[Zheng and Choueiry, 2005] Yaling Zheng and Berthe Y. Choueiry. New Structural Decomposition Techniques for Constraint Satisfaction Problems. In Boi Faltings et al., editor, *Recent Advances in Constraints*, volume 3419 of *Lecture Notes in Artificial Intelligence*, pages 113–127. Springer, 2005.

Appendix A

Computing All k-Connected Subgraphs

This appendix presents a fast algorithm for generating fixed-sized connected combinations of nodes in a graph.

A.1 Introduction

Identifying all connected subgraphs of a fixed size k of a graph G is a crucial step for enforcing relational consistency in Constraint Satisfaction Problems [Karakashian *et al.*, 2010a]. It is likely to arise in other settings that rely on analysis of graphs such as social networks. This combinatorial problem is computationally challenging in practice because the number of combinations of vertices of G of size k grows exponentially with the size of the graph. However, in sparse graphs, the number of *connected* such subgraphs is significantly smaller than the number of combinations. Thus, it is important to design an algorithm that enumerates only the connected combinations of vertices by exploiting the structure of the graph.

Here we propose, discuss, and evaluate ConSubg, an algorithm for this purpose. The two main features of our approach are the construction of a combination tree T and the definition of an operator $\otimes_t$. The combination tree T rooted at a vertex $v \in G$ has the property that the depth-first tree rooted at v of every G', where G' is a connected subgraph induced on G by at most k vertices including v, is isomorphic to a subgraph of T rooted at v. The operator $\otimes_t$ generates from T, without duplication, all connected subgraphs of G of size k including v. We evaluate it empirically on randomly generated graphs, scale-free graphs commonly used to model social networks, and graphs derived from constraint satisfaction problems. We use the simple example of Figure A.1 throughout this appendix to illustrate the operation of ConSubg. For example, the connected subgraphs of size $k = 4$ for the graph shown in Figure A.1 are:

$$\text{ConSubg}(k, G) = \{\{a, b, c, d\}, \{a, b, c, e\}, \{a, b, d, e\}, \{a, d, c, e\}\}. \tag{A.1}$$

Note that $\{b, c, d, e\}$ is not a connected subgraph of G and is thus excluded from the result.

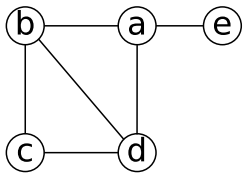

Figure A.1: Simple graph.

Definition 5 Given a graph $G = (V, E)$ and a constant k, ConSubg(k, G) returns all sets of V' vertices where $V' \subseteq V$, $|V'| = k$, and the subgraph G' of G induced by V' is connected. We call such sets of k vertices *k-ConnVertices*.

The graph in Figure A.1 represents the dual graph of a Constraint Satisfaction Problem (CSP) [Dechter, 2003]. A vertex in this graph represents a constraint, defined

as a relation on a set of variables, which is the scope of the constraint. An edge connects two vertices whose scopes overlap (i.e., share a variable). Enforcing the consistency properties R(i,k)C [Dechter and van Beek, 1997] and R($*$,k)C [Karakashian *et al.*, 2010a] on this CSP requires computing all connected subgraphs of size k.[1] Beyond our original motivation, we believe that an algorithm that implements such a functionality is useful in other contexts in Constraint Programming in particular and in Combinatorics in general.

This appendix is structured as follows. Section A.2 reviews alternative approaches. Section A.3 constitutes the bulk of this appendix: It discusses in great detail ConSubG and its various components, introducing data structures that we designed for this purpose and discussing the complexity, soundness and completeness of the constituent components of ConSubG. Section A.4 proposes to improve the performance of the algorithm by memoization. Section A.5 discusses the time and space complexity of ConSubG and Section A.6 its correctness. Section A.7 demonstrates the practical usefulness of our algorithm by comparing its performance on randomly generated graphs, scale-free networks, and constraint satisfaction problems. Finally, Section A.8 concludes this appendix.

A.2 Alternative Approaches

A straightforward algorithm for implementing ConSubG is to first generate all k combinations of the V vertices of the graph G, then to remove those combinations that are not connected subgraphs of G. A simple algorithm for generating such a combination consists of k nested loops, which we call the 'brute-force algorithm'

[1]The graph in Figure A.1 can also represent the constraint network of a binary CSP where a vertex represent a variable and an edge represent a binary constraint. The connected subgraphs of size k are useful for enforcing the consistency property k-consistency [Freuder, 1978].

and denote BF-ConSubg. BF-ConSubg (Algorithm 9) enumerates all possible combinations of k vertices storing only those that correspond to connected subgraphs.

Algorithm 9: BF-ConSubg(k, G)

```
   Input: k, G
   Output: A list of all k-ConnVertices of G
1  pos: a vector of the vertices of G;
2  list ← ∅;
3  for i_1 ← pos[1] to pos[s − k + 1] do
4      for i_2 ← pos[i_1] to pos[s − k + 2] do
5          for i_3 ← pos[i_2] to pos[s − k + 3] do
6              ...
7              for i_k ← pos[i_{k−1}] to pos[s] do
8                  if (i_1, i_2, ..., i_k) forms a connected subgraph of G then
9                      Push((i_1, i_2, i_3, ..., i_k), list)
10 return list
```

BF-ConSubg fails to exploit the connectivity of the graph: it may generate many subgraphs that are not connected and have to be discarded, which is wasteful of computing resources. In contrast, ConSubg exploits the connectivity of the graph and generates only connected subgraphs. At the risk of significantly oversimplifying it, ConSubg operates as follows:

1. It considers an arbitrary node in the graph as a 'root' node.
2. It restricts itself to the nodes of a distance k from this root node.
3. It generates all k-*ConnVertices* that include the root node.
4. It removes the root node from the graph.
5. Finally, it iteratively applies the above process to the remaining nodes of the graph.

The strength of ConSubG stems from the particular structures and processes implemented in the above mentioned Steps 2 and 3. In order to show that the effectiveness of our approach is not limited to the above 'decomposition' strategy but that we do exploit the topology of the graph in a much stronger sense, we modify the brute-force algorithm BF-ConSubG (Algorithm 9) to apply it in a localized manner similarly to the above-listed strategy, yielding LBF-ConSubG (Algorithm 10).

Algorithm 10: LBF-ConSubG(k, G).

Input: k, G
Output: A list of all k-*ConnVertices* of G

```
1 list ← ∅;
2 queue ← Vertices(G);
3 foreach v ∈ queue do
4 |   G' ← the subgraph of G induced by vertices within distance k from v;
5 |   list ← list ∪ BF-ConSubG(k, G');
6 |   Remove(v, G);
7 return list
```

While the worst-case complexity of all algorithms remains exponential in k (because the number of k-*ConnVertices* may be exponential in k), we conduct, in Section A.7, an extensive empirical evaluation to compare the performance of ConSubG, BF-ConSubG and LBF-ConSubG on various types of graphs, and empirically establish the advantages of ConSubG.

A.3 Description of the Algorithm

For the sake of clarity and readability and to facilitate the analysis, we decompose the presentation of our algorithm into components shown in Table A.1. After the presentation of each component of the algorithm, we illustrate its operation on the simple example of Figure A.1. When applicable, we also discuss the complexity,

Table A.1: A quick reference table to the proposed algorithms.

Algorithm	Pseudocode	Calls algorithm(s)	Section
ConSubg	Algorithm 11	CombinationsWithV	Section A.3.1
CombinationsWithV	Algorithm 12	CombinationTree CombinationsFromTree	Section A.3.1
CombinationTree	Algorithm 13	BuildTree	Section A.3.2
BuildTree	Algorithm 14	Self	Section A.3.2
CombinationsFromTree	Algorithm 15	Self k-combinations k-compositions	Section A.3.3

soundness, and completeness of the proposed component.

A.3.1 ConSubg and CombinationsWithV

ConSubg (Algorithm 11) takes as input an integer k and a graph G and returns all lists of k vertices inducing connected subgraphs of G. Starting from an arbitrary node, it calls CombinationsWithV (Algorithm 12) on a vertex of G to generate all k-*ConnVertices* that include that vertex. Then it removes the vertex from the graph and repeats the same operation on each of the remaining vertices in the graph.

Algorithm 11: ConSubg(k, G).

Input: k, G
Output: A list of all k-*ConnVertices* of G

```
1 list ← ∅;
2 queue ← Vertices(G);
3 foreach v ∈ queue do
4 |   list ← list ∪ CombinationsWithV(v, k, G);
5 |   Remove(v, G);
6 return list
```

CombinationsWithV (Algorithm 12) calls:

- CombinationTree (Algorithm 13), which builds a combination tree rooted at the vertex given as input, and

- CombinationsFromTree (Algorithm 15), which operates on the generated combination tree to compute the set of *k-ConnVertices*.

Algorithm 12: CombinationsWithV(v, k, G).

Input: v, k, G
Output: A list of all *k-ConnVertices* of G that include vertex v
1 *tree* ←CombinationTree(v, k, G);
2 *ncombs* ← CombinationsFromTree$(tree, k)$;
3 **return** Labels$(ncombs)$

Illustrating the execution of ConSubg and CombinationsWithV: Below we discuss the application of ConSubg (Algorithm 11) with $k = 4$ to the graph of Figure A.2. The queue is initialized in Line 2 to $\{a, b, c, d, e\}$, which is the list of vertices of the graph. Calling CombinationsWithV (Algorithm 12) with a and $k = 4$ on G returns the list of all sought *k-ConnVertices* that include the vertex a. Thus, a can be removed from G (Line 5) for all subsequent calls to CombinationsWithV. The process is repeated on the remaining vertices (i.e., b, c, d, and e).

CombinationsWithV receives as input a vertex, the combination size, and the graph. In Line 1, it generates a special tree structure, which we call *combination tree* and discuss in Section A.3.2. The algorithm uses the tree in Line 2 to collect the sought *k-ConnVertices* that include the vertex given as input. In the following sections, we describe how the tree is built for the graph in Figure A.2 with the selected node a and combination size 4.

A.3.2 Building the combination tree

In this section, we study the process of building a combination tree. We introduce the algorithms, illustrate their application to a simple example, discuss their complexity, and establish their soundness and completeness.

CombinationTree (Algorithm 13) calls BuildTree (Algorithm 14). Together, these two algorithms yield a tree structure that we call the *combination tree*. We refer to the vertices of the combination tree as "nodes" in order to distinguish them from the vertices of the graph. The combination tree, rooted at a node n, is of maximum depth k. The node n corresponds to the graph vertex given as input, and each node n_t in the tree corresponds to some vertex of G, denoted Vertex(n_t). Two or more nodes in the generated tree may correspond to the same vertex in G. Further, any two nodes that are connected in the tree correspond to two connected vertices in the graph G. Figure A.3 shows the tree generated by calling CombinationTree with

Algorithm 13: CombinationTree(v, k, G).

Input: v, k, G
Output: The root of a combination tree
1 $root \leftarrow$ a new tree node corresponding to v;
2 **for** $i \leftarrow 0$ *to* $(k-1)$ **do** $list[i] \leftarrow \emptyset$;
3 $list[0] \leftarrow \{v\}$;
4 BuildTree($root, 1, G, k$) ;
5 **return** $root$

the parameters a, $k = 4$, and the graph of Figure A.2.

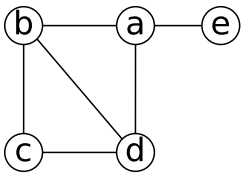

Figure A.2: Simple graph.

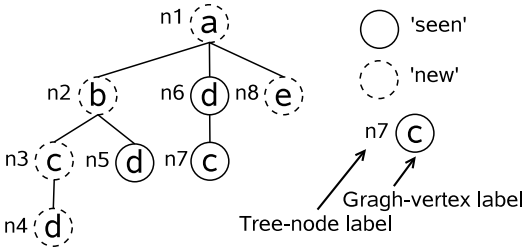

Figure A.3: Combination tree for a, $k = 4$, and Fig. A.2.

BuildTree proceeds in a depth-first manner. For each node n_t at depth l in the tree such that $l < k$, it adds, as children to n_t, all nodes n'_t that satisfy the following two conditions:

Condition 1: $\textsc{Vertex}(n'_t) \in \textsc{Neighbors}(\textsc{Vertex}(n_t))$.

Condition 2: The vertex of n'_t is not the vertex of an ancestor, sibling, or a sibling of any ancestor of n_t.

Notably, BuildTree may visit a given vertex of the graph more than once, which occurs when the vertex can be reached through an alternative path from the root. The goal of Condition 2 is to:

1. Limit the size of the generated tree by pruning subtrees as argued in Proposition 6, and

2. Guarantee the existence of a subtree elsewhere in the combination tree that contains the vertices of the pruned subtree.

Indeed, Condition 2 above yields the following two propositions:

Proposition 1 *No two siblings of a tree node in the combination tree correspond to the same vertex of the graph.*

Proof: Follows directly from Condition 2.

Proposition 2 *The maximum branching factor of the combination tree is bounded.*

Proof: Given that the number of vertices in the graph is bounded and given Proposition 1, each tree node has a bounded number of children. □

In order to generate a tree that satisfies the two above-listed conditions, each node n_t in the tree maintains:

1. A list of the vertices of the ancestors of n_t in the tree, and

2. A list of the vertices of the siblings of the ancestors of n_t generated *before* the node n_t itself was generated.

A child for n_t is generated only when the corresponding vertex does not appear in the list of n_t. When the condition is not met, we say that the subtree rooted at this child is *omitted*.[2] When adding n'_t to the tree, the following operations are performed in sequence:

1. The vertex corresponding to n'_t is added to the list of n_t.
2. The list of n'_t is a copy of the list of n_t.

The pseudocode of BuildTree (Algorithm 14) uses two marking functions: MarkV for graph vertices and MarkN for tree nodes:

1. MarkV is used to mark a vertex of the graph as 'visited.' We assume that all graph vertices are initially marked as 'unvisited.'
2. MarkN is used to mark a node in the tree as 'new,' thus indicating that the corresponding graph vertex has not yet been encountered. Otherwise, the tree node is marked as 'seen' indicating that there already exists, in the tree, another node corresponding to the same graph vertex.

A.3.2.1 Illustrating the execution of CombinationTree

Below we illustrate the generation of the tree shown in Figure A.5, obtained by applying CombinationTree (Algorithm 13) on the vertex a, $k = 4$, and the graph of Figure A.4. Line 1 of Algorithm 13 generates the root of the tree, $n1$, to correspond to the vertex a. Lines 2 and 3 initialize the vector array $list[]$. Line 4 calls BuildTree (Algorithm 14) with the two parameters $n1$ and 1 (for the tree depth) to build the children of the root.

[2]This terminology is used in several of the proofs below.

Algorithm 14: BUILDTREE$(n_t, depth, G, k)$

Input: $n_t, depth, G, k$

```
1  list[depth] ← list[depth − 1];
2  foreach v′ ∈ NEIGHBORS(VERTEX(n_t)) do
3      if v′ ∉ list[depth] then
4          add n′_t as a child to n_t with VERTEX(n′_t) = v′;
5          list[depth] ← list[depth] ∪ {v′};
6          if MARKV(v′) ≠ visited then
7              MARKN(n′_t) ← new;
8              MARKV(v′) ← visited;
9          else
10             MARKN(n′_t) ← seen;
11         if depth + 1 ≤ k then BUILDTREE(n′_t, depth + 1, G, k)
```

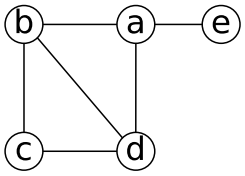

Figure A.4: Simple example.

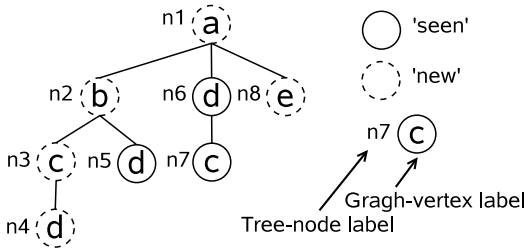

Figure A.5: Combination tree rooted at vertex a with $k = 4$ for the graph in Figure A.4.

In Line 1 of Algorithm 14, the list of 'ancestors' is copied from that of the parent. Thus, we have $list[1] = \{a\}$. Then, the subtrees corresponding to each of the neighbors of a (i.e., b, d and e) are built, see Figure A.5.

First the vertex b is considered. Because $b \notin list[1]=\{a\}$, a node $n2$ corresponding vertex to b is added as a child to the root. The vertex b is added to $list[1]$ (i.e., $list[1]=\{a,b\}$) for the sake of the descendants of $n2$. $n2$ is marked as 'new' because b was not visited before. The vertex b is marked as 'visited.' Then, Line 11 calls Algorithm 14 recursively to generate the children of the node $n2$ corresponding to vertex b.

In the new recursive call to Algorithm 14, the set of ancestors at depth 2 is set

to $\{a, b\}$ (i.e., $list[2]$={a,b}). Vertices a, c, and d are adjacent to vertex b. Because $a \in list[2]$, it is skipped. The node $n3$ is created for vertex c and added as a child of node $n2$. Then the node $n3$ and the vertex c are appropriately marked as 'new' and 'visited,' respectively. Now, $list[2] = \{a, b, c\}$. The recursive call generates a child $n4$ for $n3$, where $n4$ corresponds to vertex d.

At this point, we have $depth = 3$. The condition in Line 11 is not satisfied, which ends the recursion. Back to node $n2$ at the previous level in the recursion, the second neighbor d of b is considered. The list of ancestors is $list[2] = \{a, b, c\}$ and $d \notin list[2]$. Therefore, a tree node $n5$ corresponding to the vertex d is added as a child of $n2$. The list of ancestors at this level $list[2]$ is updated to $\{a, b, c, d\}$. Because vertex d was visited in a previous recursive call, the node $n5$ is marked as 'seen.' Similarly the rest of the nodes are added to the tree resulting in the tree shown in Figure A.5.

A.3.2.2 Complexity of CombinationTree and BuildTree

We make the following observations about the combination tree. The depth of the generated tree is $(k - 1)$.

If the maximum degree of the graph is d, the size of the list at $depth$=1 can be at most $2d$, and the size of the list at $depth$=$(k - 1)$ inheriting from the ancestors is bounded by $\mathcal{O}(d \cdot k)$.

Because Algorithms 13 and 14 proceed in a depth-first manner, only the lists along the current path are stored. Thus, the space complexity of the lists is $\mathcal{O}(d \cdot k^2)$. These lists are stored in an $1 \times k$ array indexed by the depth of the node in the tree.

Proposition 3 (Complexity of CombinationTree and BuildTree.) *The number of nodes in the tree is $\mathcal{O}(d^{(k-1)})$ assuming that the maximum degree of G is d. Thus, the time and space complexity of* CombinationTree *and* BuildTree *is $\mathcal{O}(d^{(k-1)})$.*

A.3.2.3 Soundness and completeness of combination trees

Below, we prove that:

1. The combination trees generated by ConSubg partition the set of all *k-ConnVertices* of the graph.
2. BuildTree terminates.
3. All connected subgraphs of size k including a given vertex are 'represented' in the combination tree built for this vertex.

Proposition 4 (Partitioning of combinations) *No k-ConnVertices set can be extracted from two different combination trees generated by Algorithm 13.*

Proof: Every *k-ConnVertices* set extracted by CombinationsFromTree from the combination tree includes the vertex of the root of the tree. Moreover, once a combination tree has been processed, the vertex of the root is removed from the graph. Hence, the same combination cannot be extracted from subsequent combination trees. □

Proposition 5 *Let T be the combination tree generated by applying* CombinationTree *on v and G. For every connected subgraph G' induced on G by at most k vertices including v, the depth-first tree of G' rooted at v is isomorphic to a subgraph of T rooted at v. Moreover, every node in T is necessary for this property to hold.*

Proof: Let T be the combination tree rooted at v resulting from applying BuildTree on G, and let G' be an induced connected subgraph of G of at most k vertices including v. Let T' a depth-first traversal of G' rooted at v. We prove that T' is isomorphic to a subgraph of T. Because T visits G in a depth-first manner without skipping already visited vertices except those violating Condition 2, a subgraph

isomorphic to T' exists in T unless pruned by Condition 2. We next show that even after the application of Condition 2, there exists in T a subgraph T'' of T that is isomorphic to T'.

Consider a node n_p of T such that (1) $\textsc{Vertex}(n_p) \in G'$, (2) the vertices of the ancestors of n_p in T are in G', and (3) n_p is pruned by Condition 2. We show that the path from the root of T to n_p cannot be isomorphic to a path in T', but that there exists a path in T from the root to a node n'_p such that $\textsc{Vertex}(n_p)=\textsc{Vertex}(n'_p)$ that is isomorphic to a path in T'. Because n_p is pruned by Condition 2, then a node n'_p where $\textsc{Vertex}(n_p)=\textsc{Vertex}(n'_p)$ must exist in T where the ancestors of n'_p are all in G' (by Condition 2). Consequently, there are two paths p and p' in T where (1) p is the path from the root of T to n_p, (2) p' is the path from the root to n'_p, and (3) the vertices of the nodes in p and p' are all in G'. Thus, there must exist two paths in G' from v to v' that are isomorphic to p and p'. Further, only one of those two paths in G' appears in T', which is our depth-first traversal of G'. A path isomorphic to p cannot appear in T' because of the canonical ordering of the vertices is used to build the trees. Thus, there exists a path in T' that is isomorphic to p', and p' must be isomorphic to a path in T'. As a conclusion, the pruning by Condition 2 will maintain in T a tree isomorphic to T'.

Now, we prove that every node in T is necessary for the above property to hold. Consider a node $n \in T$, and let p be the path in T from the root to n. Let G' be the subgraph in G induced by the vertices of the nodes in p. Given the canonical ordering of the graph vertices, p is isomorphic to the depth-first tree of G'. Because no two siblings in T have the same vertex label, p is the only subgraph of T isomorphic to the depth-first tree of G'. Therefore, if n was removed from T, there will not be a subgraph of T that is isomorphic to the depth-first tree of G', and the above property will be lost. □

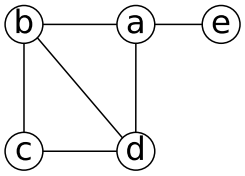

Figure A.6: Simple example.

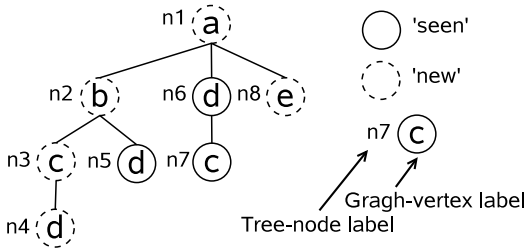

Figure A.7: The tree rooted at vertex a for $k = 4$ for the graph in Figure A.6.

Proposition 6 CombinationTree *terminates.*

Proof: BuildTree (Algorithm 14) traverses the graph in a depth-first manner without skipping already visited vertices. Thus, the termination of BuildTree is a legitimate concern. The algorithm stops proceeding down a path under two conditions:

1. The condition in Line 11, which guarantees that the length of the 'current' path is always smaller than or equal to k, e.g. node $n4$ in Figure A.7. Thus, the depth of the tree generated by Algorithm 14 is never larger than k.

2. The condition in Line 3 fails, which enforces Condition 2 of Section A.3.2. Proposition 2 guarantees that the branching factor of the tree generated by Algorithm 14 is bounded.

Consequently, the size of the tree generated by BuildTree is bounded, and BuildTree terminates. □

A.3.3 Extracting k-*ConnVertices* from a combination tree

CombinationsFromTree (Algorithm 15) is recursive and calls itself at Line 11. It also calls the functions k-combinations and k-compositions, and uses a new set operator $\otimes_t$.

- k-COMBINATIONS(i,s) generates all combinations of size i of the elements of a set s. We assume that each element in the generated set is ordered. For example,

$$k\text{-COMBINATIONS}(2, \{n2, n6, n8\}) = \{\{\text{n2,n6}\},\{\text{n2,n8}\},\{\text{n6,n8}\}\}.$$

  BF-CONSUBG (after removing Line 8) is an obvious implementation for k-COMBINATIONS. Other implementations are reported in [Ruskey, 2010; Arndt, 2010]. Ours is described in [Karakashian and Choueiry, 2010].

- k-COMPOSITIONS generates all strings of length $size$ on the integer interval [1,($Sum - size + 1$)] such that the sum of the elements of a string is equal to Sum. For example,

$$k\text{-COMPOSITIONS}(3, 4) = \{\{1, 1, 2\}, \{1, 2, 1\}, \{2, 1, 1\}\}.$$

  Because every element in the generated set is a string, the element is considered to be ordered. A recursive algorithm for k-COMPOSITIONS is attributed to Knuth [Wilf, 1989]. Implementations are reported in [Ruskey, 2010; Arndt, 2010]. Our implementation is tree based and described in [Karakashian and Choueiry, 2010].

- The binary operator $\otimes_t$ operates on sets of sets and is discussed in Section A.3.3.1.

Below, we formally define and analyze the operator $\otimes_t$, provide the pseudocode of A.3.3, illustrate its execution on our running example, and discuss the implementation of the operator $\otimes_t$.

A.3.3.1 Defining of the $\otimes_t$ operator

We introduce the following definition for an operator that operates on two sets:

Definition 6 (UnionProduct) We define the binary operator UnionProduct, denoted $\otimes$, as the operator that combines two sets of sets as follows:

$$S_1 \otimes S_2 = \{\ x \mid (x = s_1 \cup s_2) \wedge (s_1 \in S_1) \wedge (s_2 \in S_2)\ \} \tag{A.2}$$

UnionProduct is a cross-product-like operator in which two elements are combined by union instead of forming the usual tuple.

We refine the UnionProduct operator into a binary operator denoted $\otimes_t$, which we use in CombinationsFromTree (Line 14 of Algorithm 15). $\otimes_t$ operates on two sets of nodes from a combination tree as follows:

$$S_1 \otimes_t S_2 = \begin{cases} \emptyset, \textbf{ if } S_1 = \emptyset \\ S_1, \textbf{ if } S_2 = \emptyset \\ \{x \mid (x = s_1 \cup s_2) \wedge (s_1 \in S_1) \wedge (s_2 \in S_2) \\ \quad \wedge (\forall i \in s_1, j \in s_2, \textsc{Vertex}(i) \neq \textsc{Vertex}(j)) \\ \quad \wedge ((\exists j \in s_2\, \textsc{MarkN}(j) = \text{`new'}) \vee (\forall i \in s_1, j \in s_2, l \in \textsc{Children}(i), \\ \qquad\qquad \textsc{Vertex}(j) \neq \textsc{Vertex}(l)))\}, \textbf{ otherwise}. \end{cases} \tag{A.3}$$

Let us explain the meaning of the two conditions in Expressions (A.3). The first condition is:

$$\forall i \in s_1, j \in s_2, \textsc{Vertex}(i) \neq \textsc{Vertex}(j). \tag{A.4}$$

This condition guarantees that no two nodes in an element of $S_1 \otimes_t S_2$ correspond to the same graph vertex. The goal is to guarantee that every element of $S_1 \otimes_t S_2$

has only nodes corresponding to distinct graph vertices. The second condition is the disjunction of the two following conditions:

$$\exists j \in s_2 \, \textsc{MarkN}(j) = \text{`}new\text{'} \tag{A.5}$$

$$\forall i \in s_1, j \in s_2, l \in \textsc{Children}(i), \textsc{Vertex}(j) \neq \textsc{Vertex}(l). \tag{A.6}$$

The condition in Expression (A.5) guarantees that an element is added to $S_1 \otimes_t S_2$ when at least one of the tree nodes in s_2 is 'new,' that is, it corresponds to a vertex that had not been encountered before. The condition in Expression (A.5) is thus to ensure that elements not encountered before are included in $S_1 \otimes_t S_2$.

The intuition behind the condition in Expression (A.6) is as follows. When a tree node $j \in s_2$ corresponds to the same vertex as a child of a tree node $i \in s_1$, then the set of vertex labels obtained from the subtree rooted at j can also be obtained from the subtree rooted at i and from subtrees rooted at siblings, parents, and siblings of parents of i. Hence, $s_1 \cup s_2$ is omitted from $S_1 \otimes_t S_2$.

Note that, while the operator $\otimes$ is commutative, the operator $\otimes_t$, by definition, is associative but not commutative.

Proposition 7 (Time complexity of $S_1 \otimes_t S_2$.) *The time complexity of $S_1 \otimes_t S_2$ is $\mathcal{O}\left(|S_1| \cdot |S_2| \cdot |s_1| \cdot |s_2|\right)$, where $|s_1|$ and $|s_2|$ are the sizes of the largest elements of $|S_1|$ and $|S_2|$ respectively.*

A.3.3.2 Pseudocode of CombinationsFromTree

CombinationsFromTree (Algorithm 15) takes as parameters a combination tree and a combination size k. It returns combinations of nodes of the tree that:

1. Include the root of the tree and

2. Correspond to the connected subgraphs of size k of the original graph.

Algorithm 15: CombinationsFromTree($tree, k$)

Input: $tree, k$
Output: A list of sets of nodes of the tree including the root node

```
1  t ← root_tree;
2  lnodesets ← ∅;
3  if k = 1 then return {t};
4  for i ← 1 to Min(|Children(t)|, (k − 1)) do
5      foreach NodeComb ∈ k-combinations(i, Children(t)) do
6          foreach string ∈ k-compositions(i, (k − 1)) do
7              fail ← false;
8              for pos ← 1 to i do
9                  stRoot ← element in position pos in NodeComb;
10                 size ← element in position pos in string;
11                 S[pos] ← CombinationsFromTree(stRoot, size);
12                 if S[pos] = ∅ then fail ← true; break
13             if fail then continue;
14             foreach combProduct in S[1] ⊗_t ··· ⊗_t S[i] do
15                 lnodesets ← lnodesets ∪ {combProduct ∪ {t}};
16 return lnodesets
```

A.3.3.3 Illustrating the execution of CombinationsFromTree

Consider the graph in Figure A.8 and its corresponding combination tree shown in Figure A.9. The tree is passed to Algorithm 15 with $k = 4$, yielding:

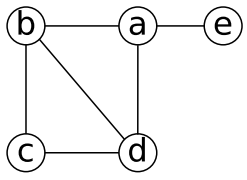

Figure A.8: Simple graph.

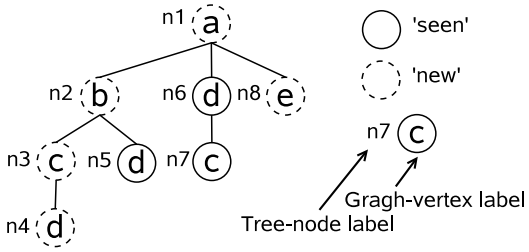

Figure A.9: The tree rooted at vertex a for $k = 4$ for the graph in Figure A.8.

$$\{\{n1, n2, n3, n4\}, \{n1, n2, n3, n8\}, \{n1, n2, n5, n8\}, \{n1, n6, n7, n8\}\}, \tag{A.7}$$

which is mapped in a straightforward manner to yield the following combinations of vertices:

$$\{\{a, b, c, d\}, \{a, b, c, e\}, \{a, b, d, e\}, \{a, d, c, e\}\}. \tag{A.8}$$

Below, we explain step by step how CombinationsFromTree reaches the result in Expression (A.7). The call to CombinationsFromTree($n1$,4) yields three iterations for i=1, 2, and 3 in Line 4. Thus, in Line 5, *NodeComb* iterates over the elements of the following sets:

- For $i = 1$, k-combinations(1,$\{n2, n6, n8\}$)= $\{\{n2\}, \{n6\}, \{n8\}\}$,
- For $i = 2$, k-combinations(2,$\{n2, n6, n8\}$)= $\{\{n2, n6\}, \{n2, n8\}, \{n6, n8\}\}$, and
- For $i = 3$, k-combinations(3,$\{n2, n6, n8\}$)= $\{\{n2, n6, n8\}\}$.

At Line 6, *string* iterates over the elements of the following sets:

- For $i = 1$, k-compositions(1,3)=$\{\{3\}\}$,
- For $i = 2$, k-compositions(2,3)=$\{\{1,2\},\{2,1\}\}$, and
- For $i = 3$, k-compositions(3,3)=$\{\{1,1,1\}\}$.

In order to continue our illustration of the operation of CombinationsFromTree, we introduce the following definition:

Definition 7 (Configuration) We define a *configuration* to be a set of 3-tuple $\langle j, N_j, C_j \rangle$ where:

1. j is a positive integer denoting the number of subtrees of the combinations tree to consider,

2. N_j is an ordered set of size j ($|N_j| = j$) of tree nodes that have the same parents in the combination tree, and

3. C_j is an ordered set of positive integers ($|C_j| = j$). Each integer in C_j specifies the size of the combination of tree nodes to be extracted from the subtree rooted at the node at the same position in N_j.

Examples of configurations in Figure A.9 are

$$\langle 1, \{n2\}, \{3\}\rangle, \langle 2, \{n2, n6\}, \{1, 2\}\rangle, \langle 3, \{n2, n6, n8\}, \{1, 1, 1\}\rangle. \tag{A.9}$$

The three nested loops from Line 4 to Line 15 generate all configurations for the children of a given root (Lines 4, 5, and 6), generate the combinations of tree nodes from each configuration (Line 11), then combine the resulting combinations within each configuration (Line 14). Below, we illustrate this process for i=1, 2, and 3.

For $i = 1$**,** $NodeComb \in \{\{\text{n2}\}, \{\text{n6}\}, \{\text{n8}\}\}$, and $string \in \{\{3\}\}$. We have three configurations at this point:

$$\langle 1, \{n2\}, \{3\}\rangle, \langle 1, \{n6\}, \{3\}\rangle, \langle 1, \{n8\}, \{3\}\rangle. \tag{A.10}$$

For pos=1, Line 11 calls CombinationsFromTree on each node appearing in a configuration as follows:

- CombinationsFromTree($n2$,3) returns $S[1]= \{\{n2, n3, n4\}\}$.
- CombinationsFromTree($n6$,3) returns $S[1]= \emptyset$.

- CombinationsFromTree($n8$,3) returns $S[1]= \emptyset$.

Given that each configuration has only one node, the operator $\otimes_t$ is not applied. Given that the first configuration yields one element and the second and third configurations yield empty results, Line 14 is called only once for $i = 1$, yielding:

$$combProduct \quad = \quad \{\{n2, n3, n4\}\} \tag{A.11}$$

For $i = 2$, $NodeComb \in \{\{n2, n6\}, \{n2, n8\}, \{n6, n8\}\}$, and $string \in \{\{1,2\}, \{2,1\}\}$. We have six configurations at this point:

$$\begin{aligned} &\langle 2, \{n2, n6\}, \{1, 2\}\rangle, \langle 2, \{n2, n6\}, \{2, 1\}\rangle, \\ &\langle 2, \{n2, n8\}, \{1, 2\}\rangle, \langle 2, \{n2, n8\}, \{2, 1\}\rangle, \\ &\langle 2, \{n6, n8\}, \{1, 2\}\rangle, \langle 2, \{n8, n8\}, \{2, 1\}\rangle. \end{aligned} \tag{A.12}$$

Line 11 calls CombinationsFromTree on each node appearing in a configuration as follows:

1. For configuration $\langle 2, \{n2, n6\}, \{1, 2\}\rangle$, we have the following calls:
   - pos=1, CombinationsFromTree($n2$,1) returns $S[1]$=$\{\{n2\}\}$.
   - pos=2, CombinationsFromTree($n6$,2) returns $S[2]$=$\{\{n6$,$n7\}\}$.
2. For configuration $\langle 2, \{n2, n6\}, \{2, 1\}\rangle$, we have the following calls:
   - pos=1, CombinationsFromTree($n2$,2) returns $S[1]$=$\{\{n2$,$n3\}$, $\{n2$, $n5\}\}$.
   - pos=2, CombinationsFromTree($n6$,1) returns $S[2]$=$\{\{n6\}\}$.
3. For configuration $\langle 2, \{n2, n8\}, \{1, 2\}\rangle$, we have the following calls:

- pos=1, CombinationsFromTree($n2$,1) returns $S[1]$={{$n2$}}.
- pos=2, CombinationsFromTree($n8$,2) returns $S[2]$= $\emptyset$.

4. For configuration $\langle 2, \{n2, n8\}, \{2, 1\}\rangle$, we have the following calls:
   - pos=1, CombinationsFromTree($n2$,2) returns $S[1]$={{$n2$,$n3$}, {$n2$,$n5$}}.
   - pos=2, CombinationsFromTree($n8$,1) returns $S[2]$= {{$n8$}}.
5. For configuration $\langle 2, \{n6, n8\}, \{1, 2\}\rangle$, we have the following calls:
   - pos=1, CombinationsFromTree($n6$,1) returns $S[1]$={{$n6$}}.
   - pos=2, CombinationsFromTree($n8$,2) returns $S[2]$= $\emptyset$.
6. For configuration $\langle 2, \{n8, n8\}, \{2, 1\}\rangle$, we have the following calls:
   - pos=1, CombinationsFromTree($n6$,2) returns $S[1]$={{$n6$,$n7$}}.
   - pos=2, CombinationsFromTree($n8$,1) returns $S[2]$= {{$n8$}}.

Hence, Line 14 is called only four times because two of the above results for $pos = 2$ are empty:

$$\begin{aligned} combProduct &= \{\{n2\}\} \otimes_t \{\{n6, n7\}\} \\ &= \emptyset \qquad (A.13) \\ combProduct &= \{\{n2, n3\}, \{n2, n5\}\} \otimes_t \{\{n6\}\} \\ &= \emptyset \qquad (A.14) \\ combProduct &= \{\{n2, n3\}, \{n2, n5\}\} \otimes_t \{\{n8\}\} \\ &= \{\{n2, n3, n8\}, \{n2, n5, n8\}\} \qquad (A.15) \\ combProduct &= \{\{n6, n7\}\} \otimes_t \{\{n8\}\} \\ &= \{\{n6, n7, n8\}\} \qquad (A.16) \end{aligned}$$

For $i = 3$, $NodeComb \in \{\{n2, n6, n8\}\}$ and $string \in \{\{1, 1, 1\}\}$. We have one configuration at this point: $\langle 3, \{n2, n6, n8\}, \{1, 1, 1\}\rangle$. Line 11 calls CombinationsFromTree on each node in this unique configuration with the corresponding results:

- pos=1, CombinationsFromTree is called with ($n2$,1), which returns $S[1]$= $\{\{n2\}\}$.
- pos=2, CombinationsFromTree is called with ($n6$,1), which returns $S[2]$= $\{\{n6\}\}$.
- pos=3, CombinationsFromTree is called with ($n8$,1), which returns $S[3]$= $\{\{n8\}\}$.

Hence, Line 14 is called only once to combine the results of the above calls, yielding:

$$\begin{aligned} combProduct &= (\{\{n2\}\} \otimes_t \{\{n6\}\}) \otimes_t \{\{n8\}\} \\ &= \emptyset \end{aligned} \tag{A.17}$$

At the end, adding the root node n_1, we have

$$lnodesets = \{\{n1, n2, n3, n4\}, \{n1, n2, n3, n8\}, \{n1, n2, n5, n8\}, \{n1, n6, n7, n8\}\}.$$

Thus, the set of combinations of four vertices extracted from the combination tree of Figure A.9 is $\{\{a, b, c, d\}, \{a, b, c, e\}, \{a, b, d, e\}, \{a, d, c, e\}\}$.

A.3.3.4 Implementation of the $\otimes_t$ operator

In general, the process of computing $S_1 \otimes_t S_2$, where S_1 and S_2 are sets of sets, can be executed in two steps:

1. The computation of $S_1 \otimes S_2$ as specified in Expression (A.2), and
2. The removal from the resulting set of those elements that do not satisfy the conditions of $\otimes_t$ specified in Expression (A.3).

We propose to compute $S_1 \otimes_t S_2 \otimes_t \ldots \otimes_t S_n$ by modeling the problem as a Constraint Satisfaction Problem (CSP) [Dechter, 2003]. A CSP, $\mathcal{P}=(\mathcal{V}, \mathcal{D}, \mathcal{C})$, is fully defined by specifying the set of variables $\mathcal{V}$, the set of their respective domains $\mathcal{D}$, and the set of constraints $\mathcal{C}$ that restrict the allowed combinations of values to variables. A solution to a CSP is an assignment of a value to each variable such that all constraints are simultaneously satisfied. In general, the task is to find one or all solutions to the CSP. CSPs are commonly used to model combinatorial problems and solve using advanced search techniques and constraint propagation algorithms [Dechter, 2003]. We model the execution of the operator $\otimes_t$ over a sequence of sets $S_1, S_2, \ldots, S_n$ as a CSP as follows. A variable V_i of the CSP is (the 'name' of) the set $S_i \in \{S_1, S_2, \ldots, S_n\}$. The domain of the variable V_i is the definition of the corresponding set. A binary constraint is applied to every two variables V_i and V_j such that $i < j$. It constrains the acceptable combinations of values for V_i and V_j to satisfy the conditions[3] specified by Expression (A.3). We solve the CSP using exhaustive backtrack search [Dechter, 2003], which yields all solutions to the problem, thus the set $S_1 \otimes_t S_2 \otimes_t \ldots \otimes_t S_n$. The ordering of the variables in the search is fixed and static, and follows that of the sets S_i. We use a standard partial-lookahead technique to improve the performance of

[3]These conditions are discussed in the proof of Theorem 18.

the search known as *forward checking* (FC) [Haralick and Elliott, 1980]. In summary, backtrack search with FC operates as follows:

1. A variable V_i is assigned a value from its domain. Initially $i = 1$.

2. All variables $V_{j>i}$ are 'revised' given V_i. To revise a variable V_j given a variable V_i, we remove all values in the domain of V_j that do not have at least one consistent value in the domain of V_i.

The search process is repeated by assigning any unassigned variable, until all the variables have been assigned. When all the variables are assigned, the assignment is a solution to the CSP, and consequently a valid element of $S_1 \otimes_t S_2 \otimes_t \ldots \otimes_t S_n$. If, at some point during search, the domain of any of the unassigned variables is empty or after a solution is found, we backtrack chronologically to consider alternative assignments to the variables. The process ends when all the values for the first variable have been considered.

A.3.3.5 Completeness & soundness of CombinationsFromTree

We first establish that CombinationsFromTree generates only connected subgraphs, then use this result to prove its soundness and completeness.

Proposition 8 *Every combination of tree nodes generated by* CombinationsFromTree *induces a connected subgraph of the combination tree and corresponds to a set of vertices that induce a connected subgraph in the graph.*

Proof: We first prove that every combination generated by CombinationsFromTree induces a connected subgraph in the combination tree. Then, we prove that the vertices corresponding to the generated combination of tree nodes form a connected subgraph of the original graph.

The proof is by induction. When Algorithm 15 is called on a tree of depth zero, the only combination returned is the root, which induces a connected subgraph in the tree. Thus, we established the base case. We form the inductive hypothesis as follows: all generated combinations from tree of depth $(d-1)$ are connected. Then, we state and prove the inductive step: If all generated combinations from tree of depth $(d-1)$ are connected, then the combinations returned from the tree of depth d are also connected. When Algorithm 15 is called on a tree of depth d rooted at *root*, it is recursively called on the children of the *root*. Each combination generated from the tree is formed of combinations of nodes obtained from calls to Algorithm 15 on subtrees of depth at most $(d-1)$. Each of those subtrees are rooted at a child of *root*, hence each combination returned from the subtrees includes a child of *root*, and is connected. When these combinations are combined, and *root* is added to them, the result is a combination of nodes that induces a connected subgraph in the combination tree.

An edge between two nodes in the combination tree exists when the graph vertices to which the tree nodes correspond are adjacent. Thus, every edge in the combination tree corresponds to an edge in the original graph. Consequently, the set of vertices corresponding to a combination of connected tree nodes are also connected in the graph. Thus, the proof holds by the principle of mathematical induction. □

Theorem 18 (CombinationsFromTree is sound and complete.) *Given a combination tree generated from a graph G with vertex v and the parameter k,* CombinationsFromTree *generates all unique k-ConnVertices sets that include the vertex v.*

Proof: First, we prove that CombinationsFromTree (Algorithm 15) generates all *k-ConnVertices* sets that include v. Consider a combination of k vertices of G

that includes vertex v and induces a connected subgraph of G. Consider also the combination tree T generated from G with vertex v and the parameter k. Proposition 5 insures that, for every connected subgraph G' induced on G by at most k vertices including v, the depth-first tree of G' rooted at v is isomorphic to a subgraph of T rooted at v. Therefore, considering all possible connected subgraphs of T of size k rooted at v guarantees that all connected subgraphs in G that include vertex v are considered.

The argument now shifts to showing that all possible induced subtrees in the combination tree including the root are considered in Line 14 of Algorithm 15. Let *root* be the root of the combination tree considered. The three loops in Lines 4, 5, and 6 ensure that Algorithm 15 systematically enumerates all the configurations[4] that lead to combinations of size $(k-1)$. Using these configurations, Algorithm 15 is recursively called in Line 11 on subtrees rooted at the children of *root*, and then the results are passed to the operator $\otimes_t$ to generate combinations of tree nodes of size $(k-1)$. The task is thus now to prove that Line 14 produces all *k-ConnVertices* sets.

The operator $\otimes_t$ is applied to the sets $S[pos]$, where *pos* varies from one to the number of subtrees in the considered configuration. The sets $S[pos]$ are produced by Algorithm 15 (see Line 11) from subtrees t_i rooted at children of *root*. Further, each element in $S[pos]$ is a combination of tree nodes and induces a connected subgraph in the tree by Proposition 8. Each element in the set produced at Line 11 by the application of the $\otimes_t$ operator is a set of tree nodes of size $(k-1)$. The set produced at Line 14 is, by the definition[5] of operator $\otimes_t$, a subset of the cross-product-like operation of the sets to which it is applied. The elements that are not removed from the 'complete' cross-product are those that verify the conditions specified in

[4]See Definition 7.
[5]See Definition 6.

Expression (A.3). The task is now to prove that:

- The elements 'ruled out' by the Expression (A.3) yield combinations of tree nodes that are already in the set (i.e., they are 'duplicate' elements).
- No 'duplicate' elements are present in the resulting set.

Let us return to the application of the operator $\otimes_t$ in Algorithm 15. Let $S[pos]$ and $S[pos+1]$ be two sets of tree-nodes combinations obtained from recursive calls to Algorithm 15 and to which Algorithm 15 applies the operator $\otimes_t$. At the lowest level of the recursive calls, the tree roots given as arguments to CombinationsFromTree in Line 11 of Algorithm 15 are single tree nodes. In Line 14, the result from subtrees that have the same parent are combined using the operation $\otimes_t$. Hence, the Expression (A.6) must hold for every pair of nodes in an element of $S[pos] \otimes_t S[pos+1]$,

Let *comb* and *comb*$'$ be two combinations generated from a combination tree such that the set of tree nodes in *comb* is different from that in *comb*$'$, but such that *comb* and *comb*$'$ correspond to the same set of graph vertices. Given a combination of tree nodes, which is element of $S[pos] \otimes_t S[pos+1]$, we showed in Proposition 8 that the nodes in the combination are connected in the tree. By construction, the root of the combination tree is one of the nodes in the combination. Hence, both *comb* and *comb*$'$ induce connected sets of nodes in the tree, and include the root node of the combination tree. Given that $comb \neq comb'$, there must necessarily exist a tree node n that has a child n_c, such that $n \in comb$, $n \in comb'$, $n_c \in comb$ and $n_c \notin comb'$. Because *comb* and *comb*$'$ both correspond to the same set of graph vertices, there must be a node $n'_c \in comb'$ such that Vertex(n'_c) =Vertex(n_c), which is impossible because it violates Expression (A.6). Indeed, we have the nodes $n, n'_c \in comb'$ such that $\exists n_c \in$Children(n) such that Vertex(n_c)=Vertex(n'_c). Because of this impossibility, we conclude that no two combinations of tree nodes in

$S[pos] \otimes_t S[pos+1]$ can correspond to the same set of graph vertices. In conclusion, no two elements in $S[pos] \otimes_t S[pos+1]$ are the same. □

A.4 Memoization

At a node n at depth *depth* in the combination tree, CombinationsFromTree (Algorithm 15) recursively calls itself (Line 11) at most the following number of times:

$$\sum_{i=1}^{k-depth-1} \left(id^i (k - depth - i - 1)^{(i-1)} \right), \tag{A.18}$$

where d denotes the degree of the graph. At each call, the arguments passed to CombinationsFromTree are a child of n and a value of *size* ranging from 1 to $(k - depth - 1)$. Hence, there are at most $d \cdot (k - depth - 1)$ distinct calls that can be made from a single node at *depth* in the combination tree.

To avoid executing the redundant calls CombinationsFromTree, the first time Algorithm 15 is called on a tree node with a given combination size, the result is stored in the node. The next time the call is made on the same node with the same combination size, the stored result is retrieved and used, which avoids re-executing the call. Hence we store at most $(k - depth - 1)$ sets of *k-ConnVertices* sets at each node. These *k-ConnVertices* sets are stored in an array indexed by the size of the combination.

Likewise, the results of the calls to k-compositions are also memoized in a data structure that is global to ConSubg (Algorithm 11). The former memoization (i.e., in CombinationsFromTree) proved to be extremely effective in reducing running time. The latter (i.e., in k-compositions) was also quite effective but to a lesser extent than the former. Neither introduced running-time overhead. The memory

overhead of the former dominates that of the latter and is analyzed in Section A.5.2.

A.5 Complexity Analysis of ConSubg

The procedure presented in this appendix can be divided into two parts:

1. Construction of the combination tree in BuildTree (Algorithm 14), and
2. Generation of the combinations from the generated combination tree in CombinationsFromTree (Algorithm 15).

Let V be the set of vertices of the graph G and k be the size of the combinations sought. Below, we assume that d is the degree of the graph.

A.5.1 Time complexity

Algorithm 15 is called recursively in Line 11 on the nodes of the combination tree generated by Algorithm 14 (Section A.3.2). The depth of tree is $(k-1)$. At a given depth, Algorithm 15 is called on each node for combination sizes varying from 1 to $(k - depth)$. Therefore, except for the root node, Algorithm 15 is called on each node $(k - depth)$ distinct times. Algorithm 15 is called only once on the root node with the combination size k. The results of calling Algorithm 15 recursively on a node of the combination tree is stored by memoization for future use in the recursion (see Section A.4). Consequently, to analyze the time complexity of Algorithm 15, we only need to account for the cost of the *distinct* calls on each node.

The number of nodes at depth *depth* is d^{depth}. Let $T(k)$ be the time complexity of calling Algorithm 15 on a node with combination size k that does *not* include the cost of the recursive step. The value $T(k)$ depends on the value of k. Adding the

number of nodes and the number of distinct calls to Algorithm 15 on each node of the combination tree yields the following time complexities per depth:

$$depth = 0 \quad : \quad T(k) \tag{A.19}$$

$$depth = 1 \quad : \quad [T(1) + T(2) + \cdots + T(k-1)] \times d \tag{A.20}$$

$$depth = 2 \quad : \quad [T(1) + T(2) + \cdots + T(k-2)] \times d^2 \tag{A.21}$$

$$depth = 3 \quad : \quad [T(1) + T(2) + \cdots + T(k-3)] \times d^3 \tag{A.22}$$

$$\cdots$$

$$depth = k-2 \quad : \quad [T(1) + T(2)] \times d^{k-2} \tag{A.23}$$

$$depth = k-1 \quad : \quad [T(1)] \times d^{k-1} \tag{A.24}$$

Expression (A.19) is the cost of calling Algorithm 15 on a node with combination size k. Expressions (A.20) to (A.24) are the cost of the distinct recursive calls. At each depth, there are d^{depth} nodes, and the algorithm is called with combination sizes ranging from 1 to $(k - depth - 1)$. Given the memoization mechanism explained in Section A.4, only $(k - depth)d^{depth}$ recursive calls are effectively executed at a given depth $depth$. Summing and grouping Expressions (A.19) to (A.24) yields the complexity of Algorithm 15 including all distinct recursive calls:

$$T(k)+T(k-1)d+T(k-2)\sum_{i=1}^{2} d^i+T(k-3)\sum_{i=1}^{3} d^i \cdots+T(2)\sum_{i=1}^{k-2} d^i+T(1)\sum_{i=1}^{k-1} d^i \tag{A.25}$$

Let C_k be an upper bound on the number of k-*ConnVertices* sets returned by Algorithm 15. All the combinations include the vertex in the label of the root node. Given that a combination tree has $\mathcal{O}(d^{(k-1)})$ nodes, we have $C_k = \mathcal{O}(d^{(k-1)^2})$. On the other hand, the $\mathcal{O}(d^{(k-1)})$ nodes of the combination tree correspond to at most $|V|$

vertices of the original graph. Thus,

$$C_k = \mathcal{O}(min(|V|^{(k-1)}, d^{(k-1)^2})). \tag{A.26}$$

When C_k is close to $\Theta(|V|^{(k-1)})$, using the brute-force algorithm, LBF-ConSubg (Algorithm 9), to generate all *k-ConnVertices* sets, is justified. Our approach is justified when C_K is much smaller than $\Theta(|V|^{(k-1)})$.

Below, we express the complexity of Algorithm 15 in terms of C_k. First, we consider a direct application of Proposition 7.

Corollary 2 (Time complexity of $S_1 \otimes_t S_2 \ldots \otimes_t S_{k-1}$ in Algorithm 15.) *The time complexity of $S_1 \otimes_t S_2 \ldots \otimes_t S_{k-1}$ is $\mathcal{O}\left(|S_1| \cdot |S_2| \cdot \ldots \cdot |S_{k-1}| \cdot k^3\right)$, where k is the size of combinations sought in Algorithm 15.*

Proof: We assume that the check $l \in$ Children(i) in Expression (A.3) is performed in constant time using a hash-table data-structure for the children of the node. The operator $\otimes_t$ as used in Algorithm 15 in Line 14 acts on s_1, s_2, …, s_{k-1}, which are elements of S_1, S_2, …, S_{k-1} respectively such that $s_1 \cup s_2 \cup \ldots \cup s_{k-1}$ is a candidate combination of size $(k-1)$ and $(|s_1| + |s_2| + \ldots + |s_{k-1}|) = (k-1)$. The operator $\otimes_t$ is left associative. Therefore, when it is applied $|S_i| \cdot |S_j|$ times the two left-most operands S_i and S_j, it yields an operand of size $|S_i| \cdot |S_j|$ whose elements' size is less than k. The cost of the application of $\otimes_t$ on S_i and S_j is $\mathcal{O}(|S_i| \cdot |S_j| \cdot k^2)$. The resulting set is used as an operand in the following application of $\otimes_t$. The operator $\otimes_t$ is applied $(k-2)$ times. Thus the cost of $(k-2)$ applications is $\mathcal{O}(|S_1| \cdot |S_2| \cdot \ldots \cdot |S_{k-1}| \cdot k^3)$. □

Proposition 9 (The complexity of $T(k)$ of Expression (A.19).)

$$T(k) = \mathcal{O}\left(k^{(k+2)} d^{(k-1)} (C_{k-1})^{(k-1)}\right). \tag{A.27}$$

Proof: We will consider the time complexity of the recursive call to be constant, because it is already accounted for in Expression (A.25). We assume without loss of generality that $k < d$. The bounds will also hold for $d \leq k$ because Line 4 of Algorithm 15 chooses the minimum of k and d. The loop in Line 4 iterates at most $(k-1)$ times, the loop in Line 5 iterates $\binom{d}{(k-1)}$ times, and the loop in Line 6 iterates $\mathcal{O}\left(k^{(k-2)}\right)$ times. Multiplying the three costs yields

$$\mathcal{O}\left(kd^{(k-1)}k^{(k-2)}\right). \tag{A.28}$$

The loop in Line 8 iterates at most k times. We consider that the body of this loop is executed in constant time (see above). The loop in Line 14 iterates at most $(C_{k-1})^{(k-1)}$ times, and the cost of the loop in Line 14 is $\mathcal{O}\left((C_{k-1})^{(k-1)}k^3\right)$. The complexity of the body of the loop in Line 6 is dominated by the complexity of the loop in Line 14. Thus, the cost for the body of the loop in Line 6 is:

$$\mathcal{O}\left((C_{k-1})^{(k-1)}k^3\right). \tag{A.29}$$

Combining Expressions (A.28) and (A.29) yields

$$T(k) = \mathcal{O}\left(k^{(k+2)}d^{(k-1)}(C_{k-1})^{(k-1)}\right). \tag{A.30}$$

□

Substituting Expression (A.27) in Expression (A.25) yields:

$$\mathcal{O}\left(k^{(k+2)}d^{(k-1)}(C_{k-1})^{(k-1)}\right) \tag{A.31}$$

$$+ \quad \mathcal{O}\left(k^{(k+1)}d^{(k-1)}(C_{k-2})^{(k-2)}\right) \tag{A.32}$$

$$+ \quad \mathcal{O}\left(k^{(k)}(d^{(k-1)}+d^{(k-2)})(C_{k-3})^{(k-3)}\right) \tag{A.33}$$

$$+ \quad \mathcal{O}\left(k^{(k-1)}(d^{(k-1)}+d^{(k-2)}+d^{(k-3)})(C_{k-4})^{(k-4)}\right) \tag{A.34}$$

$$\cdots$$

$$+ \quad \mathcal{O}\left(k^4\sum_{i=2}^{k-1}d^i(C_1)\right) \tag{A.35}$$

$$+ \quad \mathcal{O}\left(k^3\sum_{i=1}^{k-1}d^i\right). \tag{A.36}$$

Note that the Expressions (A.32) to (A.36) belong to $\mathcal{O}\left(k^{(k+1)}d^{(k-1)}(C_{k-1})^{(k-1)}\right)$, because $(C_{k-1})^{(k-1)} > (C_\alpha)^\alpha$ for all $\alpha < (k-1)$ and $kd^{(k-1)} > \sum_{i=1}^{k-1} d^i$. There are $(k-1)$ terms in Expressions (A.32) to (A.36). Hence, their sum, which is the complexity of computing the combinations from a combination tree, is bounded by

$$\mathcal{O}\left(k^{(k+2)}d^{(k-1)}(C_{k-1})^{(k-1)}\right). \tag{A.37}$$

Because Algorithm 15 is repeated $|V|$ times, the overall complexity of Algorithm 11 is

$$\mathcal{O}\left(|V|k^{(k+2)}d^{(k-1)}(C_{k-1})^{(k-1)}\right). \tag{A.38}$$

Note that, in Expression (A.38), d is the degree of the graph. Thus, the complexity of our algorithm, ConSubg (Algorithm 11), depends on the degree of the graph. Alternatively, on graphs of bounded degree, the complexity of ConSubg is dominated by the number of connected subgraphs (i.e., C_{k-1}^{k-1}). In contrast, the complexity of the

brute-force algorithm BF-ConSubg (Algorithm 9) is dominated by the number of vertices in the graph (i.e., $\Theta|V|^k$) and that of its localized version, LBF-ConSubg (Algorithm 10), depends on the degree of the graph but not on the number of connected components (i.e., $\mathcal{O}(|V|d^{k(k-1)})$).

A.5.2 Space complexity

The space complexity is dominated by the space required to store the results of Algorithm 15 for every considered k in each node of the combination tree generated by Algorithm 14. Therefore, the space complexity is

$$\mathcal{O}\left(d^{(k-1)}(C_{k-1})^{(k-1)}\right). \tag{A.39}$$

This memory requirement is detrimental to the performance of our algorithm on problems that have many connected subgraphs, a situation that corresponds to high density graphs and large value of k. Indeed, Figure A.11 illustrates a situation where ConSubg cannot terminate on a graph of 100 vertices, of density 40%, and with $k = 6$.

A.6 Correctness

Theorem 19 (Soundness and completeness of ConSubg) ConSubg *(Algorithm 11) generates all unique* k-ConnVertices *sets from the combination trees.*

Proof: Let v be the first vertex considered in Algorithm 11. Algorithm 15 returns all unique combinations of vertices including v, as established by Theorem 18. Therefore, any *k-ConnVertices* set that includes v is generated from the combination tree with a root node labeled with vertex v.

After removing vertex v from the graph, Algorithm 11 repeats the same process for a vertex v' chosen arbitrarily from the graph. All *k-ConnVertices* sets starting at v' in the updated graph are generated in a similar manner. Thus, all *k-ConnVertices* sets in the graph that include v' are generated either when Algorithm 11 processes v (and those combinations would thus include v) or when it processes v'. Hence, no *k-ConnVertices* set that includes v' can be missed. Finally, all *k-ConnVertices* sets for each of the remaining graph vertices are generated in a similar manner. Indeed, the *k-ConnVertices* sets for a given vertex are generated by Algorithm 11 at any point either before the vertex is considered or when it is processed (at the latest).

Proposition 4 asserts that no *k-ConnVertices* set can be generated from two distinct combination trees. Theorem 18 guarantees that all *k-ConnVertices* sets generated from a combination tree are unique. Therefore, all *k-ConnVertices* sets generated by Algorithm 11 are also unique. □

A.7 Empirical Evaluations

Below, we compare the performance of the proposed algorithm, ConSubg, to that of the brute-force algorithm, BF-ConSubg, and its localized variant, LBF-ConSubg. We measured the CPU time of executing the algorithms on graphs with a fixed degree (Section A.7.1), scale-free graphs (Section A.7.2), and graphs of constraint satisfaction benchmarks (Section A.7.3). For random graphs (i.e., fixed degree and scale-free graphs), we generated 30 instances per data point. In all cases, we averaged the results on the instances that completed within one hour of CPU time. A missing data point corresponds to an experiment that did not terminate within the one-hour time limit.

A.7.1 Graphs of a fixed degree

Because ConSubg is designed to exploit the structure of the graph, one would expect that its performance would be worse on graphs where all vertices have the same degree (i.e., graphs lacking structure). For this purpose, we wrote a generator to generate connected random graphs where all vertices have the same degree. Given the number of vertices of a graph and the degree of the graph (which is a constant less than the number of vertices), we first determine the number of edges in the graph using the hand-shaking theorem. Then, we repeat the following steps until each edge is connected to two vertices. We select an edge that has not been connected to any vertices. Then, we select two random vertices, and connect them with the edge only if the two vertices are not already adjacent and if the degree of each of them is less than the specified degree entered as input. If the resulting graph is not connected, we discard it and repeat the process.

To test our algorithms on the graphs generated as described above, we conducted the experiments summarized in Table A.2. In those experiments, we investigated the

Table A.2: Experiments on random graphs of a fixed degree.

Experiment	$\|V\|$	**Degree**	**Size of subgraphs**	**Figure**
I	100	10	$k = 3, 4, 5, 6, 7$	Figure A.10
	100	40	$k = 3, 4, 5, 6$	Figure A.11
II	{100,150,...,900}	10	$k = 4$	Figure A.12
	{100,150,...,900}	40	$k = 4$	Figure A.13
III	100	{5,10,...,40}	$k = 4$	Figure A.14
	300	{5,10,...,40}	$k = 4$	Figure A.15
	400	{5,10,...,40}	$k = 4$	Figure A.16

effect of increasing the size of the subgraph on sparse and dense graphs of 100 vertices (Experiment I), the effect of increasing the number of vertices for a fixed size of the subgraph on sparse and dense graphs (Experiment II), and the effect of increasing

the density of the graph for a fixed subgraph size and on graphs with 100, 300, and 400 vertices.

Experiment I Figure A.10 shows the performance of the three algorithms on sparse graphs for increasing combination values k. On those graphs, ConSubg clearly outperforms BF-ConSubg and LBF-ConSubg. Notably, BF-ConSubg and LBF-ConSubg fail to terminate within the CPU time limit for $k = 7$ while ConSubg succeeds. Figure A.11 shows the only experiment where ConSubg fails to terminate because of the memory limitation while BF-ConSubg and LBF-ConSubg do. This situation occurs for $k = 6$. Note that the graph density in this experiment is 40.40%. Clearly, ConSubg fails to handle large values of k on dense graphs because of its memory requirements as discussed in Section A.5.2. However, note that large dense graphs are not of much use in practice because they have a prohibitively large number of connected subgraphs, which are challenging to store and operate on even if we were able to generate them.

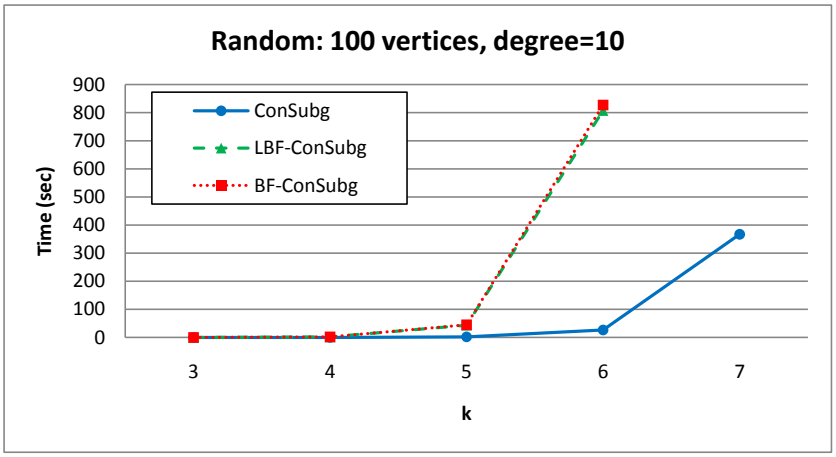

Figure A.10: Increasing k with $|V|$=100 and of degree 10.

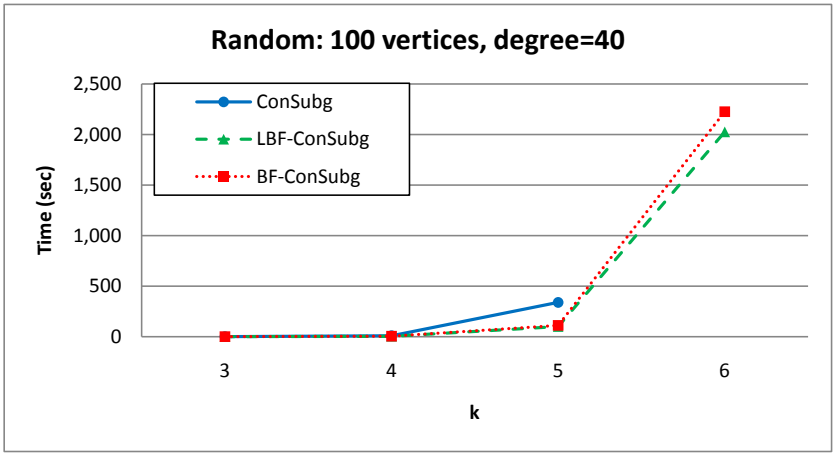

Figure A.11: Increasing k on graphs with $|V|$=100 and of degree 40.

Experiment II Figures A.12 and A.13 show that our new algorithm ConSubg vastly outperforms the brute-force algorithm BF-ConSubg and its localized version LBF-ConSubg as the size of the network increases. Indeed, on graphs of degree 10, BF-ConSubg and LBF-ConSubg cannot handle graphs beyond 650 vertices. On graphs of degree 40, they both stop at graphs with 600 vertices. ConSubg easily scales to larger graphs in both cases and its cost remains relatively negligible.

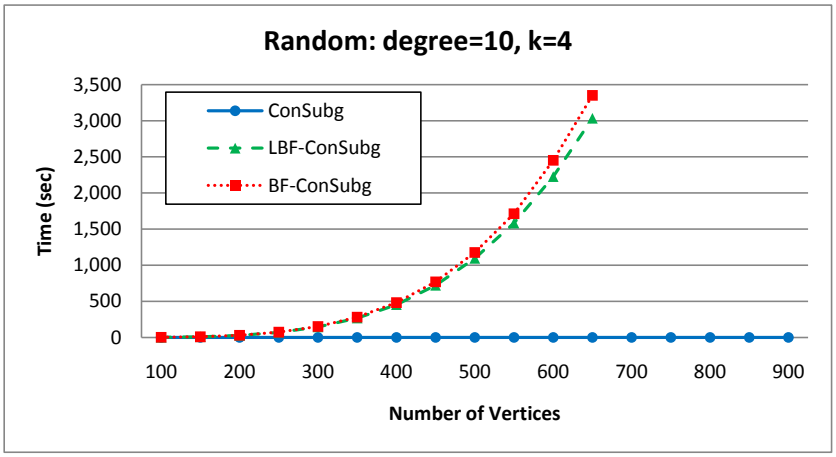

Figure A.12: Increasing the number of vertices for $k = 4$ and graphs of degree 10.

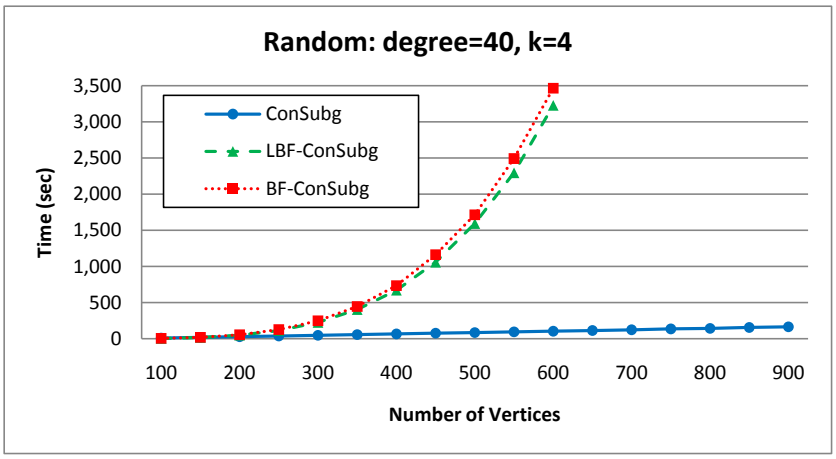

Figure A.13: Increasing the number of vertices for $k = 4$ and graphs of degree 40.

Experiment III Figures A.14, A.15, and A.16 show the performance of the algorithms as the degree of the graph grows on graphs with 100, 300, and 500 vertices respectively and for a fixed combination size $k = 4$. Here again, we see that the performance of our new algorithm ConSubg deteriorates as the density of the graph increases, again reaffirming that ConSubg is not suitable for dense graphs (see Figure A.14). However, as the number of vertices increases, the brute-force algorithms BF-ConSubg and its localized version LBF-ConSubg are an order of magnitude more costly than ConSubg (see Figures A.15 and A.16). The main drawback of BF-ConSubg and LBF-ConSubg is that they generate many subgraphs that are not connected and, thus, must be discarded after they are generated, which ConSubg is designed to not do. Incidentally, in Figures A.14, A.15, and A.16 the number of combinations generated by LBF-ConSubg and BF-ConSubg is constant for all degree values. However the corresponding curves present a slight positive slope. This slight slope can be attributed to the cost of testing the connectivity of the generated combinations.

In summary and in all our experiments on randomly generated graphs of a fixed

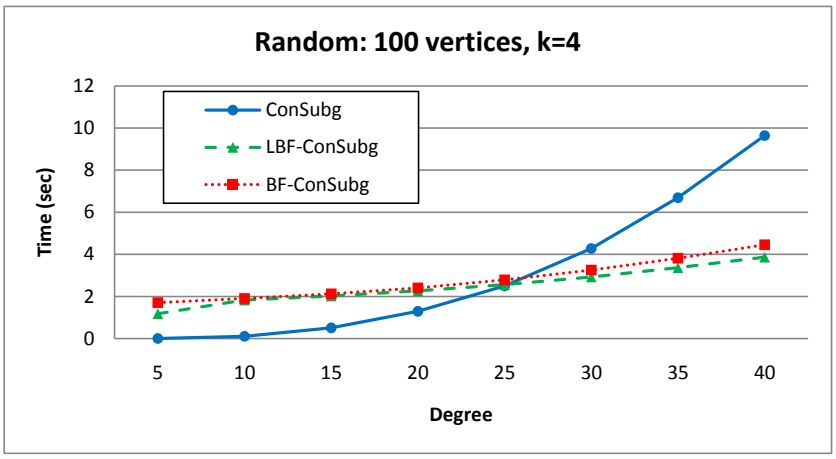

Figure A.14: Increasing the degree of the vertices for $|V|$=100 and $k = 4$.

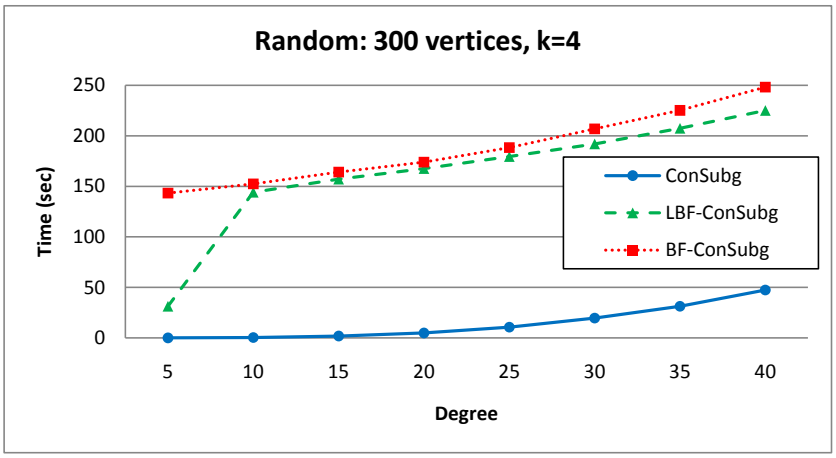

Figure A.15: Increasing the degree of the vertices for $|V|$=300 and $k = 4$.

degree, ConSubg usually and largely outperforms LBF-ConSubg, which always outperforms BF-ConSubg. In particular:

1. The performances of LBF-ConSubg and BF-ConSubg are notably similar, while the localized version is always slightly quicker than, or at least as quick as, the original brute-force algorithm.

2. As the density of the graph increases, the likelihood that a given combination of k vertices induces a connected subgraph increases, and the benefit of exploiting

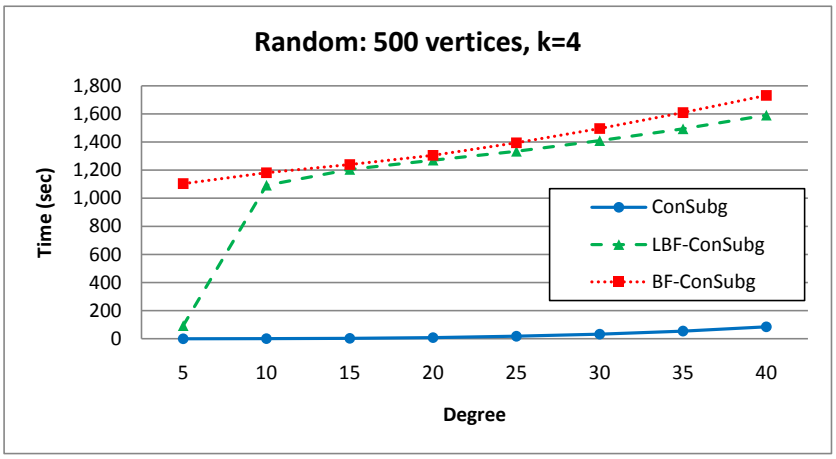

Figure A.16: Increasing the degree of the vertices for $|V|$=500 and $k = 4$.

the structure of the graph obviously decreases. At some point, the cost of building the data structures necessary for ConSubg becomes detrimental. Note that, of all the experiments we conducted, this problem is visible only in the experiments shown in Figures A.11 and A.16 where the graph density is 40.40%, which is considered a high-density graph in practice.

A.7.2 Scale-free graphs

Scale-free graphs are commonly thought to model social networks and have received an increased attention in recent years. To Generate Scale-Free Networks, we used the procedure `scale_free_graph` from the open-source software `NetworkX`.[6] The procedure is based on the model proposed in [Bollobás *et al.*, 2003]. We chose the default parameters for `scale_free_graph` (alpha=0.41, beta=0.54, gamma=0.05, delta_in=0.2, and delta_out=0) to generate the directed graph, and used the procedure `to_undirected` in NetworkX to obtain the corresponding undirected graph. We generated undirected graphs of 100, 200, ..., 900 vertices.

[6] `NetworkX 1.3` http://networkx.lanl.gov/.

Increasing the number of vertices Figure A.17 shows the CPU time needed to generate all subgraphs of size four (i.e., $k = 4$) by each of the three algorithms compared. We see that both our algorithms ConSubg and LBF-ConSubg scale significantly better with increasing number of vertices than the brute-force algorithm BF-ConSubg. Also, ConSubg clearly outperforms LBF-ConSubg.

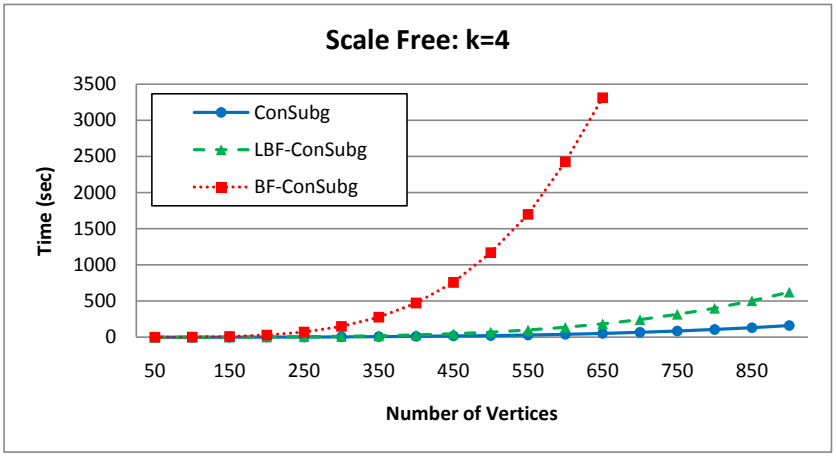

Figure A.17: Increasing the number of vertices with $k = 4$ in scale-free networks.

Increasing k, the combination size Figure A.18 compares the performance of the three algorithms on scale-free networks of 100 vertices as k grows. The brute-force algorithm, BF-ConSubg, does not terminate within the time limit of one hour for $k = 7$, and the performance of its localized version, LBF-ConSubg, is an order of magnitude worse than that of ConSubg.

In summary, ConSubg clearly outperforms its competitors on scale-free graphs, which are of practical importance. Interestingly, the performance of the localized variant of the brute-force algorithm, LBF-ConSubg, is significantly better than that of the original algorithm, albeit it is not as good as that of ConSubg. Thus, while localization helps, it does not take full advantage of the problem structure.

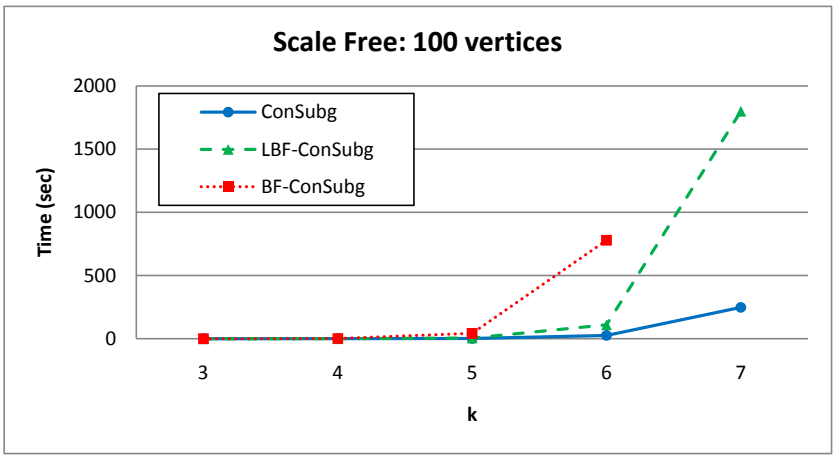

Figure A.18: Increasing k in scale-free networks of 100 vertices.

A.7.3 CSP graphs

We examined the benchmarks of constraint satisfaction problems (CSPs) used in the 2009 Constraint Solver Competition,[7] and considered the dual graphs of 1689 CSP instances. Given that our algorithm is best suited for sparse graphs, it is appropriate to report the density of those benchmarks. 56.5% of the dual graphs of the 1689 benchmark instances have density less than or equal to 15%. The dual graphs can be reformulated, without loss of information, into equivalent graphs by removing redundant edges [Dechter, 2003]. We applied the algorithm proposed in [Janssen *et al.*, 1989; Jégou and Vilarem, 1993] to remove redundant edges. The resulting minimal dual graphs that have density less than or equal to 11% constitute 79.7% of all tested instances. Consequently, it is fair to say that most benchmark problems, including the most challenging ones, have sparse dual graphs.

We executed ConSubg, LBF-ConSubg, and BF-ConSubg with $k = 5$ on the dual graphs of those 1689 instances after removing redundancies. Each experiment on a single instance was limited to a one hour. Table A.3 shows a summary of the

[7] http://www.cril.univ-artois.fr/CSC09/benchs/CSC09.tar.

results. Below we relate some observations:

Table A.3: Summary of results on 1689 CSP benchmark instances.

Number of instances	ConSubG	LBF-ConSubG	BF-ConSubG
Completed	1633	1602	918
Not completed	56	87	771
Algorithm performs best	1296	35	0
Completed by no algorithm	21		
Missed by only ConSubG	35		

- ConSubG is clearly the champion, both in terms of the number of instances it solves (1633 for ConSubG versus 1602 for LBF-ConSubG and 918 for BF-ConSubG) and the number of instances on which it performs as good as, or better than, the other two algorithms (1296 for ConSubG versus 35 for LBF-ConSubG and 0 for BF-ConSubG).

- ConSubG failed to complete the 56 instances because it ran out of memory space as predicted in Section A.5.2 well before the one-hour time limit imposed on the experiments. However, LBF-ConSubG and BF-ConSubG failed to complete 87 and 771 instances respectively only because of the time limitation.

- ConSubG does not terminate within one hour processing time on 35 instances that were completed by both LBF-ConSubG and BF-ConSubG. A quick examination of those 35 instances shows that they all come from a single problem class (called `bddSmall`). They have high density (average 31.59%) and relatively few vertices (exactly 133 vertices), which means that most of the combinations enumerated by LBF-ConSubG and BF-ConSubG are connected. Such problems are not suited for ConSubG, which is intended for large problems with small density where LBF-ConSubG and BF-ConSubG would fail. Further, those 35 instances yield a *huge* number of *k-ConnVertices* (from 65,848,590

to 102,891,308), which is one to two orders of magnitude the number of *k-ConnVertices* of all 1654 other instances. Thus, that constraint propagation algorithms intended to be applied on such problems become totally impractical. In conclusion, this class of problems is not relevant to the techniques targeted by our approach.

- Excluding the 35 `bddSmall` instances discussed above, we notice that the set of 21 instances not completed by ConSubG is a strict subset of the set of 87 instances not completed by LBF-ConSubG, which is in turn a strict subset of the 771 instances not completed by BF-ConSubG. Thus, except for the 35 `bddSmall` instances, ConSubG terminates on more instances than LBF-ConSubG, which terminates on more instances than BF-ConSubG.

Tables A.4, A.5, and A.6 provide condensed information on 1617 instances pertaining to 123 classes of problems (the remaining 72 instances tested were too simple to be reported). For each class, the tables provide the number of instances, the average number of vertices of the dual graphs after removing redundant edges, and the average density of the resulting graphs. The tables provide also the average CPU time and the number of instances solved by ConSubG, LBF-ConSubG, and BF-ConSubG. The average CPU time is computed over the number of completed instances by the algorithm. Finally, those tables give the average number of connected subgraphs of size 5. Entries shown in boldface in the table correspond to the best values found. The dash character (-) indicates that the algorithm did not terminate on any instance in the class. ConSubG runs out of memory, and LBF-ConSubG and BF-ConSubG run out of time.

Tables A.4 and A.5 show instances where ConSubG clearly outperforms the other two algorithms, frequently solving instances that resisted other algorithms and always

reducing the CPU by often *several orders of magnitude*.

Table A.6 shows seven problem classes where the performance of ConSubG was the least spectacular. As one can clearly see, the graphs of those instances have relatively few vertices but high density. However, except for the class `bddSmall`, ConSubG solves all instances solved by the other algorithms and CPU time does not exceed half a second.

A.8 Conclusion

In this appendix, we proposed a new algorithm, ConSubG, for computing all connected subgraphs of a graph that have a fixed size. This problem is particularly important for enforcing high levels of consistency on Constraint Satisfaction Problems. We compared the performance of ConSubG to that a brute-force algorithm, BF-ConSubG, that generates all subgraphs then discards those that are not connected and also to a localized version of the brute-force algorithm, LBF-ConSubG, which we also proposed. We showed that ConSubG outperforms all other algorithms on structured graphs but is not suited for dense graphs when k is relatively large.

Our contributions are: (1) the posing of the problem of generating all connected subgraphs of a graph that have a fixed size, (2) the identification of an application where it is needed, and (3) the design and evaluation of a new algorithm for solving it. We are currently investigating how to reduce, or eliminate, the memory requirements while maintaining the processing time within practical limits.

Table A.4: Results of experiments on CSP benchmarks for $k = 5$ (Part 1).

Benchmark	#Instances	#Vertices (average)	Density % (average)	ConSub Time [ms] (average)	ConSub #Instances solved	LBF Time [ms] (average)	LBF #Instances solved	BF Time [ms] (average)	BF #Instances solved	#Combinations (average)
aim-100	24	262.58	1.82	**126.67**	**24**	292,679.38	16	740,320.63	16	42,677.21
aim-200	24	532.75	0.94	**419.17**	**24**	1,765,608.57	7	-	0	134,159.83
aim-50	24	129.58	3.55	**41.67**	**24**	229,633.33	**24**	99,642.50	20	14,076.46
allIntervalSeries	14	563.43	2.58	**172.86**	**14**	16,051.43	**14**	387,881.25	8	33,688.64
BH-4-4	10	431.00	0.86	**110.00**	**10**	22,413.00	**10**	-	0	26,673.60
BH-4-7	15	1,261.00	0.30	**417.33**	**15**	89,696.67	**15**	-	0	89,585.80
bqwh-15-106	10	592.30	0.62	**129.00**	**10**	247,296.00	**10**	-	0	30,528.40
bqwh-18-141	10	876.90	0.42	**207.00**	**10**	485,233.00	**10**	-	0	47,419.70
chessbdColor	6	405.67	3.04	**1,486.67**	**6**	896,425.00	4	859,603.33	3	369,164.67
coloring	11	198.73	3.34	**39.09**	**11**	468,465.45	**11**	12,216.67	9	9,603.55
composed-25-1-2	6	224.00	1.66	**50.00**	**6**	14,581.67	**6**	2,347,030.00	**6**	12,398.83
composed-25-1-25	5	247.00	1.52	**58.00**	**5**	23,394.00	**5**	-	0	14,528.00
composed-25-1-40	5	262.00	1.44	**62.00**	**5**	31,174.00	**5**	-	0	15,795.80
composed-25-1-80	6	302.00	1.26	**66.67**	**6**	40,040.00	**6**	-	0	19,014.67
composed-25-10-20	5	620.00	0.59	**144.00**	**5**	328,602.00	**5**	-	0	34,134.20
composed-75-1-2	5	624.00	0.60	**156.00**	**5**	1,317,028.00	**5**	-	0	41,653.00
composed-75-1-25	5	647.00	0.58	**168.00**	**5**	1,383,910.00	**5**	-	0	43,806.40
composed-75-1-40	5	662.00	0.57	**170.00**	**5**	1,475,066.00	**5**	-	0	45,147.60
composed-75-1-80	5	702.00	0.54	**184.00**	**5**	1,754,266.00	**5**	-	0	48,667.80
dag-half	15	56.00	21.68	**1,595.33**	**15**	2,490.67	**15**	2,576.00	**15**	343,818.73
driver	2	2,136.00	0.82	**845.00**	**2**	1,430.00	1	1,993,930.00	1	136,263.50
dubois	13	65.38	5.47	**3.08**	**13**	14.62	**13**	103,950.77	**13**	597.85
ehi-85	5	4,108.40	0.09	**1,740.00**	**5**	-	0	-	0	310,019.40
ehi-90	5	4,368.00	0.09	**1,910.00**	**5**	-	0	-	0	329,943.00
frb30-15	5	225.40	1.65	**50.00**	**5**	52,640.00	**5**	1,696,837.50	4	13,743.00
frb35-17	5	312.00	1.15	**70.00**	**5**	179,202.00	**5**	-	0	20,030.80
frb40-19	5	371.80	0.97	**86.00**	**5**	232,160.00	**5**	-	0	24,209.20
frb45-21	5	436.60	0.83	**100.00**	**5**	348,554.00	**5**	-	0	28,827.00
frb50-23	5	480.40	0.77	**120.00**	**5**	479,054.00	**5**	-	0	32,679.80
frb53-24	5	540.40	0.67	**134.00**	**5**	605,546.00	**5**	-	0	36,977.20
frb56-25	5	558.80	0.66	**148.00**	**5**	691,146.00	**5**	-	0	38,599.20
frb59-26	5	596.60	0.62	**164.00**	**5**	868,088.00	**5**	-	0	41,594.80
geom	10	422.80	0.90	**99.00**	**10**	73,799.00	**10**	-	0	24,598.40
golombRlrArity3	11	751.00	0.90	**233.64**	**11**	39,200.00	**11**	180,920.00	1	47,821.55
golombRlrArity4	5	238.40	1.39	**136.00**	**5**	517,702.00	**5**	792,390.00	3	46,421.60
hanoi	5	46.60	13.43	**0.00**	**5**	**0.00**	**5**	24,966.00	**5**	42.60
haystacks	5	1,539.20	0.38	**454.00**	**5**	11,812.00	**5**	-	0	65,296.60
jobShop-e0ddr1	10	265.00	1.37	**54.00**	**10**	17,356.00	**10**	-	0	11,570.00
jobShop-e0ddr2	10	265.00	1.37	**51.00**	**10**	15,673.00	**10**	-	0	11,555.70
jobShop-enddr1	10	265.00	1.37	**51.00**	**10**	17,593.00	**10**	-	0	11,570.00
jobShop-enddr2	6	265.00	1.37	**50.00**	**6**	14,650.00	**6**	-	0	11,571.50
jobShop-ewddr2	10	265.00	1.37	**48.00**	**10**	15,849.00	**10**	-	0	11,555.70
js-taillard-15	10	1,785.00	0.21	**505.00**	**10**	234,190.00	**10**	-	0	88,542.80
js-taillard-20	10	4,180.00	0.09	**1,677.00**	**10**	519,587.00	**10**	-	0	220,239.10
js-taillard-20-15	10	3,130.00	0.12	**1,113.00**	**10**	357,110.00	**10**	-	0	165,069.40
knights	10	52.00	18.46	**9.00**	**10**	266.00	**10**	114.44	9	2,009.00
langford	4	380.75	3.21	**97.50**	**4**	3,905.00	**4**	110.00	1	19,725.50
langford2	14	446.71	2.94	**122.14**	**14**	4,650.71	**14**	327,998.33	6	23,874.36
langford3	11	948.82	1.09	**300.00**	**11**	11,165.45	**11**	44,445.00	2	53,292.36
langford4	10	999.00	1.10	**321.00**	**10**	11,833.00	**10**	86,395.00	2	56,382.60
lexHerald	10	487.90	8.34	8,895.00	**10**	110.00	4	115.00	4	2,502,517.60
lexPuzzle	14	289.00	7.87	**2,211.54**	**13**	414,564.00	10	106,602.22	9	657,299.38

Results continue in next table.

Table A.5: Results of experiments on CSP benchmarks for $k = 5$ (Part 2).

Benchmark	#Instances	#Vertices (average)	Density % (average)	ConSub Time [ms] (average)	ConSub #Instances solved	LBF Time [ms] (average)	LBF #Instances solved	BF Time [ms] (average)	BF #Instances solved	#Combinations (average)
modifiedRenault	40	151.58	1.78	**16.50**	**40**	8,893.00	**40**	323,745.25	**40**	6,036.60
nengfa	2	976.50	1.40	**145.00**	**2**	33,065.00	**2**	217,290.00	1	40,707.00
ogdPuzzle	15	263.60	14.88	**2,164.29**	**14**	365,969.09	11	88,025.00	10	598,629.64
os-gp	15	1,000.00	0.38	**270.67**	**15**	108,932.67	**15**	-	0	52,697.00
os-taillard-10	10	900.00	0.42	**236.00**	**10**	256,556.00	**10**	-	0	51,909.80
os-taillard-15	10	3,150.00	0.12	**1,151.00**	**10**	901,699.00	**10**	-	0	191,352.40
os-taillard-4	10	48.00	7.09	**7.00**	**10**	388.00	**10**	914.00	**10**	1,563.90
os-taillard-5	10	100.00	3.54	**18.00**	**10**	4,112.00	**10**	39,267.00	**10**	4,203.00
os-taillard-7	10	294.00	1.25	**60.00**	**10**	48,247.00	**10**	-	0	15,300.80
pigeons	10	309.20	3.90	**79.00**	**10**	3,756.00	**10**	88,726.00	5	16,210.60
pret	8	70.00	5.36	**2.50**	**8**	35.00	**8**	19,972.50	**8**	497.00
primes-10	15	44.00	11.01	**28.00**	**15**	2,839.33	**15**	3,246.00	**15**	9,520.73
primes-15	16	46.25	10.39	**80.63**	**16**	2,403.13	**16**	3,223.75	**16**	25,388.38
primes-20	15	48.00	10.81	**103.33**	**15**	3,601.33	**15**	4,612.00	**15**	32,392.33
primes-25	15	48.00	11.81	**99.33**	**15**	2,928.67	**15**	3,838.67	**15**	31,125.33
primes-30	15	60.00	8.96	**121.33**	**15**	5,952.67	**15**	6,198.00	**15**	38,786.40
QCP-10	15	822.00	0.46	**213.33**	**15**	281,187.33	**15**	-	0	47,558.87
QCP-15	15	2,519.27	0.15	**853.33**	**15**	1,306,854.67	**15**	-	0	155,672.87
queenAttacking	6	723.50	2.39	**235.00**	**6**	17,148.33	**6**	65,610.00	2	40,884.00
queens	6	141.17	12.25	**31.67**	**6**	971.67	**6**	212,408.00	5	6,536.67
queensKnights	8	426.38	3.59	**127.50**	**8**	4,282.50	**8**	29,032.00	5	22,682.25
QWH-10	10	756.00	0.49	**195.00**	**10**	309,081.00	**10**	-	0	43,646.00
QWH-15	10	2,324.00	0.16	**760.00**	**10**	1,324,365.00	**10**	-	0	142,604.00
ramsey3	8	794.63	1.35	**1,188.75**	**8**	468,151.43	7	101,260.00	1	287,281.25
ramsey4	1	2,300.00	0.25	**4,000.00**	**1**	-	0	-	0	921,557.00
rand-2-23	10	253.00	1.52	**60.00**	**10**	1,877.00	**10**	-	0	12,194.00
rand-2-24	10	276.00	1.39	**66.00**	**10**	2,127.00	**10**	-	0	13,466.00
rand-2-25	10	300.00	1.28	**68.00**	**10**	2,351.00	**10**	-	0	14,801.00
rand-2-26	10	325.00	1.19	**79.00**	**10**	2,610.00	**10**	-	0	16,199.00
rand-2-27	10	351.00	1.10	**83.00**	**10**	2,872.00	**10**	-	0	17,660.00
rand-2-30-15	20	220.90	1.70	**48.50**	**20**	45,228.00	**20**	2,184,997.00	**20**	13,641.25
rand-2-30-15-fcd	20	221.55	1.69	**50.50**	**20**	48,451.00	**20**	2,223,963.50	**20**	13,700.70
rand-2-40-19	25	337.88	1.12	**83.60**	**25**	148,037.20	**25**	-	0	22,288.68
rand-2-40-19-fcd	25	338.60	1.12	**79.60**	**25**	157,371.60	**25**	-	0	22,361.60
rand-2-50-23	25	467.44	0.81	**121.20**	**25**	374,346.40	**25**	-	0	32,173.88
rand-2-50-23-fcd	25	466.72	0.81	**117.60**	**25**	370,328.80	**25**	-	0	32,116.40
rand-3-20-20	25	58.44	6.07	**13.20**	**25**	2,000.80	**25**	2,563.20	**25**	4,122.32
rand-3-20-20-fcd	25	58.72	6.02	**11.20**	**25**	2,045.60	**25**	2,619.60	**25**	4,109.84
rand-3-24-24	25	74.44	4.95	**20.00**	**25**	6,536.80	**25**	8,828.00	**25**	6,766.32
rand-3-24-24-fcd	25	74.76	4.95	**18.80**	**25**	6,652.00	**25**	8,943.60	**25**	7,066.68
rand-3-28-28	30	93.00	4.08	**30.67**	**30**	19,161.00	**30**	27,041.67	**30**	10,964.03
rand-3-28-28-fcd	29	93.00	4.08	**31.38**	**29**	19,083.10	**29**	27,315.17	**29**	10,855.48
renault	2	123.50	2.06	**15.00**	**2**	2,530.00	**2**	121,715.00	**2**	3,918.00
rlfapGraphs	10	2,638.00	0.18	**767.00**	**10**	480,292.00	**10**	-	0	125,167.30
rlfapGraphsMod	10	2,680.20	0.11	**774.00**	**10**	468,671.00	**10**	-	0	115,128.20
rlfapScens	10	3,702.60	0.12	**1,238.00**	**10**	458,117.00	**10**	-	0	175,810.00
rlfapScens11	10	4,103.00	0.09	**1,343.00**	**10**	440,730.00	**10**	-	0	188,087.00
rlfapScensMod	10	1,975.10	0.33	**607.00**	**10**	207,821.00	**10**	-	0	86,347.10

Results continue in next table.

Table A.6: Results of experiments on CSP benchmarks for $k = 5$ (Part 3).

Benchmark	#Instances	#Vertices (average)	Density % (average)	ConSub Time [ms] (average)	ConSub #Instances solved	LBF Time [ms] (average)	LBF #Instances solved	BF Time [ms] (average)	BF #Instances solved	#Combinations (average)
schurrLemma	9	375.22	3.64	**168.89**	**9**	622,366.67	**9**	325,518.33	6	44,669.67
ssa	7	1,505.71	0.22	**135.71**	**7**	48,061.43	**7**	694,430.00	1	28,497.00
subs	9	385.00	1.05	**92.22**	**9**	10,845.56	**9**	2,307,540.00	1	19,848.78
super-queens	5	211.80	4.93	**54.00**	**5**	9,212.00	**5**	493,927.50	4	13,722.20
tightness0.1	15	752.07	0.52	**214.00**	**15**	11,058.67	**15**	-	0	42,869.33
tightness0.2	15	414.00	0.92	**106.00**	**15**	119,200.67	**15**	-	0	27,847.67
tightness0.35	15	250.00	1.48	**54.00**	**15**	133,288.00	**15**	-	0	15,384.33
tightness0.5	15	180.00	1.99	**32.00**	**15**	67,550.00	**15**	768,524.00	**15**	9,522.47
tightness0.65	15	135.00	2.54	**20.00**	**15**	26,266.00	**15**	178,738.67	**15**	5,952.00
tightness0.8	25	103.00	3.16	**10.40**	**25**	7,806.80	**25**	45,634.40	**25**	3,498.24
tightness0.9	25	84.00	3.67	**7.20**	**25**	2,733.60	**25**	16,226.80	**25**	2,192.64
TSP-20	15	230.00	1.59	**45.33**	**15**	1,572.00	**15**	2,643,796.67	**15**	10,168.00
TSP-25	15	350.00	1.06	**78.67**	**15**	2,800.67	**15**	-	0	16,613.00
ukPuzzle	13	234.00	13.69	**1,497.50**	**12**	76,992.00	10	95,790.00	10	456,936.25
varDimacs	9	810.56	0.95	**113.33**	**9**	116,662.22	**9**	821,695.00	2	29,054.44
wordsPuzzle	14	253.21	14.20	**2,252.86**	**14**	352,576.00	10	17,054.44	9	631,028.36
bddSmall	35	133.00	31.59	-	0	**386,073.43**	**35**	435,296.00	**35**	80,665,957.60
dag-rand	15	16.00	94.17	66.67	**15**	**4.00**	**15**	4.67	**15**	4,367.13
ogdVg	45	21.62	51.85	460.22	**45**	**59.56**	**45**	64.67	45	65,490.60
rand-8-20-5	20	18.00	52.58	41.00	**20**	**7.00**	**20**	**7.00**	**20**	6,549.05
ukVg	45	21.20	51.91	425.78	**45**	**56.22**	**45**	61.11	**45**	61,197.53
lexVg	40	21.35	51.92	427.00	**40**	**56.50**	**40**	60.50	**40**	60,771.28
wordsVg	40	21.53	52.32	413.50	**40**	**54.75**	**40**	57.50	40	58,599.68
Tally	*1617*				*1579*		*1566*		*879*	

Appendix B

The Solution Cover Problem is in NP-Complete

We prove in this appendix that finding the minimum number of solutions that cover all the tuples of a minimal CSP is $\mathcal{NP}$-hard.

B.1 Introduction

Given a CSP with a set of constraints $\mathcal{C}$, we want to verify that every tuple in every relation R_i defining a constraint $C_i \in \mathcal{C}$ is *covered* by a solution to the CSP. A tuple is covered by a solution if the projection of the solution on the scope of the tuple equals the tuple. The verification can be done with a subset of solutions that cover all the tuples. *Minimum Solution Cover* problem is the problem of finding the smallest subset of solutions that cover all the tuples, and it is in $\mathcal{NP}$-Hard. The decision problem of finding a subset of solutions of size k or less solutions such that every tuple is covered by a solution from this set is in $\mathcal{NP}$-Complete. We reduce the set cover problem to solution cover by mapping the subsets to solutions and the elements to tuples.

By proving that the solution cover problem is in $\mathcal{NP}$-Hard, we establish the hardness of finding the minimum number of solutions necessary to compute the minimal constraint network of a constraint satisfaction problem.

B.2 Constraint Satisfaction Problem (CSP)

A Constraint Satisfaction Problem (CSP) is defined by $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where:

1. $\mathcal{X} = \{A, B, \ldots\}$ is a set of variables.
2. $\mathcal{D} = \{D_A, D_B, \ldots\}$ is the set of finite domains, where D_A, the domain of variable A, is a set of values that can be assigned to A.
3. $\mathcal{C} = \{C_1, C_2 \ldots, C_n\}$ is a set of constraints restricting the allowed combination of values to variables in its scope, denoted $scope(C_i)$, and is defined by the relation R_i. The scope is the set of variables to which the constraint applies and the relation is a subset of the Cartesian product of the domains of the variables in the scope of C_i.

A *solution* to a CSP is an assignment of one value to each variable such that all constraints are simultaneously satisfied. Solving a CSP corresponds to finding a solution, which is a satisfiability problem, or finding all solutions, which is a counting problem. In general, the satisfiability is $\mathcal{NP}$-complete and the counting problem is #P.

B.3 The Solution Cover Problem (SolCP)

Given a CSP, the solution cover problem (SolCP) is to answer if there is a solution cover (SolC) of size less or equal to k, i.e. a subset of solutions to the CSP of size k

or less such that every tuple is covered by at least one of the solutions. A tuple is covered by a solution if the projection of the solution on the scope of the tuple equals the tuple. We prove that this problem is $\mathcal{N}P$-Complete by constructing a polynomial time transformation from the set cover problem (SCP).

The minimum solution cover is an optimization problem, where we want to find the minimum SolC. It is in $\mathcal{N}P$-Hard.

B.4 Proof of $\mathcal{N}P$-Completeness

We first show that SolCP is in $\mathcal{N}P$, then we choose the set cover problem (SCP) and construct a polynomial time transformation from SCP to SolCP.

B.4.1 SolCP is in $\mathcal{N}P$

Given a set of solutions of size k, we can check each tuple against this set to verify that the tuple is covered by at least one of the solutions.

Theorem 20 *The* SolCP *is in* $\mathcal{N}P$.

Proof: Given a set of solutions $\mathcal{S}$ of size k to the CSP, we can verify in polynomial time that all the tuples can be covered by the solutions in $\mathcal{S}$. It is sufficient to find for each tuple $\tau \in R$, a solution $s \in \mathcal{S}$ such that: the combination of values given by τ is the projection of s on the variables in $scope(R)$.

Each tuple can be checked against a solution in $\mathcal{O}(|\mathcal{X})|$. Therefore, the solution can be verified in $\mathcal{O}(tk \cdot |\mathcal{X})|)$, where t is the total number of tuples in all the relations. □

B.4.2 The set cover problem (SCP) is in $\mathcal{NP}$-Complete

Consider the Set Cover problem (SCP) $(\mathcal{U}, \mathcal{S})$ with a finite set of elements $\mathcal{U}$ and a collection $\mathcal{S}$ of subsets of $\mathcal{U}$. Is there a set cover (SC) of $\mathcal{S}$ of size k or less, i.e. $SC \subseteq \mathcal{S}$ with $|SC| \leq k$ such that every element in $\mathcal{U}$ belongs to at least one member of SC? SCP is in $\mathcal{NP}$-Complete [Garey and Johnson, 1979].

B.4.3 Polynomial transformation from SCP to SolCP

We construct a polynomial transformation from SCP to SolCP such that a SC of size k exists iff a SolC of size $(2 \cdot |\mathcal{S}| + k)$ exists.

Given a SCP $(\mathcal{U}, \mathcal{S})$, we construct the CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ corresponding to the SCP. Each CSP variable corresponds to an element in $\mathcal{U}$, and each solution to the CSP corresponds to a subset in $\mathcal{S}$. Moreover, the constraints correspond to the elements in $\mathcal{U}$, and the tuples correspond to the membership of the elements in the subsets in $\mathcal{S}$. In addition, we add an *umbrella* constraint to restrict the solutions to the subsets in $\mathcal{S}$.

The CSP has additional variables and tuples in the relations to help the construction which are detailed below.

B.4.3.1 Variables

We construct a CSP variable for each element in $\mathcal{U}$, in addition to *identifier* variables. Hence, for clarity, we partition the variables into two sets X_U and X_I, the former corresponding to the set of elements in $\mathcal{U}$ and the latter containing the identifier variables:

1. X_U: each element corresponds to an element in $\mathcal{U}$.

2. X_I: the elements in this set are used to identify the tuples in the relations.

$$\mathcal{X} = X_U \cup X_I$$

$$X_U = \{A | A \in \mathcal{U}\}$$

$$X_I = \{I_A | A \in \mathcal{U}\}$$

B.4.3.2 Constraints

The set of constraints is composed of an *umbrella constraint* C_0 and a set of constraints, one for each element in $\mathcal{U}$ called *element constraint*.

For each element A in $\mathcal{U}$ we have a constraint C_A. The scope of C_A is binary with the variables A and I_A. Relation R_A defines C_A. By default, a tuple with value x_A assigned to A is added to R_A called *element tuple*. Also, for each occurrence of A in a subset in $\mathcal{S}$, an additional element tuple is added. Thus, multiple tuples in R_A may have the value x_A for variable A. The variable I_A in the scope of C_A is used to give a unique identity to each tuple.

In addition, *helper tuples* are added to R_A that assign a numerical value to A. We have a helper tuple in R_A for each subset in $\mathcal{S}$.

The element constraints do not have any common variable in their scopes. To map each solution to a subset in $\mathcal{S}$ (in addition to the empty set), the umbrella constraint C_0 is added.

The scope of C_0 has all the variables in $\mathcal{U}$. Each subset in $\mathcal{S}$ is *encoded* as a tuple in R_0. These tuples are called *subset tuples*. The subset tuple that encodes the subset S_i assigns value x_A to the variable A if $A \in S_i$ and the numerical value i otherwise. Moreover, for every subset in $\mathcal{S}$, an extra tuple is added to R_0 called *empty-set tuple*. The extra tuples generate solutions which correspond to the empty.

The constraints are:

$$\mathcal{C} = \{C_o\} \cup \{C_A | \forall A \in \mathcal{U}\}$$

With the scopes:

$$scope(C_o) = \langle A | \forall A \in \mathcal{U} \rangle$$
$$scope(C_A) = \langle I_A, A \rangle, A \in \mathcal{U}$$

The scopes are given as ordered sequences to allow referring to a variable at given position in the scope. The relations are defined as:

$$R_0 = \Big\{\langle 1, \ldots, 1\rangle, \langle 2, \ldots, 2\rangle, \ldots, \langle |\mathcal{S}|, \ldots |\mathcal{S}|\rangle\Big\} \cup \bigcup_{\forall S_i \in \mathcal{S}} \Big\{\Big\langle f(S_i, 1), \ldots, f\Big(S_i, |scope(C_0)|\Big)\Big\rangle\Big\}$$
$$R_A = \Big\{\langle 1, 1\rangle \ldots, \langle |\mathcal{S}|, |\mathcal{S}|\rangle\Big\} \cup \Big\{\big\langle |\mathcal{S}| + 1, x_A\big\rangle \ldots, \big\langle |\mathcal{S}| + \alpha_A + 1, x_A\big\rangle\Big\}, \forall A \in \mathcal{U}$$

Where:

$$\alpha_A = \big|\{S' | A \in S', S' \in \mathcal{S}\}\big|$$
$$f(S_i, j) = \begin{cases} x_A & \text{if } A \in S_i, \text{ where } A = scope(C_0)[j] \\ i & scope(C_0)[j] \notin S_i \end{cases}$$

B.4.3.3 Domains

The domain of each variable $A \in X_U$ has a unique value for each subset in $\mathcal{S}$, in addition to the element 'x_A'. Hence, the domain of each variable in X_U has numbers from 1 to $|\mathcal{S}|$ and an extra value x_A. The domain of an identifier variable $I_A \in X_I$

has a value for each tuple in R_A. Hence, the domains are:

$$\mathcal{D} = D_U \cup D_I$$
$$D_U = \left\{ D_A \middle| D_A = \{1 \ldots, |\mathcal{S}|\} \cup \{x_A\}, \forall A \in X_U \right\}$$
$$D_I = \left\{ D_{I_A} \middle| D_{I_A} = \{1 \ldots, (|\mathcal{S}| + \alpha_A + 1)\}, \forall I_A \in X_I \right\}$$

B.5 Transformation Example from SCP to SolCP

In this section we present an example demonstrating how SCP is transformed to a SolCP. Consider the following set-cover problem $(\mathcal{U}, \mathcal{S})$:

1. $\mathcal{U} = \{A, B, C, D, E, F\}$
2. $\mathcal{S} = \left\{ \{A, B, C, E\}, \{C, D, E, F\}, \{A, D\}, \{B, E\}, \{C, F\} \right\}$

The transformation of $(\mathcal{U}, \mathcal{S})$ to the CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ is as follows:

1. $\mathcal{X} = \{A, B, C, D, E, F, I_A, I_B, I_C, I_D, I_E, I_F\}$
2. $\mathcal{D} = \{D_A, D_B, D_C, D_D, D_E, D_F, D_{I_A}, D_{I_B}, D_{I_C}, D_{I_D}, D_{I_E}, D_{I_F}\}$

   $D_A = \{1, 2, 3, 4, 5, X_A\}$

   $D_B = \{1, 2, 3, 4, 5, X_B\}$

   $D_C = \{1, 2, 3, 4, 5, X_C\}$

   $D_D = \{1, 2, 3, 4, 5, X_D\}$

   $D_E = \{1, 2, 3, 4, 5, X_E\}$

   $D_F = \{1, 2, 3, 4, 5, X_F\}$

   $D_{I_A} = \{1, 2, 3, 4, 5, 6, 7, 8\}$

   $D_{I_B} = \{1, 2, 3, 4, 5, 6, 7, 8\}$

   $D_{I_C} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$D_{I_D} = \{1, 2, 3, 4, 5, 6, 7, 8\}$

$D_{I_E} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$D_{I_F} = \{1, 2, 3, 4, 5, 6, 7, 8\}$

3. $\mathcal{C} = \{C_o, C_A, C_B, C_C, C_D, C_E, C_F\}$

   $scope(C_o) = \langle A, B, C, D, E, F \rangle$

   $scope(C_A) = \langle I_A, A \rangle$

   $scope(C_B) = \langle I_B, B \rangle$

   $scope(C_C) = \langle I_C, C \rangle$

   $scope(C_D) = \langle I_D, D \rangle$

   $scope(C_E) = \langle I_E, E \rangle$

   $scope(C_F) = \langle I_F, F \rangle$

The relations are shown in Tables B.1 and B.2. The first five tuples in R_0 are the empty-set tuples and the rest are the subset tuples. In all of the relations R_A, R_B, R_C, R_D, R_E, and R_F, the first five tuples are the helper tuples, and the rest are the element tuples.

Figure B.1 shows all the tuples in the CSP represented by dots and the SolC represented by lines. Each row corresponds to a relation. The first row is for the umbrella relation, where the white dots are the empty-set tuples and the black dots are the subset tuples. In the remaining rows, the white dots are the helper tuples, and the black dots are the element tuples. Each line going from a dot in the top row to a dot in the bottom row is a solution. The only valid substitution of tuples in a solution can be obtained by substituting one black dot for another in the same row.

R_0					
A	B	C	D	E	F
1	1	1	1	1	1
2	2	2	2	2	2
3	3	3	3	3	3
4	4	4	4	4	4
5	5	5	5	5	5
X_A	X_B	X_C	1	X_E	1
2	2	X_C	X_D	X_E	X_F
X_A	3	3	X_D	3	3
4	X_B	4	4	X_E	4
5	5	X_C	5	5	X_F

Table B.1: The umbrella relation with the empty set and subset tuples.

R_A		R_B		R_C		R_D		R_E		R_F	
I_A	A	I_B	B	I_C	C	I_D	D	I_E	E	I_F	F
1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5	5	5	5	5
6	X_A	6	X_B	6	X_C	6	X_D	6	X_E	6	X_F
7	X_A	7	X_B	7	X_C	7	X_D	7	X_E	7	X_F
8	X_A	8	X_B	8	X_C	8	X_D	8	X_E	8	X_F
				9	X_C			9	X_E		

Table B.2: The element relations with the helper and element tuples.

For each tuple τ_i in R_0, a single solution (shown in solid lines) is necessary and sufficient to cover τ_i and all but one tuple in every element relation. This is because every element relation R_A has one element tuple more than the number of subsets that element A belongs to in $\mathcal{S}$. Thus, additional solutions are necessary to cover the rest of the uncovered tuples which are shown in dashed and dotted lines. The additional solutions correspond to the solution to the SCP.

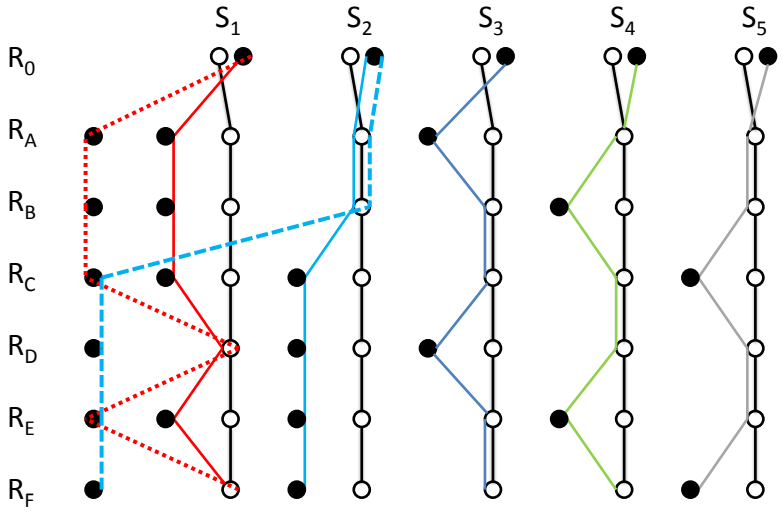

Figure B.1: A solution subset of size $(|\mathcal{S}| * 2) + k = 12$ that covers all the tuples.

B.6 Proof of the Polynomial Transformation

We first prove that the reduction can be done in polynomial time in the size of the SCP. Then we prove that a SolC of size $(2s + k)$ exists iff a SC of size k exists.

Theorem 21 *The reduction requires Polynomial time and space.*

Proof: Given a SCP $(\mathcal{U}, \mathcal{S})$, transforming it to the corresponding CSP takes polynomial time and space. Let $e = |\mathcal{U}|$ and $s = |\mathcal{S}|$.

- **The Variables:** The CSP has $2e$ variables and are generated in $\mathcal{O}(e)$.
- **The Domains:** A variable can have a maximum domain size of $2s$. The maximum value can be reached for the domain of a variable that enumerates the element tuples for the element that appears in every subset. Therefore, the domains can be constructed in $\mathcal{O}(es)$.
- **Constraints:** The CSP has $e + 1$ constraints. The umbrella constraint has $2s$ tuples, and the size of each tuple is e. A single scan of the set $\mathcal{S}$ is enough to

construct this constraint in $\mathcal{O}(es)$.

The element constraints can be constructed by scanning each subset in $\mathcal{S}$ once, and adding a tuple for each member element. Each element constraint has additional s tuples. Thus, all the element constraints can be constructed in $\mathcal{O}(es)$ time. Therefore, the construction of all the constraints can be done in $\mathcal{O}(es)$. □

Theorem 22 *A SC of size k exists iff a SolC of size $(2s + k)$ exists.*

Proof: Given a SC of size k, we can find a SolC of size $(2s + k)$. We have four categories of tuples:

- s empty-set tuples in the umbrella relation.
- s subset tuples in the umbrella relation.
- s helper tuples in each element relation and es in total.
- $\alpha_A + 1$ element tuples in each element relation R_A and $\sum_{\forall A \in \mathcal{X}} (\alpha_A + 1)$ in total.

Each empty-set tuple can only be extended to a solution that covers a single helper tuple in every element relation. Note that each helper tuple in each element relation only matches one empty-set tuple. Since there are s empty-set tuples in the umbrella relation and s helper tuples in each element relation, with s solutions, all the empty set and helper tuples will be covered.

In an element relation R_A, each element tuple requires a solution with one of the subset tuples t_i in the umbrella relation such that t_i corresponds to a subset S_i, and $A \in S_i$. There are α_A such tuples in the umbrella-relation. Any element tuple in R_A can be matched with any such t_i in the umbrella relation without conflicting with the choice of other tuples in the other element relations. Therefore, α_A element tuples in

R_A can be covered by α_A solutions with α_A different subset tuples that correspond to the subsets that A belongs to.

There are $\alpha_A + 1$ element tuples in each element relation; therefore, all but one element tuple will be covered in each element relation with the s solutions. We need to cover a single element-tuple in each element relation with only k solutions.

There are total of k subsets in the solution to SCP that cover all the elements in $\mathcal{U}$. Thus, additional k solutions to the CSP, each extending a subset tuple that corresponds to one of the k subsets, will cover at least one element tuple in each element relation. Therefore, a SolC of size $2s + k$ exists.

We now prove that given a SolC of size $(2s + k)$, we can find a SC of size k. We will proof by construction. We will show that the tuples covered by the $2s$ solutions necessarily leave uncovered a single element tuple from each element relation. Since all the tuples are covered by the $(2s + k)$ solutions, the remaining k solutions necessarily cover the remaining tuples. Consequently, k subsets that correspond to the k different subset tuples in the k solutions, cover all the elements in $\mathcal{U}$.

s solutions are necessary to cover the s empty-set tuples of the umbrella relation. Each empty-set tuple matches exactly one helper tuple in each element relation. Hence, all the empty set and helper tuples will be covered with s solutions.

When an element A belongs to two subsets S_i and S_j, two subset tuples t_i and t_j in R_0 corresponding to S_i and S_j respectively, match any element tuple in R_A. Next we argue that matching t_i and t_j to two different element tuples will always lead to no fewer covered tuples than matching the same element tuple to t_i and t_j.

Proposition 10 *When two distinct subset tuples t_i and t_j in R_0 match two distinct element tuples in the same element relation R_A, it is safe to match t_i to one and t_j to another element tuple.*

Let two solutions Sol_1 and Sol_2 corresponding to the subsets S_i and S_j, cover two distinct subset tuples t_i and t_j respectively such that element $A \in S_i$ and $A \in S_j$. Consider two valid cases:

1. The two solutions cover two different element tuples in R_A.

2. The two solutions cover the same element tuple in R_A.

In both cases, the same tuples can be covered with the other solutions except for the one element tuple in R_A, which may or may not be covered in case 2.

Hence, whenever we have to choose between case (1) and case (2), and we choose case (1), that is to cover two distinct element tuples in the same element relation with two distinct solutions, we will be guaranteed to cover with the rest of the solutions all the tuples that would be covered if we choose case (2). Therefore, it is safe to cover two distinct element tuples whenever we can. □

s solutions are necessary to cover each of the s subset tuples in R_0. Each subset tuple corresponding to the subset S_i only matches the element tuples in each element relation whose corresponding element is in S_i. Thus, the s solutions with the s subset tuples will cover $\sum_{\forall A \in \mathcal{X}} \alpha_A$ element tuples if we choose the case (2) in Proposition 10.

This is the maximum number of element tuples that can be covered with $2s$ solutions. Thus, with the $2s$ solutions necessary to cover all the tuples in the umbrella relation, we cover all the helper tuples and $\sum_{\forall A \in \mathcal{X}} \alpha_A$ element tuples. Since there are $\sum_{\forall A \in \mathcal{X}} (\alpha_A + 1)$ element tuples in total, e tuples are left to be covered with the k remaining solutions. Moreover, the e remaining element tuples are distributed with one in each element relation.

Since we have a SolC of size $2s + k$, the remaining k solutions necessarily cover the remaining uncovered tuples. The remaining tuples are all element tuples; hence, only

subset tuples can be matched. Since only k solutions are available, k subset tuples extend to k solutions to cover the remaining element tuples. Note that two solutions with the same subset tuple cannot cover any more of the uncovered element-tuples than a single solution will, with the same subset tuple. Each subset tuple used in the k solutions corresponds to a subset in $\mathcal{S}$, and the covered element tuples correspond to the elements in the subset. Therefore, the subsets corresponding to the k subset tuples are the subsets that cover all the elements in $\mathcal{U}$. □

Appendix C

Proofs of Main Theorems

C.1 Proofs from Section 3.2

Theorem 1 *If a network is R(∗,m)C, domain filtering by GAC cannot enable further constraint filtering by R(∗,m)C.*

Proof: The proof is by contradiction. Recall that a CSP is GAC iff for every constraint, any value in the domain of any variable in the scope of the constraint can be extended to a tuple satisfying the constraint. Assume that filtering the domains with GAC after enforcing R(∗,m)C removes value x from the domain of variable V_i. Then, there exists a relation R_a that applies to V_i where the value x for V_i does not appear in any tuple in R_a. For GAC to enable further constraint filtering by R(∗,m)C, there must exist at least one constraint R_b that applies to V_i and the value x for V_i appears in some tuple in R_b. Thus, there must be a tuple in R_b that cannot be extended to a tuple in R_a, which yields a contradiction because the problem is R(∗,m)C. □

Theorem 2 *RmC is strictly stronger than R(∗,m)C.*

Proof: Consider a CSP $\mathcal{P}$ and let $\mathcal{P}_{rmc}$ and $\mathcal{P}_{r*mc}$ be the problems obtained after enforcing RmC and R(∗,m)C on $\mathcal{P}$, respectively. Consider a partial assignment τ over some of the variables of $\mathcal{P}$, $scope(\tau)$, that is consistent with the constraints of $\mathcal{P}_{rmc}$. We prove that τ must necessarily be consistent with the constraints in $\mathcal{P}_{r*mc}$. Assume that τ is not consistent with the constraints in $\mathcal{P}_{r*mc}$. Thus, there must be at least one relation R_{x*} in $\mathcal{P}_{r*mc}$ s.t. $\tau \notin \pi_{scope(\tau)}(R_{x*})$. For every relation R in $\mathcal{P}$ there is a relation in $\mathcal{P}_{r*mc}$ and another one in $\mathcal{P}_{rmc}$ with the same scope as R. $\mathcal{P}_{r*mc}$ does not have any additional relations but $\mathcal{P}_{rmc}$ does. Thus, $\mathcal{P}_{rmc}$ must have a relation R_x s.t. $scope(R_{x*})$=$scope(R_x)$. Since τ is a consistent partial solution in $\mathcal{P}_{rmc}$, then $\tau \in \pi_{scope(\tau)}(R_x)$. $\tau \in \pi_{scope(\tau)}(R_x)$ and $\tau \notin \pi_{scope(\tau)}(R_{x*})$ is impossible because joining more relations of $\mathcal{P}_{rmc}$ and projecting them on the same scope cannot possibly introduce more tuples. Thus, we reach a contradiction and RmC is stronger than R(∗,m)C.

Below, we provide an example that is R(∗,m)C but not RmC. Let $\mathcal{P}$ be the following Boolean CSP with the four variables V_1, V_2, V_3, and V_4 and the four constraints: $C_{V_1,V_2} = C_{V_2,V_3} = C_{V_3,V_4} = C_{V_4,V_1} = \{\langle 0,0\rangle, \langle 1,1\rangle\}$. Let $\mathcal{P}_{rmc}$ and $\mathcal{P}_{r*mc}$ be the problems after RmC and R(∗,m)C are enforced on $\mathcal{P}$, respectively. The partial assignment $\langle (V_1,0), (V_3,1)\rangle$ is consistent in $\mathcal{P}_{r*mc}$ because $\mathcal{P}$ has no constraint between V_1 and V_3 and by definition, R(∗,m)C does not add new constraints. However, this partial assignment violates the constraint $C_{V_1,V_3} = \{\langle 0,0\rangle, \langle 1,1\rangle\}$ which is added in $\mathcal{P}_{rmc}$ by RmC. Thus, RmC is strictly stronger than R(∗,m)C. □

C.2 Proofs from Section 3.3

Theorem 5 *$\forall a, b \in \mathbb{N}$ where $a<b\leq|\mathcal{C}|$,* wR(∗,b)C *is strictly stronger than* wR(∗,a)C *on the same connected minimal dual graph of the CSP.*

Proof: Let Φ_a and Φ_b be the set of combinations of wR(∗,a)C and wR(∗,b)C, respectively. For every $\varphi_a \in \Phi_a$ there exists $\varphi_b \in \Phi_b$ such that $\varphi_a \subset \varphi_b$. *wR(∗,b)C* is stronger than *wR(∗,a)C*.

Consider the Boolean CSP $\mathcal{P}_e$ with the three variables V_1, V_2, and V_3 and the three constraints: $C_{V_1,V_2} = C_{V_2,V_3} = C_{V_1,V_3} = \{\langle 0,1\rangle, \langle 1,0\rangle\}$. Clearly, $\mathcal{P}_e$ is wR(∗,2)C but not wR(∗,3)C. □

Theorem 6 *$\forall m>2$,* R(∗,m)C *is strictly stronger than* wR(∗,m)C *on any connected minimal dual graph of the CSP.*

Proof: Every combination of relations considered by wR(∗,m)C is also considered by R(∗,m)C. Hence, R(∗,m)C is stronger than wR(∗,m)C.

Assume that wR(∗,m)C is stronger than R(∗,m)C and that the CSP of Figure 3.2 is inconsistent because there is no assignment for the variables A, B, D that simultaneously satisfies relations $\{R_1, R_2, R_3\}$. For example, assume that $\pi_{AB}(R_1) = \pi_{BC}(R_2) = \{\langle 1,1\rangle, \langle 0,0\rangle\}$, and $\pi_{AC}(R_3) = \{\langle 0,1\rangle, \langle 1,0\rangle\}$. For m=3, the combination $\{R_1, R_2, R_3\}$ considered by R(∗,3)C uncovers the inconsistency. However, this combination is not considered by wR(∗,m)C on the minimal dual graph obtained from removing the two dashed-line edges because the combination induces a disconnected sub-graph of that minimal dual graph. Therefore, wR(∗,m)C fails to uncover the inconsistency uncovered by R(∗,m)C. □

C.3 Proofs from Section 4.3.2

Theorem 8 *Given a CSP, the problem that answers the following question is $\mathcal{NP}$-Complete: is there a set of at most k solutions such that every tuple in every relation of the minimal CSP appears in at least one solution?*

Proof sketch. We reduce Minimum Set Cover [Garey and Johnson, 1979] to this problem in polynomial time. Given a collection $\mathcal{C}$ of subsets of a finite set $\mathcal{S}$ and a positive integer k, a set cover of size k or less exists iff a set of at most $(2 \cdot |\mathcal{S}| + k)$ solutions exists. The reduction is accomplished by constructing a CSP with a variable for each element in $\mathcal{S}$, and domains and relations to have a solution corresponding to each subset in $\mathcal{C}$. The details of the construction are given in Appendix B. □

C.4 Proofs from Section 5.2.2

Theorem 9 *R(∗,m)C is strictly stronger than cl-R(∗,m)C.*

Proof: Every connected combination of relations in a cluster is considered by R(∗,m)C. However, some connected combinations of relations in R(∗,m)C are not necessarily considered by cl-R(∗,m)C. This situation arises when a relation R_i is in one cluster, and another relation R_j is in the neighboring cluster: cl-R(∗,m)C will not consider them together in a combination even if they share a variable. Indeed, in the case of cl-R(∗,m)C, the transfer of information between clusters is through the domains of the variables. For example, consider a problem that has constraints R_i and R_j, such that R_i and R_j are not in the same cluster and $scope(R_i) \cap scope(R_j) = \{A, B\}$. Let R_i be the equality constraint and R_j be the all different constraint. The inconsistency is detected by R(∗,2)C but not by cl-R(∗,2)C. □

Theorem 10 *cl-R(∗,m)C and maxRPWC are not comparable for $m \geq 2$.*

Proof: Theorem 3 guarantees that that R(∗,2)C is strictly stronger than *maxRPWC*. Thus, cl-R(∗,m)C is strictly stronger than *maxRPWC* within a cluster. However, if two constraints that have more than one common variable in their scopes are not in

the same cluster, then cl-R($*$,m)C will not guarantee the requirements of $maxRPWC$. Namely, cl-R($*$,m)C will not check if a tuple in one constraint has a matching tuple in the other constraint. Therefore, cl-R($*$,m)C and $maxRPWC$ are not comparable. □

C.5 Proofs from Section 6.3

Theorem 11 *R(∗,2)C and cl+proj-R(∗,2)C are equivalent.*

Proof: We show that for any combination of two constraints, R($*$,2)C and cl+proj-R($*$,2)C are equivalent, and conclude that R($*$,2)C and cl+proj-R($*$,2)C are equivalent on the whole problem. Consider s the set of variables in the scope of two constraints R_i and R_j: $s = scope(R_i) \cap scope(R_j)$. Given a partial assignment τ to the variables in s, by the definition of R($*$,2)C, $\pi_s(R_i) = \pi_s(R_j)$. If R_i and R_j are in the same cluster, cl+proj-R($*$,2)C is equivalent to R($*$,2)C.

$\pi_s(R_i) = \pi_s(R_j)$ is true when R_i and R_j are in different clusters. Consider two clusters C_i and C_j, such that $R_i \in \psi(C_i)$ and $R_j \in \psi(C_j)$. First assume that C_i and C_j are adjacent. By the definition of projected constraints, there must be a constraint $R'_i \in sep(C_i, C_j)$, such that $R'_i = \pi_{\chi(C_i) \cap \chi(C_j)} R_i$. Thus, $\pi_s(R_i) = \pi_s(R'_i)$ and $\pi_s(R'_i) = \pi_s(R_j)$. Therefore, $\pi_{scope(\tau)}(R_i) = \pi_{scope(\tau)}(R_j)$.

Now assume that C_i and C_j are not adjacent. By the definition of tree decomposition, the variables in $scope(R_i) \cap scope(R_j)$ appear in every cluster C_k on the path from C_i to C_j. Therefore, there must exist some constraint R_k in every cluster C_k such that $scope(R_i) \cap scope(R_j) \subseteq scope(R_k)$, and consequently $\pi_s(R_i) = \pi_s(R_k)$, and $\pi_s(R_k) = \cdots = \pi_s(R_j)$. Therefore, $\pi_s(R_i) = \pi_s(R_j)$. □

Theorem 12 *cl+proj-R(∗,2)C and cl+bin-R(∗,2)C are equivalent.*

Proof: Consider a binary constraint R, $scope(R) = \{A, B\}$. Initially, R will have all the allowed tuples, i.e., the cross product of the domains of A and B. The intersection of $scope(R)$ with the scope of another constraint can either be $\{A\}$ or $\{B\}$. Without loss of generality, let tuples in R be removed after revising R with R_A, such that variable $A \in scope(R_A)$, and R_A has no tuples for some value 'x' for A. R can propagate consistency in two cases, but in both cases, R is not necessary.

In the first case, tuples are deleted in some other relation R'_A, which has A in its scope, after revising it with R. In this case, the same tuples in R'_A are deleted when R_A and R'_A are revised. Therefore, the same result can be obtained without R.

In the second case, tuples are deleted in some relation R'_B, $B \in scope(R'_B)$, after revising it with R. We will show that this case happens only when the inconsistency of the problem is detected. In order to delete tuples in R'_B by revising it with R, there must be some value 'y' for B, such that R has no tuples with that value for B. R will lose tuples only when revised with R_A. In order for R to lose all tuples with value 'y' for B, R_A must have no tuples, in which case the problem will be inconsistent irrespective of R. Also, R is not useful for propagating messages across clusters, because the scope of R can intersect with other constraints' scopes in at most one variable. Consequently, the message that R can pass across clusters, can be passed through the domains of the separator variables. □

Theorem 13 *cl+bin-R(∗,m)C is strictly stronger than cl+proj-R(∗,m)C for $m \geq 2$.*

Proof: cl+bin-R(∗,m)C is as strong as cl+proj-R(∗,m)C, since it processes all the combinations of relations that cl+proj-R(∗,m)C does. In the next example, cl+proj-R(∗,m)C holds but not cl+bin-R(∗,m)C. Consider four variables $\{A, B, C, D\}$ in

the separator of a cluster, and the constraints $\{R_{AB}, R_{BD}, R_{AC}, R_{CD}\}$, where R_{AB}, R_{BD} and R_{AC} are equality constraints, and R_{CD} is the all different constraint. The subscripts of the constraints' names indicate their scopes. This problem is clearly cl+proj-R(∗,3)C.

The constraint R_{AD} will be added in the case of cl+bin-R(∗,m)C. After processing the combination $\{R_{AB}, R_{BD}, R_{AD}\}$, R_{AD} will only allow equal values for A and D. However, when relational consistency is enforced on the combination $\{R_{AC}, R_{CD}, R_{AD}\}$, R_{AD} will only allow different values for A and D. Therefore, cl+bin-R(∗,m)C does not hold. □

Theorem 14 *R(∗,3)C and cl+proj-R(∗,3)C are equivalent.*

Proof: It is first necessary to show that if the dual graph induced by three constraints of a combination is acyclic, then R(∗,3)C and cl+proj-R(∗,3)C are equivalent. Second, it is necessary to show that for every cycle of three nodes in the dual-graph, there is an equivalent set of three constraints, which occur in the same cluster. Then it can be shown that R(∗,3)C and cl+proj-R(∗,3)C are equivalent.

Given a combination of three constraints, if the dual graph induced by the three constraints is acyclic, then pairwise consistency is sufficient to make the constraints minimal [Janssen *et al.*, 1989]. Because pairwise consistency corresponds to R(∗,2)C, and R(∗,2)C and cl+proj-R(∗,2)C are equivalent, cl+proj-R(∗,2)C is sufficient to make the constraints minimal. Also, cl+proj-R(∗,3)C is sufficient because every pair of constraints considered by cl+proj-R(∗,2)C is also considered by cl+proj-R(∗,3)C.

Now consider the case where the dual graph of each combination of three constraints is not acyclic. For any three constraints R_1, R_2, and R_3, if a cluster C_i exists such that $\{R_1, R_2, R_3\} \subseteq \psi(C_i)$, then cl+proj-R(∗,3)C is equivalent to R(∗,3)C. The case

where none of the two constraints are in one cluster is impossible because it violates the tree decomposition definition. Hence, the only other case to consider is when two clusters C_i and C_j exist, such that $\{R_1, R_2, R_3\} \subseteq \psi(C_i) \cup \psi(C_j)$.

Without loss of generality, assume $scope(R_1) \backslash (scope(R_2) \cup scope(R_3)) \neq \emptyset$, $\{R_2, R_3\} \subseteq \psi(C_i)$ and $R_1 \in \psi(C_j)$. Then, there must exist a constraint $R'_1 = \pi_{\chi(C_i)} scope(R_1)$, $R'_1 \in \psi(C_i)$. Enforcing cl+proj-R(∗,3)C in C_i and then revising R_1 given R'_1 is equivalent to applying R(∗,3)C on the combination of $\{R_1, R_2, R_3\}$. Therefore, R(∗,3)C and cl+proj-R(∗,3)C are equivalent in this case.

Now consider the case where $\{R_2, R_3\} \subseteq \psi(C_i)$, $R_1 \in \psi(C_j)$ and $scope(R_1) \setminus (scope(R_2) \cup scope(R_3)) = \emptyset$. But $scope(R_2) \cup scope(R_3) \subseteq \chi(C_i)$, hence $R_1 \in \chi(C_i)$. Therefore, in this case also cl+proj-R(∗,3)C is equivalent to R(∗,3)C because $\{R_1, R_2, R_3\} \subseteq \psi(C_i)$. □

Theorem 15 *R(∗,m)C is strictly stronger than cl+proj-R(∗,m)C for $m > 3$.*

Proof: R(∗,m)C is stronger than cl+proj-R(∗,m)C, because every combination of constraints considered by cl+proj-R(∗,m)C is also considered by R(∗,m)C. Now consider four constraints R_1, R_2, R_3 and R_4 such that $\{R_1, R_2\} \subseteq \psi(C_i)$ and $R_3, R_4 \notin \psi(C_i)$. Moreover, $\nexists R_x$ such that $scope(R_3) \cap scope(R_4) \subseteq scope(R_x)$. Assume that no partial assignments to $scope(R_3) \cap scope(R_4)$ satisfy the four constraints simultaneously. R(∗,4)C detects the inconsistency but cl+proj-R(∗,4)C does not. Therefore, R(∗,m)C is strictly stronger than cl+proj-R(∗,m)C for $m > 3$. □

Theorem 16 *cl+clq-R(∗,m)C is strictly stronger than cl+bin-R(∗,m)C.*

Proof: Clearly, cl+clq-R($*$,m)C is as strong as cl+bin-R($*$,m)C because for every combination considered by cl+bin-R($*$,m)C, cl+clq-R($*$,m)C either considers the same combination, or considers a different combination, such that the scopes of the constraints are supersets of the constraints in the combination considered by cl+bin-R($*$,m)C. This is because every triangulation edge necessarily appears in some maximal clique. Moreover, clique constraints propagate more information across clusters than binary constraints do. □

Appendix D

Iterative WitnessBTD

In Chapter 7, we presented a recursive algorithm for WitnessBTD, which improves BTD for counting the solutions to a CSP. Here we present the algorithm in iterative form.

The WitnessBTD iterative algorithm searches for a witness solution before proceeding to counting all the solutions to the problem. Thus, it operates in two states: *satisfy* (for searching for a witness) and *countSol* (for counting solutions). The state is represented by *state*, and is set for each cluster. A given cluster is in *satisfy* if the parent cluster is in *satisfy* state. However, a cluster can be in either state if the parent is in *countSol* state.

Algorithm 16 forms the main loop of the algorithm. The while loop in Line 4 repeats as long as the problem has an uninstantiated variable and a variable is chosen to be the current variable. The variable is instantiated if it has values in its domain in Line 5 or backtracks in Line 18.

After instantiating the variable, Propagate is called to propagate the given consistency property. When the problem is consistent with all the variables instantiated in the cluster, the solution count for the cluster is incremented. *consistent* is set to

$false$ to force the algorithm to backtrack when searching for all solutions in Line 11. The next variable is chosen in Line 13 if the problem is consistent and there are more uninstantiated variables or searching for all solutions. Otherwise, the value of the instantiated variable is deleted from the domain in Line 15.

When the domain of the current variable is empty, the algorithm proceeds to backtrack in Line 18. The latest instantiated variable in the current cluster is assigned to be the current variable if it exists. Otherwise, the parent of the current cluster becomes the current cluster. When moved to the parent cluster, the good or nogood is recorded in Line 24.

The state of the search in the previous cluster is preserved if the problem was consistent, and a new variable is chosen from the next cluster in Line 28. ChooseVariable performs depth-first traversal (DFT) of the tree to choose the next cluster with uninstantiated variables. The second parameter to it specifies whether the traversal is backwards or not, to avoid visiting the same cluster again.

When the problem is inconsistent, the subtree rooted at the current cluster is reset, and the value of the current instantiated-variable is deleted in the block in Line 29.

Algorithm 17 performs three main tasks in addition to choosing the next unassigned variable: it maintains the state of the algorithm, checks for goods and nogoods, and computes the solution count. The main loop of the algorithm in Line 1 is for the progression of the DFT until a cluster is found with uninstantiated variables. The block in Line 3 checks if a witness is found for the current assignment in the subtree, and switches the state of the algorithm accordingly. The goods and nogoods are checked in the block of Line 12 before descending into a subtree. The structure $DFTVisited$ maintains the state of the DFT. When all the variables in the subtree of the current cluster are instantiated in Line 24, the DFT backtracks from the current cluster after computing the solution count. Finally, in Line 38, an uninstantiated

variable chosen from the cluster according to a specified variable ordering heuristic within the cluster and returned. NIL is returned if no uninstantiated variable can be found, which terminates the search.

Next, we describe the attributes and functions used in the two algorithms.

List of attributes:

- $\chi(C)$: Set of variables in the cluster C
- $\mathcal{P}$: The CSP
- *backwards*: The direction of DFT
- $Children(C)$: The children of cluster C
- *countSol*: State of counting solutions
- *consistent*: Indicates if the problem is consistent or should backtrack
- *curCluster*: The current cluster being processed
- $curDom(A)$: The current domain of the variable A
- $curVariable$: The current variable being instantiated
- $DFTVisited(C)$: Indicates if cluster C is visited in the current DFT
- $Domain(A)$: The original domain of the variable A
- $Parent(C)$: The parent cluster of C
- $Reductions(A)$: The set of value reductions in A caused by other instantiations
- $satisfy$: Satisfiability state when searching for the witness solution

- $solutionCount(C)$: The count of solution in cluster C for the current assignment of the separator
- $state(C)$: The state of the cluster C
- $subtreeInstantiated(C)$: Indicates if all the variables in the subtree of C are instantiated
- $value(A)$: The value assigned to the variable A
- $witnessFound$: Indicates if a witness solution is found

List of functions:

- GetGoodSolCount(C): Returns the stored solution count in goods for the current assignment of the separator
- HasGoodSolCount(C): Indicates if the solution count is stored for the current assignment of the separator
- Instantiate(A): Instantiates variable A with the next value it its current domain.
- IsGood(C): Indicates if the current assignment of the separator is a known good
- IsNoGood(C): Indicates if the current assignment of the separator is a known nogood
- LastAssignedVariable(C): Returns the variable assigned most recently in cluster C
- Propagate(A): Propagates the consistency algorithm given the instantiation of the variable A

- RecordGoodsNogoods(C): Records a good if a solution is found, or records a nogood otherwise
- UndoReductions(A): Undoes all the reductions caused by the instantiation of the variable A
- UpdateSubtree(C): Undoes all the instantiations of the variables in the subtree rooted at C

Algorithm 16: An iterative description of *WitnessBTD*.

Input: $\mathcal{P}, root$
Output: Number of solutions in the problem.

```
1  curCluster ← root // current cluster
2  curVariable ← ChooseVariable(curCluster, false) // current variable
3  state[curCluster] = countSol
4  while ∃A ∈ X_P s.t. value[A] =NIL AND curVariable ≠ NIL do
5      if curDom(curVariable) then
6          Instantiate(curVariable)
7          consistent ← Propagate(curVariable, P)
8          if ∄A ∈ χ(curCluster) s.t. value[A] =NIL then
9              if children(curCluster) = ∅ then
10                 solutionCount[curCluster] ← solutionCount[curCluster] + 1
11                 if state[curCluster] = countSol then consistent ← false
12         if consistent AND (∃A ∈ X_P s.t. value[A] =NIL OR
           state[curCluster] = countSol) then
13             curVariable ← ChooseVariable(curCluster, false)
14         else
15             curDom(curVariable)← curDom(curVariable)\value[curVariable]
16     else
17         UndoReductions(curVariable)
18         curDom(curVariable) ← Domain(curVariable)\ Reductions(curVariable)
19         curVariable ← NIL
20         if ∃A ∈ χ(curCluster) s.t. value[A] ≠NIL then
21             curVariable ←LastAssignedVariable(curCluster)
22             curDom(curVariable)← curDom(curVariable)\value[curVariable]
23         else
24             RecordGoodsNogoods(curCluster)
25             consistent ← solutionCount[curCluster] > 0
26             curCluster ← Parent(curCluster)
27             if consistent then
28                 curVariable ← ChooseVariable(curCluster, true)
29             else
30                 if Parent(curCluster) =NIL then state[curCluster] ← countSol
31                 else state[curCluster] ← state[Parent(curCluster)]
32                 UpdateSubtree(curCluster)
33                 curVariable ← LastAssignedVariable(curCluster)
34                 curDom(curVariable)← curDom(curVariable)\value[curVariable]
35 return solutionCount[root]
```

Algorithm 17: ChooseVariable.

Input: $curCluster, backwards$
Output: $curVariable$

```
1  while ∃A ∈ χ(curCluster) s.t. value[A] =NIL do
2      subtreeInstantiated ← true
3      if backwards=false then
4          state[curCluster] ← satisfy
5          if Parent(curCluster)=NIL) OR state[Parent(curCluster)] = countSol
           then
6              witnessFound ← true
7              foreach child ∈ Children(curCluster) do
8                  if IsGood(child) = false then witnessFound ← false; Break
9              if witnessFound then state[curCluster] ← countSol
10         if state[curCluster] ← countSol then
11             foreach child ∈ Children(curCluster) do DFTVisited[child] ← false
12     foreach child ∈ Children(curCluster) do
13         if DFTVisited[child] = false then
14             DFTVisited[child] = true
15             if IsNoGood(child) then solutionCount[child] ← 0; Break
16             if IsGood(child) then
17                 if solutionCount[child] = false OR HasGoodSolCount(child)
                   then
18                     solutionCount[child] ← GetGoodSolCount(child); Continue
19             state[child] ← state[curCluster]
20             curCluster ← child
21             if Children(curCluster) ≠ ∅ OR state[curCluster] = satisfy OR
               ∃A ∈ χ(curCluster), value[A] =NIL then
22                 subtreeAssigned ← false
23             Break
24     if subtreeAssigned then
25         solutionCount ← 0
26         if Children(curCluster) ≠ ∅ then solutionCount ← 1
27         foreach child ∈ Children(curCluster) do
28             solutionCount ← solutionCount × solutionCount[child]
29             DFTVisited[child] ← false
30         solutionCount[curCluster] ← solutionCount[curCluster] + solutionCount
31         if solutionCount[curCluster] = 0 OR state[curCluster] = countSol then
32             curVariable ← LastAssignedVariable(curCluster)
33             curDom(curVariable)← curDom(curVariable)\value[curVariable]
34         else
35             RecordGoodsNogoods(curCluster)
36             curCluster ← Parent(curCluster)
37     else Break
38 return A ∈ χ(curCluster), value[A] =NIL
```

Appendix E

Characteristics of the Benchmark Data

In this appendix, we give the characteristics of the benchmark data used in the empirical evaluations of Chapters 5 and 6. The benchmarks are selected from those used in the CSP Solver Competition.[1] The following tables list the instances in each benchmark, and give the following characteristics:

- **file:** the name of the file
- **#variables:** the number of variables
- **#constraints:** the number of constraints
- **#total tuples:** the total number of tuples in all relations
- **max domain:** the size of the largest domain
- **max arity:** the arity of the constraint with the largest scope

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

- **#clusters:** the number of clusters in the tree decomposition
- **treewidth:** the number of variables in the largest cluster
- **largest sep.:** the number of variables in the largest separator
- **max** $|\psi(cl)|$ **local:** the maximum number of constraints in a cluster without bolstering
- **max** $|\psi(cl)|$ **proj:** the maximum number of constraints in a cluster with the addition of the projection constraints
- **max** $|\psi(cl)|$ **binary:** the maximum number of constraints in a cluster with the addition of the binary constraints
- **max** $|\psi(cl)|$ **clique:** the maximum number of constraints in a cluster with the addition of the clique constraints

The tree decomposition characteristics correspond to the tree decompositions computed using an adaption for non-binary CSPs of the tree-clustering technique [Dechter and Pearl, 1989] by first triangulating the primal graph of the CSP using the min-fill heuristic [Kjærulff, 1990], and then identifying the maximal cliques in the resulting chordal graph using the MaxCliques algorithm [Golumbic, 1980].

The characteristics refer to the original constraints in the problems, except for the last three, which report the numbers of constraints resulted in each bolstering scheme. A '-' is added when the numbers were not computed.

Tables E.1 to E.12 describe the unsatisfiable binary instances, and Tables E.13 to E.16 describe the unsatisfiable non-binary instances. Similarly, Tables E.17 to E.19 describe the unsatisfiable binary instances, and Tables E.20 to E.22 describe the unsatisfiable non-binary instances.

Table E.1: Data characteristics of unsatisfiable binary instances (part 1).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
composed-25-1-2												
composed-25-1-2-0	33	224	17,960	10	2	10	20	18	128	128	172	128
composed-25-1-2-1	33	224	17,960	10	2	11	19	18	119	119	149	119
composed-25-1-2-2	33	224	17,960	10	2	12	19	18	125	125	155	125
composed-25-1-2-3	33	224	17,960	10	2	11	20	17	131	131	164	131
composed-25-1-2-4	33	224	17,960	10	2	12	19	17	126	126	152	126
composed-25-1-2-5	33	224	17,960	10	2	12	19	18	123	123	151	123
composed-25-1-2-6	33	224	17,960	10	2	12	19	18	121	121	150	121
composed-25-1-2-7	33	224	17,960	10	2	12	19	18	119	119	153	119
composed-25-1-2-8	33	224	17,960	10	2	12	19	18	123	123	158	123
composed-25-1-2-9	33	224	17,960	10	2	12	19	18	130	130	154	130
composed-25-1-25												
composed-25-1-25-0	33	247	20,145	10	2	13	21	20	142	142	192	142
composed-25-1-25-1	33	247	20,145	10	2	13	21	20	150	150	184	150
composed-25-1-25-2	33	247	20,145	10	2	13	21	20	141	141	179	141
composed-25-1-25-3	33	247	20,145	10	2	14	20	19	131	131	174	131
composed-25-1-25-4	33	247	20,145	10	2	15	19	18	122	122	152	122
composed-25-1-25-5	33	247	20,145	10	2	14	20	19	137	137	168	137
composed-25-1-25-6	33	247	20,145	10	2	14	20	19	128	128	165	128
composed-25-1-25-7	33	247	20,145	10	2	14	20	19	133	133	179	133
composed-25-1-25-8	33	247	20,145	10	2	14	20	19	133	133	162	133
composed-25-1-25-9	33	247	20,145	10	2	14	20	19	141	141	178	141
composed-25-1-40												
composed-25-1-40-0	33	262	21,570	10	2	12	22	21	140	140	186	140
composed-25-1-40-1	33	262	21,570	10	2	12	22	19	155	155	190	155
composed-25-1-40-2	33	262	21,570	10	2	13	21	20	144	144	182	144
composed-25-1-40-3	33	262	21,570	10	2	14	20	19	126	126	158	126
composed-25-1-40-4	33	262	21,570	10	2	12	22	20	146	146	194	146
composed-25-1-40-5	33	262	21,570	10	2	13	21	20	135	135	171	135
composed-25-1-40-6	33	262	21,570	10	2	12	22	20	146	146	186	146
composed-25-1-40-7	33	262	21,570	10	2	12	22	21	151	151	204	151
composed-25-1-40-8	33	262	21,570	10	2	13	21	19	146	146	183	146
composed-25-1-40-9	33	262	21,570	10	2	12	22	20	150	150	191	150

Table E.2: Data characteristics of unsatisfiable binary instances (part 2).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
composed-25-1-80												
composed-25-1-80-0	33	302	25,370	10	2	9	25	24	186	186	246	-
composed-25-1-80-1	33	302	25,370	10	2	10	24	23	177	177	222	177
composed-25-1-80-2	33	302	25,370	10	2	11	23	22	161	161	215	161
composed-25-1-80-3	33	302	25,370	10	2	9	25	23	190	190	262	190
composed-25-1-80-4	33	302	25,370	10	2	10	24	23	175	175	237	175
composed-25-1-80-5	33	302	25,370	10	2	9	25	24	187	187	251	187
composed-25-1-80-6	33	302	25,370	10	2	9	25	23	184	184	262	184
composed-25-1-80-7	33	302	25,370	10	2	8	26	24	198	198	268	198
composed-25-1-80-8	33	302	25,370	10	2	9	25	23	178	178	248	178
composed-25-1-80-9	33	302	25,370	10	2	9	25	23	173	173	252	173
composed-75-1-2												
composed-75-1-2-0	83	624	51,960	10	2	35	46	45	253	253	659	253
composed-75-1-2-1	83	624	51,960	10	2	34	47	46	267	267	676	267
composed-75-1-2-2	83	624	51,960	10	2	34	47	45	240	240	607	240
composed-75-1-2-3	83	624	51,960	10	2	32	49	48	277	277	767	277
composed-75-1-2-4	83	624	51,960	10	2	35	46	45	242	242	706	242
composed-75-1-2-5	83	624	51,960	10	2	34	47	45	264	264	690	264
composed-75-1-2-6	83	624	51,960	10	2	34	46	45	251	251	721	251
composed-75-1-2-7	83	624	51,960	10	2	32	48	45	279	279	735	279
composed-75-1-2-8	83	624	51,960	10	2	32	49	48	271	271	746	271
composed-75-1-2-9	83	624	51,960	10	2	34	47	46	273	273	728	273
composed-75-1-25												
composed-75-1-25-0	83	647	54,145	10	2	35	48	46	259	259	665	259
composed-75-1-25-1	83	647	54,145	10	2	36	48	46	259	259	668	259
composed-75-1-25-2	83	647	54,145	10	2	36	48	45	252	252	682	252
composed-75-1-25-3	83	647	54,145	10	2	35	49	46	279	279	773	279
composed-75-1-25-4	83	647	54,145	10	2	34	50	48	287	287	785	287
composed-75-1-25-5	83	647	54,145	10	2	35	49	47	282	282	756	282
composed-75-1-25-6	83	647	54,145	10	2	36	48	47	263	263	715	263
composed-75-1-25-7	83	647	54,145	10	2	37	46	45	252	252	580	252
composed-75-1-25-8	83	647	54,145	10	2	34	50	48	277	277	683	277
composed-75-1-25-9	83	647	54,145	10	2	36	48	47	279	279	699	279

Table E.3: Data characteristics of unsatisfiable binary instances (part 3).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
composed-75-1-40												
composed-75-1-40-0	83	662	55,570	10	2	36	47	46	260	260	698	260
composed-75-1-40-1	83	662	55,570	10	2	33	51	49	280	280	702	280
composed-75-1-40-2	83	662	55,570	10	2	34	50	48	266	266	706	266
composed-75-1-40-3	83	662	55,570	10	2	35	49	48	279	279	840	279
composed-75-1-40-4	83	662	55,570	10	2	34	50	45	276	276	700	276
composed-75-1-40-5	83	662	55,570	10	2	33	51	50	302	302	685	302
composed-75-1-40-6	83	662	55,570	10	2	33	51	48	298	298	743	298
composed-75-1-40-7	83	662	55,570	10	2	34	50	49	293	293	796	293
composed-75-1-40-8	83	662	55,570	10	2	32	52	51	304	304	846	304
composed-75-1-40-9	83	662	55,570	10	2	35	49	48	293	293	795	293
composed-75-1-80												
composed-75-1-80-0	83	702	59,370	10	2	29	55	53	336	336	1,109	336
composed-75-1-80-1	83	702	59,370	10	2	30	54	53	325	325	980	325
composed-75-1-80-2	83	702	59,370	10	2	32	52	48	289	289	842	289
composed-75-1-80-3	83	702	59,370	10	2	31	53	52	308	308	936	308
composed-75-1-80-4	83	702	59,370	10	2	32	52	50	291	291	784	291
composed-75-1-80-5	83	702	59,370	10	2	32	51	50	287	287	737	287
composed-75-1-80-6	83	702	59,370	10	2	30	54	51	319	319	899	-
composed-75-1-80-7	83	702	59,370	10	2	31	53	52	313	313	864	-
composed-75-1-80-8	83	702	59,370	10	2	29	55	52	332	332	905	332
composed-75-1-80-9	83	702	59,370	10	2	31	53	52	322	322	1,060	322
graphColoring-hosExtConvert												
abb313GPIA-5	1,557	53,356	1,067,120	5	2	257	121	116	3,759	3,759	4,869	-
abb313GPIA-7	1,557	53,356	2,240,952	7	2	257	121	116	3,759	3,759	4,869	-
abb313GPIA-8	1,557	53,356	2,987,936	8	2	257	121	116	3,759	3,759	4,869	-
abb313GPIA-9	1,557	53,356	3,841,632	9	2	257	121	116	3,759	3,759	4,869	-
ash331GPIA-3	662	4,181	25,086	3	2	318	89	68	143	144	240	94
ash608GPIA-3	1,216	7,844	47,064	3	2	585	122	92	211	211	349	134
ash958GPIA-3	1,916	12,506	75,036	3	2	912	127	97	239	245	420	284
will199GPIA-5	701	6,772	135,440	5	2	287	109	83	596	596	1,191	596
will199GPIA-6	701	6,772	203,160	6	2	287	109	83	596	596	1,191	-
graphColoring-mugExtConvert												
mug100-1-3	100	166	996	3	2	65	4	2	5	5	5	5
mug100-25-3	100	166	996	3	2	65	4	2	5	5	5	5
mug88-1-3	88	146	876	3	2	57	4	2	5	5	5	5
mug88-25-3	88	146	876	3	2	57	4	2	5	5	5	5

Table E.4: Data characteristics of unsatisfiable binary instances (part 4).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
graphColoring-register-mulsolExtConvert												
mulsol-i-1-05	197	3,925	78,500	5	2	67	51	50	1,243	1,243	1,243	1,243
mulsol-i-1-10	197	3,925	353,250	10	2	67	51	50	1,243	1,243	1,243	-
mulsol-i-1-15	197	3,925	824,250	15	2	67	51	50	1,243	1,243	1,243	-
mulsol-i-1-20	197	3,925	1,491,500	20	2	67	51	50	1,243	1,243	1,243	-
mulsol-i-1-25	197	3,925	2,355,000	25	2	67	51	50	1,243	1,243	1,243	-
mulsol-i-1-30	197	3,925	3,414,750	30	2	67	51	50	1,243	1,243	1,243	-
mulsol-i-1-35	197	3,925	4,670,750	35	2	67	51	50	1,243	1,243	1,243	-
mulsol-i-1-40	197	3,925	6,123,000	40	2	67	51	50	1,243	1,243	1,243	-
mulsol-i-1-45	197	3,925	7,771,500	45	2	67	51	50	1,243	1,243	1,243	-
mulsol-i-1-46	197	3,925	8,124,750	46	2	67	51	50	1,243	1,243	1,243	-
mulsol-i-1-47	197	3,925	8,485,850	47	2	67	51	50	1,243	1,243	1,243	-
mulsol-i-1-48	197	3,925	8,854,800	48	2	67	51	50	1,243	1,243	1,243	-
mulsol-i-2-05	188	3,885	77,700	5	2	96	33	31	496	496	496	496
mulsol-i-2-10	188	3,885	349,650	10	2	96	33	31	496	496	496	-
mulsol-i-2-15	188	3,885	815,850	15	2	96	33	31	496	496	496	-
mulsol-i-2-20	188	3,885	1,476,300	20	2	96	33	31	496	496	496	-
mulsol-i-2-25	188	3,885	2,331,000	25	2	96	33	31	496	496	496	-
mulsol-i-2-28	188	3,885	2,937,060	28	2	96	33	31	496	496	496	-
mulsol-i-2-29	188	3,885	3,154,620	29	2	96	33	31	496	496	496	-
mulsol-i-2-30	188	3,885	3,379,950	30	2	96	33	31	496	496	496	-
mulsol-i-3-05	184	3,916	78,320	5	2	97	33	31	496	496	496	496
mulsol-i-3-10	184	3,916	352,440	10	2	97	33	31	496	496	496	-
mulsol-i-3-15	184	3,916	822,360	15	2	97	33	31	496	496	496	-
mulsol-i-3-20	184	3,916	1,488,080	20	2	97	33	31	496	496	496	-
mulsol-i-3-25	184	3,916	2,349,600	25	2	97	33	31	496	496	496	-
mulsol-i-3-28	184	3,916	2,960,496	28	2	97	33	31	496	496	496	-
mulsol-i-3-29	184	3,916	3,179,792	29	2	97	33	31	496	496	496	-
mulsol-i-3-30	184	3,916	3,406,920	30	2	97	33	31	496	496	496	-
mulsol-i-4-05	185	3,946	78,920	5	2	97	33	31	496	496	496	496
mulsol-i-4-10	185	3,946	355,140	10	2	97	33	31	496	496	496	-
mulsol-i-4-15	185	3,946	828,660	15	2	97	33	31	496	496	496	-
mulsol-i-4-20	185	3,946	1,499,480	20	2	97	33	31	496	496	496	-
mulsol-i-4-25	185	3,946	2,367,600	25	2	97	33	31	496	496	496	-
mulsol-i-4-28	185	3,946	2,983,176	28	2	97	33	31	496	496	496	-
mulsol-i-4-29	185	3,946	3,204,152	29	2	97	33	31	496	496	496	-
mulsol-i-4-30	185	3,946	3,433,020	30	2	97	33	31	496	496	496	-
mulsol-i-5-05	186	3,973	79,460	5	2	99	33	31	480	480	480	480

Table E.5: Data characteristics of unsatisfiable binary instances (part 5).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
graphColoring-register-mulsolExtConvert												
mulsol-i-5-10	186	3,973	357,570	10	2	99	33	31	480	480	480	-
mulsol-i-5-15	186	3,973	834,330	15	2	99	33	31	480	480	480	-
mulsol-i-5-20	186	3,973	1,509,740	20	2	99	33	31	480	480	480	-
mulsol-i-5-25	186	3,973	2,383,800	25	2	99	33	31	480	480	480	-
mulsol-i-5-28	186	3,973	3,003,588	28	2	99	33	31	480	480	480	-
mulsol-i-5-29	186	3,973	3,226,076	29	2	99	33	31	480	480	480	-
mulsol-i-5-30	186	3,973	3,456,510	30	2	99	33	31	480	480	480	-
graphColoring-register-zeroinExtConvert												
zeroin-i-1-05	211	4,100	82,000	5	2	60	51	49	1,243	1,243	1,243	1,243
zeroin-i-1-10	211	4,100	369,000	10	2	60	51	49	1,243	1,243	1,243	-
zeroin-i-1-15	211	4,100	861,000	15	2	60	51	49	1,243	1,243	1,243	-
zeroin-i-1-20	211	4,100	1,558,000	20	2	60	51	49	1,243	1,243	1,243	-
zeroin-i-1-25	211	4,100	2,460,000	25	2	60	51	49	1,243	1,243	1,243	-
zeroin-i-1-30	211	4,100	3,567,000	30	2	60	51	49	1,243	1,243	1,243	-
zeroin-i-1-35	211	4,100	4,879,000	35	2	60	51	49	1,243	1,243	1,243	-
zeroin-i-1-40	211	4,100	6,396,000	40	2	60	51	49	1,243	1,243	1,243	-
zeroin-i-1-45	211	4,100	8,118,000	45	2	60	51	49	1,243	1,243	1,243	-
zeroin-i-1-46	211	4,100	8,487,000	46	2	60	51	49	1,243	1,243	1,243	-
zeroin-i-1-47	211	4,100	8,864,200	47	2	60	51	49	1,243	1,243	1,243	-
zeroin-i-1-48	211	4,100	9,249,600	48	2	60	51	49	1,243	1,243	1,243	-
zeroin-i-2-05	211	3,541	70,820	5	2	93	34	30	464	464	464	464
zeroin-i-2-10	211	3,541	318,690	10	2	93	34	30	464	464	464	464
zeroin-i-2-15	211	3,541	743,610	15	2	93	34	30	464	464	464	-
zeroin-i-2-20	211	3,541	1,345,580	20	2	93	34	30	464	464	464	-
zeroin-i-2-25	211	3,541	2,124,600	25	2	93	34	30	464	464	464	-
zeroin-i-2-27	211	3,541	2,485,782	27	2	93	34	30	464	464	464	-
zeroin-i-2-28	211	3,541	2,676,996	28	2	93	34	30	464	464	464	-
zeroin-i-2-29	211	3,541	2,875,292	29	2	93	34	30	464	464	464	-
zeroin-i-3-05	206	3,540	70,800	5	2	93	34	30	464	464	464	464
zeroin-i-3-10	206	3,540	318,600	10	2	93	34	30	464	464	464	464
zeroin-i-3-15	206	3,540	743,400	15	2	93	34	30	464	464	464	-
zeroin-i-3-20	206	3,540	1,345,200	20	2	93	34	30	464	464	464	-
zeroin-i-3-25	206	3,540	2,124,000	25	2	93	34	30	464	464	464	-
zeroin-i-3-27	206	3,540	2,485,080	27	2	93	34	30	464	464	464	-
zeroin-i-3-28	206	3,540	2,676,240	28	2	93	34	30	464	464	464	-
zeroin-i-3-29	206	3,540	2,874,480	29	2	93	34	30	464	464	464	-

Table E.6: Data characteristics of unsatisfiable binary instances (part 6).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
graphColoring-sgb-bookExtConvert												
anna-10	138	493	44,370	10	2	111	13	12	62	62	65	62
anna-5	138	493	9,860	5	2	111	13	12	62	62	65	62
anna-8	138	493	27,608	8	2	111	13	12	62	62	65	62
anna-9	138	493	35,496	9	2	111	13	12	62	62	65	62
david-10	87	406	36,540	10	2	59	14	12	75	75	77	75
david-5	87	406	8,120	5	2	59	14	12	75	75	77	75
david-8	87	406	22,736	8	2	59	14	12	75	75	77	78
david-9	87	406	29,232	9	2	59	14	12	75	75	77	80
homer-10	561	1,628	145,710	10	2	444	32	30	207	207	272	207
homer-11	561	1,628	178,090	11	2	444	32	30	207	207	272	-
homer-12	561	1,628	213,708	12	2	444	32	30	207	207	272	-
homer-5	561	1,628	32,380	5	2	444	32	30	207	207	272	210
homer-8	561	1,628	90,664	8	2	444	32	30	207	207	272	207
huck-10	74	301	26,730	10	2	32	11	6	55	55	55	55
huck-5	74	301	5,940	5	2	32	11	6	55	55	55	55
huck-8	74	301	16,632	8	2	32	11	6	55	55	55	55
huck-9	74	301	21,384	9	2	32	11	6	55	55	55	55
jean-5	80	254	5,080	5	2	51	10	8	45	45	45	45
jean-7	80	254	10,668	7	2	51	10	8	45	45	45	46
jean-8	80	254	14,224	8	2	51	10	8	45	45	45	46
jean-9	80	254	18,288	9	2	51	10	8	45	45	45	46
graphColoring-sgb-gamesExtConvert												
games120-5	120	638	12,760	5	2	65	41	35	83	83	116	83
games120-7	120	638	26,796	7	2	65	41	35	83	83	116	83
games120-8	120	638	35,728	8	2	65	41	35	83	83	116	83

Table E.7: Data characteristics of unsatisfiable binary instances (part 7).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
graphColoring-sgb-milesExtConvert												
miles1000-10	128	3,216	289,440	10	2	59	51	48	1,109	1,109	1,163	1,109
miles1000-15	128	3,216	675,360	15	2	59	51	48	1,109	1,109	1,163	-
miles1000-20	128	3,216	1,222,080	20	2	59	51	48	1,109	1,109	1,163	-
miles1000-25	128	3,216	1,929,600	25	2	59	51	48	1,109	1,109	1,163	-
miles1000-30	128	3,216	2,797,920	30	2	59	51	48	1,109	1,109	1,163	-
miles1000-35	128	3,216	3,827,040	35	2	59	51	48	1,109	1,109	1,163	-
miles1000-38	128	3,216	4,521,696	38	2	59	51	48	1,109	1,109	1,163	-
miles1000-39	128	3,216	4,766,112	39	2	59	51	48	1,109	1,109	1,163	-
miles1000-40	128	3,216	5,016,960	40	2	59	51	48	1,109	1,109	1,163	-
miles1000-41	128	3,216	5,274,240	41	2	59	51	48	1,109	1,109	1,163	-
miles1000-5	128	3,216	64,320	5	2	59	51	48	1,109	1,109	1,163	1,109
miles1500-10	128	5,198	467,820	10	2	40	78	77	2,956	2,956	2,956	-
miles1500-20	128	5,198	1,975,240	20	2	40	78	77	2,956	2,956	2,956	-
miles1500-30	128	5,198	4,522,260	30	2	40	78	77	2,956	2,956	2,956	-
miles1500-40	128	5,198	8,108,880	40	2	40	78	77	2,956	2,956	2,956	-
miles1500-50	128	5,198	12,735,100	50	2	40	78	77	2,956	2,956	2,956	-
miles1500-55	128	5,198	15,438,060	55	2	40	78	77	2,956	2,956	2,956	-
miles1500-60	128	5,198	18,400,920	60	2	40	78	77	2,956	2,956	2,956	-
miles1500-65	128	5,198	21,623,680	65	2	40	78	77	2,956	2,956	2,956	-
miles1500-70	128	5,198	25,106,340	70	2	40	78	77	2,956	2,956	2,956	-
miles1500-71	128	5,198	25,834,060	71	2	40	78	77	2,956	2,956	2,956	-
miles1500-72	128	5,198	26,572,176	72	2	40	78	77	2,956	2,956	2,956	-
miles250-6	128	387	9,810	6	2	62	10	9	37	37	38	37
miles250-7	128	387	13,734	7	2	62	10	9	37	37	38	37
miles500-10	128	1,170	105,300	10	2	68	24	22	244	244	245	-
miles500-15	128	1,170	245,700	15	2	68	24	22	244	244	245	-
miles500-18	128	1,170	358,020	18	2	68	24	22	244	244	245	-
miles500-19	128	1,170	400,140	19	2	68	24	22	244	244	245	-
miles500-5	128	1,170	23,400	5	2	68	24	22	244	244	245	244
miles750-10	128	2,113	190,170	10	2	58	41	35	576	576	577	-
miles750-15	128	2,113	443,730	15	2	58	41	35	576	576	577	-
miles750-20	128	2,113	802,940	20	2	58	41	35	576	576	577	-
miles750-25	128	2,113	1,267,800	25	2	58	41	35	576	576	577	-
miles750-28	128	2,113	1,597,428	28	2	58	41	35	576	576	577	-
miles750-29	128	2,113	1,715,756	29	2	58	41	35	576	576	577	-
miles750-30	128	2,113	1,838,310	30	2	58	41	35	576	576	577	-
miles750-5	128	2,113	42,260	5	2	58	41	35	576	576	577	576

Table E.8: Data characteristics of unsatisfiable binary instances (part 8).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
graphColoring-sgb-queenExtConvert												
queen10-10-10	100	1,470	132,300	10	2	18	80	76	924	924	2,709	-
queen10-10-11	100	1,470	161,700	11	2	18	80	76	924	924	2,709	-
queen10-10-8	100	1,470	82,320	8	2	18	80	76	924	924	2,709	-
queen10-10-9	100	1,470	105,840	9	2	18	80	76	924	924	2,709	-
queen11-11-10	121	1,980	178,200	10	2	23	96	92	1,235	1,235	4,201	-
queen11-11-11	121	1,980	217,800	11	2	23	96	92	1,235	1,235	4,201	-
queen11-11-12	121	1,980	261,360	12	2	23	96	92	1,235	1,235	4,201	-
queen11-11-8	121	1,980	110,880	8	2	23	96	92	1,235	1,235	4,201	-
queen11-11-9	121	1,980	142,560	9	2	23	96	92	1,235	1,235	4,201	-
queen12-12-10	144	2,596	233,640	10	2	26	118	115	1,739	1,739	6,311	-
queen12-12-11	144	2,596	285,560	11	2	26	118	115	1,739	1,739	6,311	-
queen12-12-12	144	2,596	342,672	12	2	26	118	115	1,739	1,739	6,311	-
queen12-12-13	144	2,596	404,976	13	2	26	118	115	1,739	1,739	6,311	-
queen12-12-14	144	2,596	472,472	14	2	26	118	115	1,739	1,739	6,311	-
queen13-13-10	169	3,328	299,520	10	2	29	138	131	2,198	2,198	8,666	-
queen13-13-11	169	3,328	366,080	11	2	29	138	131	2,198	2,198	8,666	-
queen13-13-12	169	3,328	439,296	12	2	29	138	131	2,198	2,198	8,666	-
queen13-13-13	169	3,328	519,168	13	2	29	138	131	2,198	2,198	-	-
queen13-13-14	169	3,328	605,696	14	2	29	138	131	2,198	2,198	-	-
queen14-14-12	196	4,186	552,552	12	2	27	161	151	2,822	2,822	-	-
queen14-14-13	196	4,186	653,016	13	2	27	161	151	2,822	2,822	-	-
queen14-14-14	196	4,186	761,852	14	2	27	161	151	2,822	2,822	-	-
queen14-14-15	196	4,186	879,060	15	2	27	161	151	2,822	2,822	-	-
queen14-14-16	196	4,186	1,004,640	16	2	27	161	151	2,822	2,822	-	-
queen15-15-13	225	5,180	808,080	13	2	31	184	173	3,459	3,459	-	-
queen15-15-14	225	5,180	942,760	14	2	31	184	173	3,459	3,459	-	-
queen15-15-15	225	5,180	1,087,800	15	2	31	184	173	3,459	3,459	-	-
queen15-15-16	225	5,180	1,243,200	16	2	31	184	173	3,459	3,459	-	-
queen15-15-17	225	5,180	1,408,960	17	2	31	184	173	3,459	3,459	-	-
queen16-16-14	256	6,320	1,150,240	14	2	32	219	196	4,617	4,617	-	-
queen16-16-15	256	6,320	1,327,200	15	2	32	219	196	4,617	4,617	-	-
queen16-16-16	256	6,320	1,516,800	16	2	32	219	196	4,617	4,617	-	-
queen16-16-17	256	6,320	1,719,040	17	2	32	219	196	4,617	4,617	-	-
queen16-16-18	256	6,320	1,933,920	18	2	32	219	196	4,617	4,617	-	-

Table E.9: Data characteristics of unsatisfiable binary instances (part 9).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
graphColoring-sgb-queenExtConvert												
queen5-5-4	25	160	1,920	4	2	7	19	18	87	87	141	87
queen6-6-6	36	290	8,700	6	2	10	27	25	156	156	299	-
queen7-7-6	49	476	14,280	6	2	12	38	36	276	276	599	276
queen8-12-10	96	1,368	123,120	10	2	20	73	67	787	787	2,079	-
queen8-12-11	96	1,368	150,480	11	2	20	73	67	787	787	2,079	-
queen8-12-12	96	1,368	180,576	12	2	20	73	67	787	787	2,079	-
queen8-12-8	96	1,368	76,608	8	2	20	73	67	787	787	2,079	-
queen8-8-8	64	728	40,768	8	2	15	49	47	414	414	970	-
queen8-8-9	64	728	52,416	9	2	15	49	47	414	414	970	-
queen9-9-8	81	1,056	59,136	8	2	16	66	58	702	702	1,883	-
queen9-9-9	81	1,056	76,032	9	2	16	66	58	702	702	1,883	-
QCP-15												
qcp-15-120-10	225	2,519	200,102	15	2	115	110	109	708	708	3,795	-
qcp-15-120-11	225	2,519	200,102	15	2	114	112	110	731	731	3,873	-
qcp-15-120-12	225	2,519	200,102	15	2	114	111	109	722	722	3,788	-
qcp-15-120-13	225	2,520	199,920	15	2	114	111	110	723	723	3,628	-
qcp-15-120-14	225	2,519	200,102	15	2	113	111	109	724	724	3,594	-
qcp-15-120-2	225	2,520	199,920	15	2	114	111	109	722	722	3,628	-
qcp-15-120-5	225	2,519	200,102	15	2	114	111	108	722	722	3,769	-
qcp-15-120-6	225	2,519	200,102	15	2	114	112	110	733	733	3,850	-
qcp-15-120-9	225	2,519	200,102	15	2	114	111	109	723	723	3,705	-
rlfapGraphsModExtConvert												
graph12-w1	680	1,148	703,017	44	2	593	37	29	10	28	28	28
graph13-w1	916	1,479	931,076	44	2	796	46	38	12	41	41	41
graph14-f27	916	4,638	1,081,870	19	2	615	243	214	486	498	977	-
graph14-f28	916	4,638	952,263	18	2	615	243	214	486	498	977	498
graph2-f25	400	2,245	550,494	21	2	250	88	74	154	163	173	163
graph8-f10	680	3,757	2,602,084	34	2	428	184	151	366	368	491	-
graph8-f11	680	3,757	2,463,291	33	2	428	184	151	366	368	491	368
graph9-f10	916	5,246	3,875,472	34	2	582	230	181	527	541	656	541
graph9-f9	916	5,246	4,157,294	35	2	582	230	181	527	541	656	-

Table E.10: Data characteristics of unsatisfiable binary instances (part 10).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
rlfapScens11ExtConvert												
scen11-f10	680	4,103	3,098,170	34	2	300	33	28	240	240	311	240
scen11-f11	680	4,103	2,974,001	33	2	300	33	28	240	240	311	240
scen11-f12	680	4,103	2,860,048	32	2	300	33	28	240	240	311	240
scen11-f1	680	4,103	5,279,405	43	2	300	33	28	240	240	311	-
scen11-f2	680	4,103	5,011,502	42	2	300	33	28	240	240	311	-
scen11-f3	680	4,103	4,750,411	41	2	300	33	28	240	240	311	-
scen11-f4	680	4,103	4,496,772	40	2	300	33	28	240	240	311	-
scen11-f5	680	4,103	4,249,959	39	2	300	33	28	240	240	311	-
scen11-f6	680	4,103	4,010,424	38	2	300	33	28	240	240	311	-
scen11-f7	680	4,103	3,776,246	37	2	300	33	28	240	240	311	-
scen11-f8	680	4,103	3,546,574	36	2	300	33	28	240	240	311	240
scen11-f9	680	4,103	3,320,443	35	2	300	33	28	240	240	311	240
rlfapScensModExtConvert												
scen1-f9	916	5,548	3,309,990	35	2	411	33	28	237	237	304	-
scen10-w1-f3	680	1,138	516,766	41	2	423	8	7	15	15	19	11
scen2-f25	200	1,235	358,088	21	2	96	21	17	188	188	199	-
scen3-f10	400	2,760	2,161,813	34	2	192	34	29	209	209	271	-
scen3-f11	400	2,760	2,069,789	33	2	192	34	29	209	209	271	-
scen6-w1-f2	200	319	12,032	42	2	20	5	4	10	10	10	6
scen6-w2	200	648	553,172	44	2	139	14	12	58	58	58	58
scen7-w1-f5	400	660	129,498	39	2	225	8	7	15	15	15	8
scen9-w1-f3	680	1,138	516,766	41	2	423	8	7	15	15	19	11

Table E.11: Data characteristics of unsatisfiable binary instances (part 11).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
tightness0.9												
rand-2-40-180-84-900-11	40	84	271,856	180	2	30	10	9	9	10	10	10
rand-2-40-180-84-900-14	40	84	273,008	180	2	31	10	9	12	12	14	11
rand-2-40-180-84-900-15	40	84	272,796	180	2	31	10	9	10	11	12	11
rand-2-40-180-84-900-16	40	84	272,242	180	2	30	10	9	8	9	9	9
rand-2-40-180-84-900-18	40	84	271,525	180	2	31	10	8	10	11	12	8
rand-2-40-180-84-900-22	40	84	268,772	180	2	29	10	9	10	11	12	9
rand-2-40-180-84-900-23	40	84	272,861	180	2	31	10	9	9	10	10	9
rand-2-40-180-84-900-25	40	84	271,936	180	2	31	10	8	9	10	10	9
rand-2-40-180-84-900-27	40	84	272,483	180	2	30	11	10	10	11	11	11
rand-2-40-180-84-900-28	40	84	271,875	180	2	33	8	7	11	11	12	8
rand-2-40-180-84-900-29	40	84	272,610	180	2	30	11	9	12	12	14	10
rand-2-40-180-84-900-2	40	84	272,120	180	2	29	11	9	9	10	10	10
rand-2-40-180-84-900-34	40	84	272,167	180	2	31	10	9	10	11	11	11
rand-2-40-180-84-900-35	40	84	272,022	180	2	30	11	10	11	11	11	9
rand-2-40-180-84-900-39	40	84	271,965	180	2	31	10	9	9	10	10	10
rand-2-40-180-84-900-3	40	84	271,885	180	2	32	9	8	10	11	12	8
rand-2-40-180-84-900-40	40	84	272,106	180	2	30	11	8	13	13	14	11
rand-2-40-180-84-900-41	40	84	272,696	180	2	30	11	10	11	11	11	10
rand-2-40-180-84-900-43	40	84	272,767	180	2	31	10	9	11	12	12	8
rand-2-40-180-84-900-45	40	84	272,491	180	2	32	9	8	9	11	11	9
rand-2-40-180-84-900-46	40	84	271,778	180	2	30	11	9	13	13	13	10
rand-2-40-180-84-900-4	40	84	272,319	180	2	31	10	9	14	15	16	10
rand-2-40-180-84-900-54	40	84	269,890	180	2	29	10	9	12	12	13	10
rand-2-40-180-84-900-57	40	84	272,206	180	2	31	10	9	9	10	10	8
rand-2-40-180-84-900-58	40	84	268,742	180	2	28	10	8	12	13	15	10
rand-2-40-180-84-900-60	40	84	272,722	180	2	29	12	10	7	11	11	11
rand-2-40-180-84-900-62	40	84	272,292	180	2	31	10	9	12	13	13	10
rand-2-40-180-84-900-63	40	84	272,398	180	2	30	11	9	11	11	12	9
rand-2-40-180-84-900-65	40	84	272,190	180	2	29	12	9	7	11	11	11
rand-2-40-180-84-900-67	40	84	272,113	180	2	31	10	9	13	14	14	9
rand-2-40-180-84-900-6	40	84	271,675	180	2	30	11	10	10	11	12	9

Table E.12: Data characteristics of unsatisfiable binary instances (part 12).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
tightness0.9												
rand-2-40-180-84-900-70	40	84	273,356	180	2	30	11	10	12	12	13	11
rand-2-40-180-84-900-75	40	84	272,569	180	2	30	11	9	12	12	12	9
rand-2-40-180-84-900-76	40	84	272,638	180	2	29	12	11	12	12	12	12
rand-2-40-180-84-900-77	40	84	271,261	180	2	32	9	7	9	10	10	9
rand-2-40-180-84-900-78	40	84	271,701	180	2	29	12	11	7	10	10	10
rand-2-40-180-84-900-79	40	84	272,163	180	2	30	11	10	9	11	11	11
rand-2-40-180-84-900-7	40	84	271,884	180	2	30	11	8	10	11	11	10
rand-2-40-180-84-900-80	40	84	271,973	180	2	32	9	8	9	9	9	9
rand-2-40-180-84-900-82	40	84	268,849	180	2	28	11	10	12	12	12	12
rand-2-40-180-84-900-84	40	84	271,857	180	2	30	11	9	8	9	9	9
rand-2-40-180-84-900-85	40	84	272,786	180	2	30	11	9	12	12	14	10
rand-2-40-180-84-900-86	40	84	271,878	180	2	29	12	10	8	12	12	12
rand-2-40-180-84-900-87	40	84	272,585	180	2	31	10	9	9	12	13	9
rand-2-40-180-84-900-89	40	84	272,769	180	2	29	12	10	12	13	13	11
rand-2-40-180-84-900-90	40	84	270,846	180	2	29	12	10	13	13	13	11
rand-2-40-180-84-900-92	40	84	272,420	180	2	30	11	9	8	9	9	9
rand-2-40-180-84-900-93	40	84	271,848	180	2	31	10	9	13	13	13	8
rand-2-40-180-84-900-94	40	84	272,637	180	2	29	11	10	10	11	11	10
rand-2-40-180-84-900-95	40	84	270,912	180	2	31	10	9	13	13	17	11
rand-2-40-180-84-900-97	40	84	271,985	180	2	31	10	9	9	11	11	9
rand-2-40-180-84-900-98	40	84	269,492	180	2	30	9	8	7	8	8	8
rand-2-40-180-84-900-99	40	84	272,598	180	2	30	11	9	11	12	14	11
rand-2-40-180-84-900-9	40	84	272,155	180	2	30	11	10	9	13	13	11

Table E.13: Data characteristics of unsatisfiable non-binary instances (part 1).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
aim-100												
aim-100-1-6-unsat-1	100	157	1,096	2	3	58	41	34	8	62	130	59
aim-100-1-6-unsat-2	100	150	1,032	2	3	58	40	35	9	65	145	58
aim-100-1-6-unsat-3	100	151	1,050	2	3	57	41	37	6	66	135	56
aim-100-1-6-unsat-4	100	157	1,088	2	3	60	39	37	5	62	169	52
aim-100-2-0-unsat-1	100	196	1,368	2	3	48	51	47	13	106	410	79
aim-100-2-0-unsat-2	100	192	1,337	2	3	44	54	47	20	103	417	84
aim-100-2-0-unsat-3	100	193	1,338	2	3	49	51	50	12	98	330	73
aim-100-2-0-unsat-4	100	191	1,328	2	3	48	51	46	17	95	351	82
aim-200												
aim-200-1-6-unsat-1	200	308	2,140	2	3	108	88	82	16	151	602	132
aim-200-1-6-unsat-2	200	302	2,095	2	3	113	83	79	15	145	577	-
aim-200-1-6-unsat-3	200	309	2,144	2	3	112	79	71	12	118	411	113
aim-200-1-6-unsat-4	200	316	2,208	2	3	105	92	82	16	158	613	145
aim-200-2-0-unsat-1	200	389	2,709	2	3	91	98	95	21	188	1,070	-
aim-200-2-0-unsat-2	200	383	2,661	2	3	89	105	100	19	219	1,420	-
aim-200-2-0-unsat-3	200	388	2,697	2	3	88	107	100	24	228	1,401	-
aim-200-2-0-unsat-4	200	392	2,724	2	3	88	106	96	27	216	1,299	-
aim-50												
aim-50-1-6-unsat-1	50	69	472	2	3	33	16	14	5	18	24	17
aim-50-1-6-unsat-2	50	77	536	2	3	29	21	18	6	30	53	22
aim-50-1-6-unsat-3	50	70	476	2	3	31	19	17	5	29	41	21
aim-50-1-6-unsat-4	50	76	528	2	3	29	21	18	6	36	63	29
aim-50-2-0-unsat-1	50	97	676	2	3	22	29	26	9	60	168	43
aim-50-2-0-unsat-2	50	94	652	2	3	24	27	26	7	56	122	38
aim-50-2-0-unsat-3	50	92	636	2	3	23	27	25	7	52	111	37
aim-50-2-0-unsat-4	50	94	650	2	3	27	24	22	9	41	97	31

Table E.14: Data characteristics of unsatisfiable non-binary instances (part 2).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
dag-rand												
rand-n23-d3-e16-r15-t150000-1	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-10	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-11	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-12	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-13	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-14	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-15	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-16	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-17	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-18	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-19	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-2	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-20	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-21	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-22	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-23	23	16	2,400,000	3	15	2	22	21	9	16	16	16
rand-n23-d3-e16-r15-t150000-24	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-25	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-3	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-4	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-5	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-6	23	16	2,400,000	3	15	2	22	21	12	16	16	16
rand-n23-d3-e16-r15-t150000-7	23	16	2,400,000	3	15	2	22	21	12	16	16	16
rand-n23-d3-e16-r15-t150000-8	23	16	2,400,000	3	15	1	23	0	16	16	16	16
rand-n23-d3-e16-r15-t150000-9	23	16	2,400,000	3	15	1	23	0	16	16	16	16
dubois												
dubois-100	300	200	800	2	3	198	4	3	2	2	2	2
dubois-20	60	40	160	2	3	38	4	3	2	2	2	2
dubois-21	63	42	168	2	3	40	4	3	2	2	2	2
dubois-22	66	44	176	2	3	42	4	3	2	2	2	2
dubois-23	69	46	184	2	3	44	4	3	2	2	2	2
dubois-24	72	48	192	2	3	46	4	3	2	2	2	2
dubois-25	75	50	200	2	3	48	4	3	2	2	2	2
dubois-26	78	52	208	2	3	50	4	3	2	2	2	2
dubois-27	81	54	216	2	3	52	4	3	2	2	2	2
dubois-28	84	56	224	2	3	54	4	3	2	2	2	2
dubois-29	87	58	232	2	3	56	4	3	2	2	2	2
dubois-30	90	60	240	2	3	58	4	3	2	2	2	2
dubois-50	150	100	400	2	3	98	4	3	2	2	2	2

Table E.15: Data characteristics of unsatisfiable non-binary instances (part 3).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
renault-mod-10												
renault-mod-10	111	128	204,467	42	10	87	11	10	6	8	8	8
renault-mod-12	111	128	198,878	42	10	87	11	10	5	8	8	8
renault-mod-14	111	126	200,498	42	10	88	12	11	5	8	8	8
renault-mod-15	111	128	202,514	42	10	90	11	10	6	10	10	10
renault-mod-16	111	127	200,340	42	10	88	11	10	4	8	8	6
renault-mod-17	111	127	199,013	42	10	88	11	10	5	8	9	6
renault-mod-18	111	127	198,390	42	10	86	12	11	4	8	8	7
renault-mod-19	111	128	199,586	42	10	90	10	9	5	7	7	6
renault-mod-1	111	126	200,801	42	10	90	11	9	4	7	7	7
renault-mod-20	111	135	204,879	42	10	86	12	11	9	13	13	15
renault-mod-21	111	134	206,076	42	10	88	12	11	10	13	13	13
renault-mod-22	111	134	202,392	42	10	87	10	9	11	12	12	12
renault-mod-23	111	136	205,517	42	10	90	10	9	9	14	14	14
renault-mod-24	111	135	205,262	42	10	88	12	11	9	12	12	15
renault-mod-25	111	135	201,901	42	10	87	11	10	9	14	14	15
renault-mod-26	111	136	209,114	42	10	90	10	9	11	16	16	16
renault-mod-27	111	136	203,992	42	10	90	12	10	9	13	13	13
renault-mod-28	111	135	206,411	42	10	84	10	9	13	15	15	17
renault-mod-29	111	137	208,281	42	10	87	12	10	14	18	18	18
renault-mod-30	111	131	201,400	42	10	89	11	10	4	7	8	6
renault-mod-33	111	133	202,220	42	10	89	12	11	5	8	8	8
renault-mod-35	111	133	203,144	42	10	88	11	10	6	7	8	7
renault-mod-37	111	133	199,797	42	10	90	13	12	7	12	12	12
renault-mod-39	111	132	197,984	42	10	90	13	11	5	11	12	7
renault-mod-3	111	125	199,261	42	10	91	11	10	4	7	8	8
renault-mod-40	108	128	198,070	42	10	91	12	11	6	9	9	9
renault-mod-42	108	126	197,154	42	10	91	13	11	5	9	13	8
renault-mod-47	108	128	197,403	42	10	86	13	12	5	10	10	10
renault-mod-5	111	125	199,771	42	10	88	13	11	5	8	9	10
renault-mod-6	111	125	198,651	42	10	88	11	10	4	6	7	6
renault-mod-8	111	125	197,300	42	10	90	12	11	4	7	8	7

Table E.16: Data characteristics of unsatisfiable non-binary instances (part 4).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
rand-10-20-10												
rand-10-20-10-5-10000-0	20	5	50,000	10	10	7	13	12	1	5	5	5
rand-10-20-10-5-10000-10	20	5	50,000	10	10	4	15	12	2	5	5	5
rand-10-20-10-5-10000-11	20	5	50,000	10	10	5	13	11	2	5	5	5
rand-10-20-10-5-10000-12	20	5	50,000	10	10	7	13	11	1	5	5	5
rand-10-20-10-5-10000-13	20	5	50,000	10	10	5	15	13	2	5	5	5
rand-10-20-10-5-10000-14	20	5	50,000	10	10	6	13	11	1	5	5	5
rand-10-20-10-5-10000-15	20	5	50,000	10	10	6	14	12	1	5	5	5
rand-10-20-10-5-10000-16	20	5	50,000	10	10	5	15	13	2	5	5	5
rand-10-20-10-5-10000-17	20	5	50,000	10	10	4	14	12	2	5	5	5
rand-10-20-10-5-10000-18	20	5	50,000	10	10	8	12	11	1	5	5	5
rand-10-20-10-5-10000-19	20	5	50,000	10	10	5	14	11	1	5	5	5
rand-10-20-10-5-10000-1	20	5	50,000	10	10	6	13	12	1	5	5	5
rand-10-20-10-5-10000-2	20	5	50,000	10	10	6	12	11	1	5	5	5
rand-10-20-10-5-10000-3	20	5	50,000	10	10	5	14	12	2	5	5	5
rand-10-20-10-5-10000-4	20	5	50,000	10	10	6	13	11	1	5	5	5
rand-10-20-10-5-10000-5	20	5	50,000	10	10	6	14	13	1	5	5	5
rand-10-20-10-5-10000-6	20	5	50,000	10	10	4	14	12	2	5	5	5
rand-10-20-10-5-10000-7	20	5	50,000	10	10	5	13	10	1	5	5	6
rand-10-20-10-5-10000-8	20	5	50,000	10	10	5	15	13	2	5	5	5
rand-10-20-10-5-10000-9	20	5	50,000	10	10	6	14	12	1	5	5	5
ssa												
ssa-0432-003	435	501	2,147	2	5	372	19	15	9	16	16	16
ssa-2670-130	1,359	1,660	7,558	2	5	1,157	27	20	18	26	26	26

Table E.17: Data characteristics of satisfiable binary instances (part 1).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
composed-25-10-20												
composed-25-10-20-0	105	620	47,000	10	2	82	24	23	183	183	260	-
composed-25-10-20-1	105	620	47,000	10	2	82	24	23	186	186	257	-
composed-25-10-20-2	105	620	47,000	10	2	82	24	23	183	183	251	-
composed-25-10-20-3	105	620	47,000	10	2	81	25	16	200	200	266	-
composed-25-10-20-4	105	620	47,000	10	2	82	24	23	184	184	248	-
composed-25-10-20-5	105	620	47,000	10	2	81	25	19	200	200	266	-
composed-25-10-20-6	105	620	47,000	10	2	81	24	23	186	186	255	-
composed-25-10-20-7	105	620	47,000	10	2	82	24	21	184	184	255	-
composed-25-10-20-8	105	620	47,000	10	2	80	24	22	189	189	270	-
composed-25-10-20-9	105	620	47,000	10	2	81	24	22	183	183	249	-
graphColoring-hosExtConvert												
abb313GPIA-10	1,557	53,356	4,802,040	10	2	259	121	116	3,759	3,759	4,869	-
ash331GPIA-4	662	4,181	50,172	4	2	318	89	68	143	144	240	94
ash608GPIA-4	1,216	7,844	94,128	4	2	585	122	92	211	211	349	134
ash958GPIA-4	1,916	12,506	150,072	4	2	912	127	97	239	245	420	-
will199GPIA-7	701	6,772	284,424	7	2	287	109	83	596	596	1,191	-
graphColoring-mugExtConvert												
mug100-1-4	100	166	1,992	4	2	65	4	2	5	5	5	5
mug100-25-4	100	166	1,992	4	2	65	4	2	5	5	5	5
mug88-1-4	88	146	1,752	4	2	57	4	2	5	5	5	5
mug88-25-4	88	146	1,752	4	2	57	4	2	5	5	5	5
graphColoring-register-mulsolExtConvert												
mulsol-i-1-49	197	3,925	9,231,600	49	2	126	51	50	1,243	1,243	1,243	-
mulsol-i-2-31	188	3,885	3,613,050	31	2	111	33	31	496	496	496	-
mulsol-i-3-31	184	3,916	3,641,880	31	2	107	33	31	496	496	496	-
mulsol-i-4-31	185	3,946	3,669,780	31	2	107	33	31	496	496	496	-
mulsol-i-5-31	186	3,973	3,694,890	31	2	109	33	31	480	480	480	-
graphColoring-register-zeroinExtConvert												
zeroin-i-1-49	211	4,100	9,643,200	49	2	145	51	49	1,243	1,243	1,243	-
zeroin-i-2-30	211	3,541	3,080,670	30	2	147	34	30	464	464	464	-
zeroin-i-3-30	206	3,540	3,079,800	30	2	142	34	30	464	464	464	-
graphColoring-sgb-bookExtConvert												
anna-11	138	493	54,230	11	2	111	13	12	62	62	65	-
david-11	87	406	44,660	11	2	59	14	12	75	75	77	-
homer-13	561	1,628	252,564	13	2	452	32	30	207	207	272	-
huck-11	74	301	33,000	11	2	33	11	6	55	55	55	53
jean-10	80	254	22,860	10	2	54	10	8	45	45	45	28
graphColoring-sgb-gamesExtConvert												
games120-9	120	638	45,936	9	2	65	41	35	83	83	116	36

Table E.18: Data characteristics of satisfiable binary instances (part 2).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
graphColoring-sgb-milesExtConvert												
miles1000-42	128	3,216	5,537,952	42	2	59	51	48	1,109	1,109	1,163	-
miles1500-73	128	5,198	27,320,688	73	2	40	78	77	2,956	2,956	2,956	-
miles250-8	128	387	21,560	8	2	80	10	9	37	37	38	10
miles500-20	128	1,170	444,600	20	2	68	24	22	244	244	245	-
miles750-31	128	2,113	1,965,090	31	2	58	41	35	576	576	577	-
graphColoring-sgb-queenExtConvert												
queen10-10-12	100	1,470	194,040	12	2	18	80	76	924	924	2,709	-
queen5-5-5	25	160	3,200	5	2	7	19	18	87	87	141	20
queen6-6-7	36	290	12,180	7	2	10	27	25	156	156	299	-
queen7-7-7	49	476	19,992	7	2	12	38	36	276	276	599	-
queen9-9-10	81	1,056	95,040	10	2	16	66	58	702	702	1,883	-
hanoi												
hanoi-3	6	5	246	27	2	5	2	1	1	1	1	1
hanoi-4	14	13	2,652	81	2	13	2	1	1	1	1	1
hanoi-5	30	29	19,614	243	2	29	2	1	1	1	1	1
hanoi-6	62	61	128,868	729	2	61	2	1	1	1	1	1
hanoi-7	126	125	806,646	2,187	2	125	2	1	1	1	1	1
QCP-15												
qcp-15-120-0	225	2,519	200,102	15	2	113	110	108	711	711	3,873	-
qcp-15-120-1	225	2,519	200,102	15	2	114	111	109	721	721	3,720	-
qcp-15-120-3	225	2,520	199,920	15	2	113	111	109	724	724	3,746	-
qcp-15-120-4	225	2,519	200,102	15	2	114	111	109	723	723	3,631	-
qcp-15-120-7	225	2,520	199,920	15	2	113	113	111	745	745	4,086	-
qcp-15-120-8	225	2,519	200,102	15	2	115	110	109	712	712	3,522	-
rlfapGraphsModExtConvert												
graph12-w0	680	340	24	44	2	340	2	0	-	-	-	-
graph13-w0	916	458	24	44	2	458	2	0	-	-	-	-
graph2-f24	400	2,245	597,335	22	2	250	88	74	154	163	173	59
rlfapScensModExtConvert												
scen1-f8	916	5,548	3,545,401	36	2	427	33	28	237	237	304	-
scen2-f24	200	1,235	382,310	22	2	96	21	17	188	188	199	-
scen6-w1	200	319	201,537	44	2	120	8	7	15	15	15	8
scen7-w1-f4	400	660	338,740	40	2	260	8	7	15	15	15	8
tightness0.9												
rand-2-40-180-84-900-0	40	84	272,808	180	2	30	11	9	11	12	13	10
rand-2-40-180-84-900-10	40	84	272,033	180	2	31	10	8	9	11	13	9
rand-2-40-180-84-900-12	40	84	271,724	180	2	30	11	9	7	8	8	8
rand-2-40-180-84-900-13	40	84	272,352	180	2	32	9	8	9	9	9	8
rand-2-40-180-84-900-17	40	84	272,099	180	2	31	10	9	8	10	10	10
rand-2-40-180-84-900-19	40	84	273,025	180	2	29	12	10	8	11	11	11

Table E.19: Data characteristics of satisfiable binary instances (part 3).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
tightness0.9												
rand-2-40-180-84-900-1	40	84	271,729	180	2	31	10	9	10	11	11	11
rand-2-40-180-84-900-20	40	84	272,009	180	2	30	11	9	11	13	15	10
rand-2-40-180-84-900-21	40	84	272,281	180	2	30	11	10	8	10	10	10
rand-2-40-180-84-900-24	40	84	272,204	180	2	30	11	9	6	9	9	9
rand-2-40-180-84-900-26	40	84	272,146	180	2	30	11	9	9	10	12	10
rand-2-40-180-84-900-30	40	84	272,423	180	2	29	12	11	8	12	13	10
rand-2-40-180-84-900-31	40	84	271,657	180	2	31	10	9	7	8	8	8
rand-2-40-180-84-900-32	40	84	272,237	180	2	30	11	10	12	12	17	11
rand-2-40-180-84-900-33	40	84	272,741	180	2	29	12	11	14	16	19	13
rand-2-40-180-84-900-36	40	84	272,554	180	2	30	11	10	9	10	10	10
rand-2-40-180-84-900-37	40	84	273,336	180	2	31	10	9	8	11	11	11
rand-2-40-180-84-900-38	40	84	271,877	180	2	30	11	9	8	11	11	11
rand-2-40-180-84-900-42	40	84	272,451	180	2	30	11	10	13	13	13	10
rand-2-40-180-84-900-44	40	84	272,100	180	2	30	10	8	9	10	10	9
rand-2-40-180-84-900-47	40	84	272,104	180	2	31	10	9	6	9	9	9
rand-2-40-180-84-900-48	40	84	272,260	180	2	29	12	10	9	12	12	10
rand-2-40-180-84-900-49	40	84	272,409	180	2	31	10	8	6	8	8	8
rand-2-40-180-84-900-50	40	84	272,629	180	2	30	11	10	9	10	10	10
rand-2-40-180-84-900-51	40	84	271,997	180	2	29	12	11	9	11	11	11
rand-2-40-180-84-900-52	40	84	272,765	180	2	30	11	9	8	10	10	10
rand-2-40-180-84-900-53	40	84	271,384	180	2	31	10	9	9	10	10	8
rand-2-40-180-84-900-55	40	84	272,957	180	2	30	11	8	10	10	10	10
rand-2-40-180-84-900-56	40	84	271,686	180	2	30	11	9	10	11	11	9
rand-2-40-180-84-900-59	40	84	272,860	180	2	30	11	10	10	11	11	11
rand-2-40-180-84-900-5	40	84	272,571	180	2	30	11	10	8	11	11	11
rand-2-40-180-84-900-61	40	84	272,181	180	2	30	11	10	8	10	10	10
rand-2-40-180-84-900-64	40	84	268,112	180	2	28	11	10	7	10	10	10
rand-2-40-180-84-900-66	40	84	272,164	180	2	30	11	10	10	11	14	9
rand-2-40-180-84-900-68	40	84	272,594	180	2	30	11	9	7	10	10	10
rand-2-40-180-84-900-69	40	84	272,035	180	2	31	10	9	7	8	8	8
rand-2-40-180-84-900-71	40	84	271,515	180	2	30	11	10	10	11	14	9
rand-2-40-180-84-900-72	40	84	272,743	180	2	29	12	11	10	11	13	9
rand-2-40-180-84-900-73	40	84	271,089	180	2	30	11	10	11	11	11	11
rand-2-40-180-84-900-74	40	84	272,519	180	2	31	10	9	10	11	12	9
rand-2-40-180-84-900-81	40	84	272,145	180	2	30	11	9	9	9	9	9
rand-2-40-180-84-900-83	40	84	272,326	180	2	30	11	10	13	15	17	11
rand-2-40-180-84-900-88	40	84	272,230	180	2	31	10	9	11	12	12	9
rand-2-40-180-84-900-8	40	84	272,470	180	2	30	11	10	9	-	-	-
rand-2-40-180-84-900-91	40	84	272,774	180	2	30	11	10	9	-	-	-
rand-2-40-180-84-900-96	40	84	272,863	180	2	28	13	12	12	-	-	-

Table E.20: Data characteristics of satisfiable non-binary instances (part 1).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
aim-100												
aim-100-1-6-sat-1	100	154	1,068	2	3	60	37	33	5	44	63	36
aim-100-1-6-sat-2	100	156	1,084	2	3	62	34	31	6	42	73	39
aim-100-1-6-sat-3	100	156	1,088	2	3	60	38	32	6	56	113	59
aim-100-1-6-sat-4	100	157	1,096	2	3	61	37	34	5	47	75	40
aim-100-2-0-sat-1	100	194	1,350	2	3	52	46	43	10	69	202	63
aim-100-2-0-sat-2	100	197	1,368	2	3	51	43	37	10	66	172	70
aim-100-2-0-sat-3	100	191	1,324	2	3	54	45	42	10	70	186	62
aim-100-2-0-sat-4	100	195	1,361	2	3	53	45	43	10	72	185	64
aim-100-3-4-sat-1	100	320	2,216	2	3	37	63	54	55	190	943	-
aim-100-3-4-sat-2	100	316	2,176	2	3	38	62	61	56	184	964	-
aim-100-3-4-sat-3	100	312	2,157	2	3	36	64	62	54	197	1,152	-
aim-100-3-4-sat-4	100	317	2,193	2	3	35	65	59	56	210	1,077	-
aim-100-6-0-sat-1	100	559	3,861	2	3	27	73	71	188	381	1,828	-
aim-100-6-0-sat-2	100	559	3,868	2	3	27	74	72	193	369	1,892	-
aim-100-6-0-sat-3	100	561	3,880	2	3	24	77	75	216	408	2,222	-
aim-100-6-0-sat-4	100	570	3,946	2	3	28	73	72	190	381	2,032	-
aim-200												
aim-200-1-6-sat-1	200	315	2,196	2	3	121	72	64	6	101	237	100
aim-200-1-6-sat-2	200	315	2,199	2	3	128	68	65	8	88	183	91
aim-200-1-6-sat-3	200	311	2,165	2	3	120	72	59	8	100	208	95
aim-200-1-6-sat-4	200	318	2,208	2	3	123	75	61	7	94	177	96
aim-200-2-0-sat-1	200	386	2,687	2	3	101	92	84	13	156	615	140
aim-200-2-0-sat-2	200	382	2,653	2	3	99	90	80	15	151	546	137
aim-200-2-0-sat-3	200	387	2,693	2	3	100	89	77	24	141	513	140
aim-200-2-0-sat-4	200	389	2,705	2	3	102	95	84	15	169	653	148
aim-200-3-4-sat-1	200	646	4,481	2	3	71	130	119	115	424	3,807	-
aim-200-3-4-sat-2	200	636	4,409	2	3	74	125	123	103	391	3,800	-
aim-200-3-4-sat-3	200	641	4,441	2	3	73	124	118	106	377	3,843	-
aim-200-3-4-sat-4	200	643	4,459	2	3	71	128	120	123	407	3,803	-
aim-200-6-0-sat-1	200	1,152	8,009	2	3	45	156	154	487	962	8,810	-
aim-200-6-0-sat-2	200	1,169	8,135	2	3	42	158	157	501	1,001	9,359	-
aim-200-6-0-sat-3	200	1,150	8,003	2	3	45	156	154	479	984	8,957	-
aim-200-6-0-sat-4	200	1,155	8,033	2	3	47	154	153	454	971	8,804	-

Table E.21: Data characteristics of satisfiable non-binary instances (part 2).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
aim-50												
aim-50-1-6-sat-1	50	77	536	2	3	31	19	17	6	24	39	22
aim-50-1-6-sat-2	50	76	529	2	3	33	17	15	6	20	22	16
aim-50-1-6-sat-3	50	78	536	2	3	29	20	18	5	24	37	22
aim-50-1-6-sat-4	50	77	528	2	3	31	20	17	5	22	34	23
aim-50-2-0-sat-1	50	94	653	2	3	25	26	23	8	44	87	40
aim-50-2-0-sat-2	50	96	664	2	3	24	26	22	7	39	73	33
aim-50-2-0-sat-3	50	96	668	2	3	27	24	21	8	41	94	32
aim-50-2-0-sat-4	50	93	643	2	3	28	21	19	7	30	49	29
aim-50-3-4-sat-1	50	156	1,079	2	3	19	31	29	24	96	252	50
aim-50-3-4-sat-2	50	161	1,118	2	3	18	32	29	36	95	266	39
aim-50-3-4-sat-3	50	161	1,118	2	3	19	32	31	31	104	298	41
aim-50-3-4-sat-4	50	159	1,094	2	3	19	31	29	32	84	255	39
aim-50-6-0-sat-1	50	289	2,011	2	3	13	38	35	107	193	461	39
aim-50-6-0-sat-2	50	283	1,956	2	3	13	38	36	101	174	485	40
aim-50-6-0-sat-3	50	267	1,835	2	3	13	38	36	99	173	482	41
aim-50-6-0-sat-4	50	272	1,877	2	3	14	37	36	98	173	438	40
modifiedRenault												
renault-mod-0	111	125	198,433	42	10	89	11	9	4	6	6	6
renault-mod-11	111	126	200,750	42	10	88	11	10	4	8	8	6
renault-mod-13	111	128	203,753	42	10	87	10	9	4	7	7	6
renault-mod-2	111	129	201,513	42	10	87	14	13	6	11	11	7
renault-mod-31	111	133	206,800	42	10	89	11	10	4	8	9	6
renault-mod-32	111	132	210,607	42	10	92	13	12	6	8	8	6
renault-mod-34	111	132	203,713	42	10	91	12	11	5	9	9	8
renault-mod-36	111	131	200,087	42	10	90	12	11	5	7	8	7
renault-mod-38	111	133	202,728	42	10	90	12	11	5	9	9	6
renault-mod-41	108	128	200,610	42	10	87	14	12	6	9	13	7
renault-mod-43	108	128	200,466	42	10	87	12	11	5	9	9	5
renault-mod-44	108	127	196,306	42	10	90	12	11	7	9	9	7
renault-mod-45	108	128	198,947	42	10	93	11	10	4	8	8	6
renault-mod-46	108	128	197,604	42	10	86	12	11	5	10	11	6
renault-mod-48	108	128	202,599	42	10	87	13	12	6	9	10	8
renault-mod-49	108	127	198,602	42	10	88	11	10	4	9	11	6
renault-mod-4	111	126	200,503	42	10	92	11	10	4	8	8	5
renault-mod-7	111	125	198,972	42	10	89	11	10	5	8	8	6
renault-mod-9	111	125	202,777	42	10	93	11	10	6	8	8	6

Table E.22: Data characteristics of satisfiable non-binary instances (part 3).

file	#variables	#constraints	#total tuples	max domain	max arity	#clusters	treewidth	largest sep.	max $\|\psi(cl)\|$			
									local	proj	binary	clique
rand-8-20-5												
rand-8-20-5-18-800-0	20	18	1,407,561	5	8	3	18	17	8	18	20	-
rand-8-20-5-18-800-10	20	18	1,404,280	5	8	3	18	17	10	18	18	-
rand-8-20-5-18-800-11	20	18	1,406,099	5	8	3	18	17	11	18	18	-
rand-8-20-5-18-800-12	20	18	1,405,607	5	8	3	18	17	9	18	20	-
rand-8-20-5-18-800-13	20	18	1,407,854	5	8	3	18	17	9	18	21	-
rand-8-20-5-18-800-14	20	18	1,406,879	5	8	4	17	16	9	18	18	-
rand-8-20-5-18-800-15	20	18	1,405,279	5	8	3	18	17	11	18	19	-
rand-8-20-5-18-800-16	20	18	1,405,823	5	8	3	18	17	8	18	18	-
rand-8-20-5-18-800-17	20	18	1,406,089	5	8	3	18	16	9	18	18	-
rand-8-20-5-18-800-18	20	18	1,405,715	5	8	3	18	16	10	18	21	-
rand-8-20-5-18-800-19	20	18	1,406,180	5	8	4	17	16	9	18	19	-
rand-8-20-5-18-800-1	20	18	1,406,243	5	8	3	18	16	8	18	22	-
rand-8-20-5-18-800-2	20	18	1,406,899	5	8	4	17	16	6	18	19	-
rand-8-20-5-18-800-3	20	18	1,407,581	5	8	3	18	16	11	18	19	-
rand-8-20-5-18-800-4	20	18	1,406,167	5	8	3	18	17	9	18	22	-
rand-8-20-5-18-800-5	20	18	1,405,786	5	8	3	18	16	8	18	20	-
rand-8-20-5-18-800-6	20	18	1,406,135	5	8	4	17	16	9	18	19	-
rand-8-20-5-18-800-7	20	18	1,405,338	5	8	4	17	15	8	18	20	-
rand-8-20-5-18-800-8	20	18	1,407,092	5	8	4	17	16	8	18	20	-
rand-8-20-5-18-800-9	20	18	1,405,300	5	8	3	18	17	10	18	20	-
ssa												
ssa-2670-141	391	177	655	2	4	83	6	3	6	6	6	6
ssa-6288-047	10,408	22,141	124,515	2	6	7,877	122	86	30	82	82	82
ssa-7552-038	1,501	1,985	8,657	2	6	1,300	30	21	24	26	34	20
ssa-7552-158	1,363	1,641	6,402	2	5	1,206	11	10	12	12	14	12
ssa-7552-159	1,363	1,639	6,402	2	5	1,206	11	9	12	12	14	12
ssa-7552-160	757	744	3,942	2	3	368	5	3	6	6	6	6