Domain Value Mutation and other techniques for Constraint Satisfaction Problems

Chavalit Likitvivatanavong
MSc

NATIONAL UNIVERSITY OF IRELAND, CORK

Faculty of Science

Department of Computer Science

Thesis submitted for the degree of Doctor of Philosophy

April 2017

Head of Department: Prof Cormac J. Sreenan

Supervisor: Prof James Bowen

Contents

List of Figures . . . v
List of Tables . . . vii
List of Algorithms . . . vii
Declaration and Acknowledgements . . . xi

1 Introduction 1
1.1 A new perspective on domain values . . . 3
1.1.1 Domain transmutation . . . 5
1.1.2 Extracting microstructure . . . 6
1.1.3 Virtual interchangeability . . . 6
1.2 A refutation approach to interchangeability . . . 8
1.3 Breaking up Simple Tabular Reduction . . . 9
1.4 Dual representation for table constraints . . . 11
1.5 Outline of this dissertation . . . 12

2 Domain Transmutation in Constraint Satisfaction Problems 15
2.1 Abstract . . . 15
2.2 Introduction . . . 15
2.3 Background . . . 17
2.4 Domain Transmutation Procedure . . . 23
2.5 Experimental Results . . . 25
2.6 Other Applications . . . 28
2.6.1 Robustness . . . 28
2.6.2 Decision Problems . . . 28
2.6.3 Finding a Single Solution . . . 29
2.7 Related Works . . . 30
2.8 Conclusions . . . 30

3 Extracting Microstructure in Binary Constraint Networks 31
3.1 Abstract . . . 31
3.2 Introduction . . . 31
3.3 Background . . . 33
3.4 Extracting Assignments . . . 36
3.4.1 Basic algorithm . . . 37
3.4.2 Extracting cyclic assignments . . . 40
3.5 Extracting Microstructure . . . 44
3.6 Applications . . . 45
3.7 Conclusions . . . 48

4 A Refutation Approach to Neighborhood Interchangeability in CSPs 49
4.1 Abstract . . . 49
4.2 Introduction . . . 49
4.3 Preliminaries . . . 50
4.4 Overall Process . . . 51

4.5 Identifying NI by Discrimination Tree 52
4.6 Identifying NI by Refutation 54
4.6.1 Exploiting table constraints 56
4.6.2 Implementation 56
4.7 Related Works 59
4.8 Conclusions 60

5 Eliminating Redundancy in CSPs Through Merging and Subsumption of Domain Values 63
5.1 Abstract 63
5.2 Introduction 64
5.3 Joint Interchangeability 65
5.4 Enhancing Domain Values 68
5.5 Virtual Interchangeability 70
5.6 Algorithms for Merging VI values 75
5.6.1 Experimental Results 79
5.7 Identifying Redundant Values 81
5.8 Conclusions 84

6 Improving the Lower Bound of Simple Tabular Reduction 87
6.1 Abstract 87
6.2 Multistage Simple Tabular Reduction 87
6.3 Experimental Results 93
6.4 Conclusions 95

7 STR3: A Path-Optimal Filtering Algorithm for Table Constraints 97
7.1 Abstract 97
7.2 Introduction 98
7.3 Preliminaries 100
7.3.1 Constraint Satisfaction Problem 100
7.3.2 Sparse Sets 102
7.4 STR3 103
7.4.1 Structures and Features 105
7.4.2 Algorithm 107
7.5 Illustration 109
7.6 Synchronized vs. Unsynchronized Supports 111
7.7 Correctness 115
7.8 Complexity 117
7.9 STR3 with Circular Seek 119
7.10 Related Works 122
7.10.1 GAC4 122
7.10.2 AC5TC-Tr 124
7.11 Experimental Results 127
7.11.1 STR3 versus STR2: Comparison on Problem Series . . . 127
7.11.2 STR3 versus STR2: Overall Comparison 133

7.11.3 STR3 versus STR2: Comparison on Classes of Random Problems 134
7.11.4 Comparison with Other GAC Algorithms 139
7.12 Conclusions 142

List of Figures

1.1 Two equivalent CSPs. In (2), the domain of Z has been mutated. 4
1.2 A CSP and its two transformations. 5
1.3 The assignment tuple (b, d, g) in (1) has been singled out in (2) and removed in (3). 7
1.4 In (1), d is virtually interchangeable with e while f is virtually interchangeable with g. Both pairs of values are merged in (2). . 7
1.5 Standard and dual representations of a ternary constraint involving three variables X, Y, and Z. 11

2.1 Value a can be split into four fragments. 16
2.2 A CSP and its two transformations. 17
2.3 An example of the transmutation process. 21
2.4 Experimental results for transmutation: #constraint checks. . . 26
2.5 Experimental results for transmutation: #solution. 26

3.1 An introductory example: removing (b, d, g) by extraction. . . . 33
3.2 Extracting a non-cyclic assignment. 38
3.3 An example of a non-terminating transformation. 39
3.4 Extracting (b, d, f). 41
3.5 An example of more complex extraction. 41
3.6 Extracting (b, e, g, f) 44
3.7 Extracting cycle (a, b, c, d, e, f). 45

4.1 An example for the Discrimination Tree algorithm for NI. 53
4.2 Comparing the Discrimination Tree with the Refutation Tree. . 54

5.1 Three semantically equivalent networks 67
5.2 An example of network conformation 68
5.3 An example of VI-merging. 74
5.4 VI vs. BTP 75

7.1 Standard and dual representations of the relation of a ternary constraint C with scope $\{X, Y, Z\}$. 99
7.2 A sparse set S with 4 elements: $S = \{2, 4, 6, 8\}$. 102
7.3 Status right after GAC preprocessing at node η_1. Sub-tables and dependency lists are displayed vertically (at the top and the bottom of the figure, respectively). 109
7.4 Status at η_2 109
7.5 After backtracking to η_1 110
7.6 Status at η_3 110
7.7 Transition diagram depicting the relationship between the two supports of a value (X, a). STR3 is triggered only when `dep` fails. Both supports of (X, a) are synchronized as a result. 114
7.8 Data structures of GAC4 for a constraint C such that $scp(C) = \{X, Y, Z\}$. 123

7.9 Pairwise comparison (CPU time) on 2,005 instances from many series involving table constraints. The time-out to solve an instance is 1,200 seconds. 135
7.10 Comparison (CPU time) of STR2 and STR3 on instances that are respectively satisfiable and unsatisfiable. 136
7.11 Comparison (CPU time) of STR2 and STR3 on instances where avgP is respectively less than and greater than 10%. 137
7.12 Comparison (CPU time) of STR2 and STR3 on instances where avgS is respectively less than and greater than 1000. 138
7.13 The ratio "cpu STR2" to "cpu STR3" is plotted against avgS (average size of tables during search). Dots correspond to instances solved by both algorithms within MAC (from the set of 2,005 instances used as our benchmark). 139
7.14 Mean search cost of solving instances in random classes with MAC. 140
7.15 Pairwise comparison (CPU time) on 2, 005 instances from many series involving table constraints. The time-out to solve an instance is 1,200 seconds. 141

List of Tables

5.1 Tables for Example 5. 78
5.2 A merging example for 4-ary tables. 79
5.3 Experimental results for VI-merging on random problems. . . . 81
5.4 Experiment results for VI-merging on crossword puzzles. 81
5.5 An example for identifying redundant values. 82

6.1 Results from selected instances. 93
6.2 Results from pigeonhole instances. 95

7.1 Mean CPU time (in seconds) to solve instances from different series (a time-out of $1,200$ seconds was set per instance) with MAC. . 130
7.2 Detailed results on selected instances, sorted by avgP. The symbol “-” indicates a timeout. 131
7.3 Mean CPU time (in seconds) to solve Crossword instances (a time-out of $1,200$ seconds was set per instance) with MAC. 132
7.4 Results on the crossword puzzles for the *ogd08* dictionary. The symbol “-” indicates a timing out. 132
7.5 Results on the augmented pigeonhole problems. 133
7.6 Results obtained with AC5TC (version OptSparse) and MDD^c on classical series of instances involving table constraints of large arity. Values correspond to CPU times given as percentage to the best. This table is an excerpt from Table 10 in [MHD14]. 141
7.7 Mean CPU time (in seconds) to solve selected binary problems. 142

List of Algorithms

1 High-level structure of STR2 10
2 Result after breaking up the while loop of Algorithm 1 10

3 transmutate(V) 24

4 extractLine(π) 37

5 RedundancyCheck($\mathcal{X}$,$\mathcal{C}$) 52
6 CreateTupleList(X) 52
7 BuildStruct<DT>(X, tupleList) 53
8 BuildStruct<RT>(X, tupleList) 54
9 FilterStruct<RT>(V,F) 55
10 BuildDom<RT>(X,F) 55
11 BuildStruct<RTS>(X, tupleList) 58

12 Collect(X) 76
13 Compress(C) 77
14 is-onto-substitutable(X, a) 84

15 STR2(c: constraint) 88
16 STR2w(c: constraint) 89
17 tuple-elimination(c: constraint) 89
18 support-deduction(c: constraint, *prevLimit* : integer) 90
19 value-accumulation(c: constraint) 90
20 STR2w-initialization(c: constraint) 92

21 Original sparse set operations 103
22 Sparse set operations optimized for membership testing. 104
23 Algorithm STR3 107
24 Auxiliary functions of Algorithm STR3 108
25 STR3 with circular seek 121
26 Algorithm GAC4 122
27 Algorithm AC5TC-Tr 125

I, Chavalit Likitvivatanavong, certify that this thesis is my own work and I have not obtained a degree in this university or elsewhere on the basis of the work submitted in this thesis. All external references and sources are clearly acknowledged and identified within the contents. I have read and understood the regulations of this university concerning plagiarism.

Chavalit Likitvivatanavong

[illegible]

May 3, 2017

To my family

Declaration and Acknowledgements

This dissertation is in the form of a publication-based thesis. It includes six papers which were published in peer-reviewed conferences and journals, each forming one chapter.

The list of publications is given in chronological order below. Unless noted otherwise, for each paper the author of this dissertation is the principal author who (1) came up with the fundamental concept, then initiated and led the collaboration that resulted in publication (2) acted as a chief writer for the paper, and (3) performed the empirical evaluation (if there is any).

1. “Domain Transmutation in Constraint Satisfaction Problems”,
   James Bowen and Chavalit Likitvivatanavong,
   Proceedings of AAAI-04, 2004.

2. “Extracting Microstructure in Binary Constraint Networks”,
   Chavalit Likitvivatanavong and Roland H. C. Yap,
   LNAI 4651 Recent Advances in Constraints, 2007, Springer-Verlag.

3. “A Refutation Approach to Neighborhood Interchangeability in CSPs”,
   Chavalit Likitvivatanavong and Roland H. C. Yap,
   Proceedings of the 21st Australasian Joint Conference on Artificial Intelligence (AI-08), December, 2008.

4. “Eliminating Redundancy in CSPs Through Merging and Subsumption of Domain Values”,
   Chavalit Likitvivatanavong and Roland H. C. Yap,
   ACM SIGAPP Applied Computing Review, Volume 13, Issue 2, 2013.

5. “Improving the Lower Bound of Simple Tabular Reduction”,
   Christophe Lecoutre, Chavalit Likitvivatanavong, and Roland H. C. Yap,
   Constraints, Volume 20, Issue 1, Pages 100–108, January 2015.

6. “STR3: A path-optimal filtering algorithm for table constraints”,
   Christophe Lecoutre, Chavalit Likitvivatanavong, and Roland H. C. Yap,
   Artificial Intelligence, Volume 220, Pages 1–27, March 2015.

   Remark on contribution: The author of this dissertation came up with the idea of the STR3 algorithm, wrote the code, and ran initial experiments as a proof-of-concept. The experiments that appear in the paper were carried out by Christophe Lecoutre using his own solver AbsCon, for which he implemented STR3 himself.

These works have received support from Science Foundation Ireland under Grant 00/PI.1/C075 and from National University of Singapore under Grant ARF 252-000-222-112 and MOE2012-T2-1-155.

Chapter 1

Introduction

The term Constraint Satisfaction Problem (CSP) refers to a class of NP-complete problems, a collection of difficult problems for which no fast solution is known. The standard definition of a CSP involves variables, values, and constraints: each variable must be assigned a value from a designated group of possible values (also known as the variable's domain), while a constraint on a set of variables indicates permissible combinations of values for these variables. Given a CSP, an important objective is to query whether it has a solution — an assignment of each variable to a value such that all constraints are satisfied. Solving a CSP usually requires chronological backtracking search that interleaves variable assignments with various kinds of inferences in order to reduce the search space.

A large body of important combinatorial optimization and AI problems can be expressed as CSPs. These problem areas include computer vision, scheduling, resource allocation, timetabling, bin packing, configuration, and bioinformatics. Although the definition of CSP is universal, in the sense that nothing beyond the triple (variables, values, and constraints) is necessary, further extensions or specializations are often imposed on the CSP, in order to make the task of modeling problems easier and the solving process faster. For example, although a constraint may be described through explicitly listing all possible combinations of values, it may also be described implicitly by mathematical formulas or logical constructs. Some of the constraints invented for particular purposes are counting constraints [Ŕ96], the regular language membership constraint [Pes04], the AllDifferent constraint [Rég94], the Among constraint [BC94], the Element constraint [HC88]. Variations of CSPs also exist, such as soft CSPs [BMR+99], partial CSPs [FW92], distributed CSPs [YDIK92], among others. One crucial advantage of specialized constraints is that they also allow specialized algorithms to be invented. These algorithms take advantage of the extra properties of the constraints and are often much faster than the ones that work on the general CSP model.

Specialized constraints together with other tools for mapping problems into CSPs constitute the core of what is now called Constraint Programming (CP), a form of declarative programming whose aim is to help ease the process of problem solving by letting users focus more on modeling and less on the implementation aspects. Different types of CP systems have been created, ranging from basic solvers to large systems that include modeling and programming language capability, for example, AbsCon [MLB01], Gecode [Gec15], Choco [PFL14], JaCoP [JaC15], ILOG [IBM14], and MiniZinc [NSB+07]. Another approach, Constraint Logic Programming, is an extension of Logic Programming to include constraints (see [JM94] for a survey).

Despite the emergence of CP, the basic general CSP formulation continues to be relevant as the fallback model for those cases where the problem in question cannot be encoded efficiently by other means. For instance, in configuration problems constraints often defy neat classification: the reason why one item is compatible with some components but not with others can be a matter of preferences that cannot be captured logically. As a result, the only way to model these constraints is to enumerate compatible items explicitly.

In this dissertation, we study only the classical CSP model and the fundamental algorithms for solving it. Specifically, we only consider the case where constraints are explicitly given by listing permissible combinations of values — this type of constraint is called an extensional or a table constraint. Despite being an important class, extensional constraints have received much less attention recently as most efforts have been channeled toward identifying new types of specialized constraints and coming up with corresponding algorithms. Regardless, improvements to algorithms for extensional constraints are fundamental and this dissertation will show that there is still ample opportunity for improvement in this area. By concentrating on extensional constraints, this dissertation will attempt to improve existing techniques and algorithms by examining them critically from the bottom up and approaching them from a novel direction. Sections 1.1–1.4 of this introduction chapter briefly explain various ideas that will be developed in this dissertation as well as previewing what will be studied in the upcoming chapters.

This dissertation is in publication-based format. Each chapter is reproduced from a previously refereed and published paper with adjustment for layout and formatting. No changes to contents are made except for Chapter 5, which was revised and expanded to remove some minor errors and improve clarity. Each chapter is self-contained, with its own introduction, background, and conclusions sections. Chapters are arranged in order of publication date to reflect how this research evolved over the years.

1.1 A new perspective on domain values

The first and major focus of this dissertation is on the representation of domain values in the classical formulation of the CSP. This section gives an overview of a new perspective on domain values that will form the basis for later chapters. Traditionally, each domain value has been regarded as atomic. However, we will show that domain values are essentially mutable: we can break domain values down to smaller units and recombine them to form new values.

Domain values serve two purposes: structural and semantic. These two aspects are intertwined in most CSP models, yet we would have more flexibility in manipulating a CSP if they were decoupled. To this end, a value is viewed as being composed of a set of *labels* (the ways the value can be expressed in a solution) and its *support structure* (its relations, through connecting constraints, to values in the domains of adjacent variables.)

Figure 1.1(1) illustrates an example CSP in microstructure format. (Binary CSPs will be depicted in this format throughout this dissertation.) Our convention is to use a dot to represent an individual domain value, a dashed enclosure with dots inside to represent a variable domain, and a solid line between two dots to show that the two values those two dots represent are compatible. The label(s) for each value is/are placed next to the dot which represents the value. The name of a variable is placed next to the dashed enclosure which contains the dots representing the values in the domain of the variable. To avoid cluttering, universal constraints, which forbid nothing, are not depicted in any diagram. There are three variables (X, Y, and Z) in the CSP in Figure 1.1(1); the domain of X is $\{d, e\}$, the domain of Y is $\{f, g\}$, and the domain of Z is $\{a, b, c\}$. There are four solutions to this CSP: the possible assignment tuples for the variable triple (X, Y, Z) are (d, a, f), (e, a, f), (e, b, g), and (e, c, g).

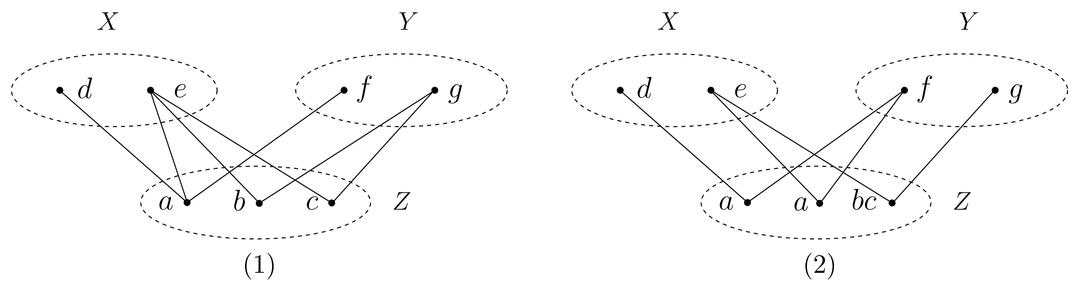


Figure 1.1: Two equivalent CSPs. In (2), the domain of Z has been mutated.

Figure 1.1(2), which shows another CSP that is equivalent to Figure 1.1(1) in term of solutions, illustrates two main features of the new approach to domain values: different values can have the same label, while some values can have multiple labels. In Figure 1.1(2), there are two values with the same singleton label set $\{a\}$ (the two dots marked with "a" in the figure) and there is one value with a non-singleton set of labels[1] $\{b, c\}$ (the dot marked with "bc").

One characteristic of the older CSP representation in Figure 1.1(1) is that each value has a unique singleton label set, which allows us to refer to a value by its label alone. By contrast, in Figure 1.1(2), in order to precisely and formally refer to a specific domain value, we will use an ordered pair that has the value's labels as the first component and a cross-product of compatible values as the second component, to indicate the value's support structure. For instance, in Figure 1.1(2), the two values with the same label set $\{a\}$, are formally denoted by $(\{\boldsymbol{a}\}, \{d\}\times\{f\})$ and $(\{\boldsymbol{a}\}, \{e\}\times\{f\})$, whereas the value with the set of labels $\{b, c\}$ is referred to as $(\{\boldsymbol{b,c}\}, \{e\}\times\{g\})$.

A complex assignment tuple (that is one which involves a value with multiple labels) is equivalent to multiple simple assignment tuples, in which each value has only a single label. For instance, consider value bc (from now on we will simply refer to a value by its label set if the set is unique): the only complex assignment tuple involving bc is (e, bc, g), which in turn is equivalent to two simple assignment tuples (e, b, g) and (e, c, g).

This new perspective on domain values underpins three chapters: Chapter 2, Chapter 3, and Chapter 5.[2] The key ideas of these chapters are previewed in

[1]To avoid cluttering our diagrams, we simply represent a set of labels by concatenating its members without adding any extra symbol — e.g. a stands for $\{a\}$ and bc stands for $\{b, c\}$.

[2]Chapter 4 does not require the new viewpoint on domain values. It comes before Chapter

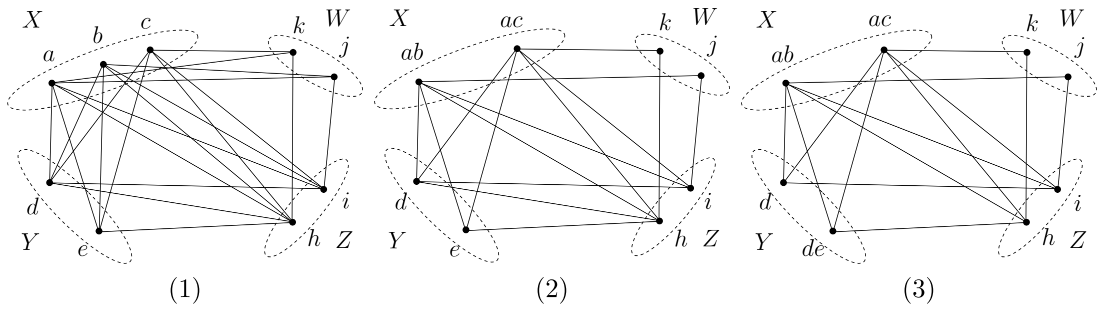


Figure 1.2: A CSP and its two transformations.

subsections 1.1.1–1.1.3.

1.1.1 Domain transmutation

Chapter 2 introduces domain transmutation,[3] the process through which domain values are split and recombined in order to eliminate all duplicate value fragments. Domain transmutation can produce CSPs with smaller search spaces, leading to shorter solving times.

Figure 1.2 illustrates. Figure 1.2(1) shows a CSP in microstructure format, while Figure 1.2(2) shows the result of splitting and merging value fragments in the domain of variable X. Figure 1.2(3) shows the result of subsequently transmutating the domain of Y. All three CSPs have the same solution set. This process is explained in detail below.

Consider the three values a, b, and c for X in Figure 1.2(1). Using the label and support structure notation, these three values are denoted by $(\{\boldsymbol{a}\}, \{d,e\}\times\{h,i\}\times\{j,k\})$, $(\{\boldsymbol{b}\}, \{d,e\}\times\{h,i\}\times\{j\})$ and $(\{\boldsymbol{c}\}, \{d,e\}\times\{h,i\}\times\{k\})$. Value a can be split into two fragments: $(\{\boldsymbol{a}\}, \{d,e\}\times\{h,i\}\times\{j\})$ and $(\{\boldsymbol{a}\}, \{d,e\}\times\{h,i\}\times\{k\})$. The support structure of the first fragment of a is identical to that of b, while the support structure of the second fragment of a is identical to that of c. These two pairs of values can then be merged together by combining their labels. Thus, after merging we obtain two new values: $(\{\boldsymbol{a,b}\}, \{d,e\}\times\{h,i\}\times\{j\})$ and $(\{\boldsymbol{a,c}\}, \{d,e\}\times\{h,i\}\times\{k\})$, depicted as ab and ac in Figure 1.2(2).

5 because the paper it is based on was published before that of Chapter 5, in keeping with the chronological ordering.

[3]Transmutation — a term from Nuclear Chemistry — describes a change in the number of protons in the nucleus of an atom, producing an atom with a different atomic number.

Similarly, consider values d and e for Y in Figure 1.2(2). Using the label and the support structure notation, these values are denoted by ({$\boldsymbol{d}$}, {*ab,ac*}×{*h,i*}) and ({$\boldsymbol{e}$}, {*ab,ac*}×{*h*}). Value d can be split into two fragments: ({$\boldsymbol{d}$}, {*ab,ac*}×{*h*}) and ({$\boldsymbol{d}$}, {*ab,ac*}×{*i*}). The first fragment can be combined with e. The result is shown in Figure 1.2(3). No duplicate value fragments exist at this point.

1.1.2 Extracting microstructure

In Chapter 3, the notion of value fragment is generalized to include assignment tuples that span multiple variables. This allows an assignment tuple to be unraveled and extracted from a CSP's microstructure. Chapter 3 gives the algorithm that performs this task and proves that it is sound and complete.

Figure 1.3(1) illustrates the idea. Suppose we want to designate the assignment tuple (b, d, g) for (X, Y, Z) as a non-solution. The usual approach would be to add a new ternary constraint that disallows (b, d, g). It is unsafe to directly delete one of b, d, or g from the domains since solutions involving them may be inadvertently eliminated in the process.

Figure 1.3(2) shows an equivalent CSP in which some domain values in Figure 1.3(1) are split. The solutions of the CSP in Figure 1.3(1) are the same as those of the CSP in Figure 1.3(2). The assignment tuple (b, d, g) in Figure 1.3(2), however, is now singled out (by the process that will be explained in Chapter 3) and thus we can simply remove ({$\boldsymbol{b}$}, {*d*}×{*g*}) or ({$\boldsymbol{d}$}, {*b*}×{*g*}) or ({$\boldsymbol{g}$}, {*b*}×{*d*}) from the CSP to mark (b, d, g) as a non-solution — removing one is enough because the other two values would be removed by consistency processing. Figure 1.3(3) depicts the resulting CSP in which the non-solution (b, d, g) has been eliminated. The CSP remains a binary CSP — there was no need to introduce a ternary constraint.

1.1.3 Virtual interchangeability

Two values are interchangeable [Fre91] if replacing one with the other in a solution produces another solution. Given a group of interchangeable values, keeping one value and eliminating the rest helps simplify the CSP without affecting its satisfiability.

Chapter 5 proposes the concept of virtual interchangeability (VI), a type of

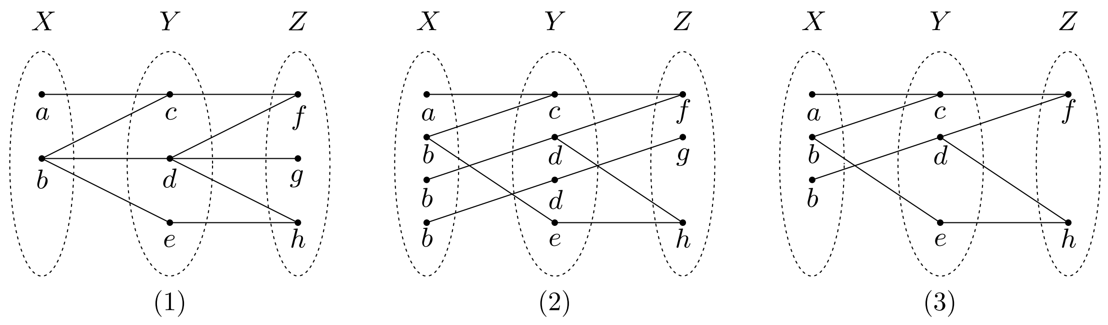


Figure 1.3: The assignment tuple (b, d, g) in (1) has been singled out in (2) and removed in (3).

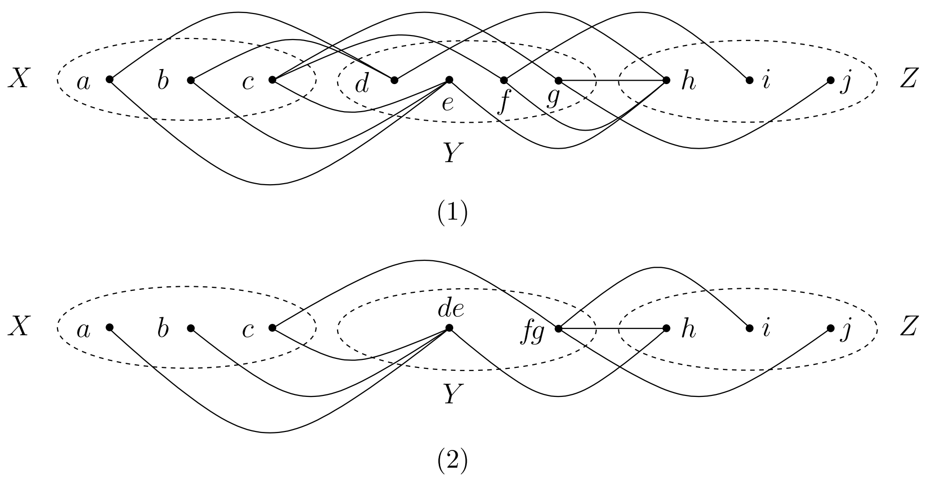


Figure 1.4: In (1), d is virtually interchangeable with e while f is virtually interchangeable with g. Both pairs of values are merged in (2).

interchangeability that can be detected locally. Values are virtually interchangeable if their support structures are almost identical — that is, they are the same except for one connecting constraint. Chapter 5 will show that a group of virtually interchangeable values can be compactly represented by just a single value. Several researchers, [Coo14, CMTZ14, CDE15], have recently compared VI to related concepts such as forbidden patterns, the broken-triangle property, and variable and value elimination. In particular, it is worth noting that the merging of a group of VI values into one has inspired the BTP-merging rule in [CMTZ14].

As an example, consider the domain of Y in the CSP in Figure 1.4(1). Value d is virtually interchangeable with value e, while value f is virtually interchangeable with value g. The CSP in Figure 1.4(2) is modified from that in Figure 1.4(1) by merging d with e and merging f with g (though union of label sets). The merging preserves solutions but also introduces spurious ones. For instance, of the two simple assignment tuples that are represented by the complex assignment tuple (c, de, h) in Figure 1.4(2), only one, (c, e, h), is a consistent assignment for the original CSP; the other, (c, d, h) is spurious — it is not a consistent assignment tuple of the original CSP. The crucial point is that, with respect to VI, having multiple labels does not necessarily mean that every one of them is legitimate; VI only guarantees that at least one of them is. But this is enough to ensure that a VI-compressed variant of a CSP (another CSP derived by merging some VI values in the original) is equivalent, in terms of satisfiability, to the original CSP — each of them admits a consistent assignment tuple if and only if the other one also does.

To obtain, from a complex assignment tuple which satisfies a VI-compressed CSP, a consistent simple assignment tuple for the original CSP, each label in the tuple for the VI-compressed CSP has to be examined against the original CSP. However, this checking phase is computationally inexpensive and can, of course, be eliminated if the sole objective is to ascertain whether the original CSP is satisfiable or not.

Given that (c, de, h) is a solution to the CSP in Figure 1.4(2), we know a solution to the CSP in Figure 1.4(1) exists according to the property of VI. If the goal is to find out whether the CSP in Figure 1.4(1) is satisfiable, then we can stop here, knowing that it is. If a solution is required as well, one can be obtained from the complex assignment tuple (c, de, h). This is done by generating the two simple assignment tuples that are implicit in (c, de, h) — that is, (c, d, h) and (c, e, h) — and checking their consistencies against the CSP in Figure 1.4(1). Only (c, e, h) is consistent; therefore, (c, e, h) is a solution of the original CSP.

1.2 A refutation approach to interchangeability

Neighborhood interchangeability (NI) [Fre91] is a type of local interchangeability and was the first local interchangeability to be proposed. Two values are neighborhood interchangeable if their support structures are identical. For example, in Figure 1.1(1), values b and c are NI with each other. Removing or grouping neighborhood interchangeable values together eliminates redundancy and improve overall solving time [BCZ01, LCF05, WF99, Has93].

NI can be detected in quadratic time by the discrimination tree algorithm (DT)

[Fre91]. DT works by assuming zero knowledge and by building up relationships between values. One disadvantage is that determining whether two values are NI with each other requires all values in the adjacent variables to be tested.

Chapter 4 proposes a different method that is able to detect values that are not NI early on, without checking all the values. Instead of testing whether values are NI, we assume they all are NI in the beginning and try to disprove this assumption when new information comes along. When enough is known about a value such that the value is certain not to be NI with any other value, it can be immediately removed from future consideration. The new algorithm is proved to have a better lower bound than the DT algorithm.

1.3 Breaking up Simple Tabular Reduction

Since the difficulty of a CSP correlates with the size of its search space, a common solving strategy involves removing inconsistent parts of the CSP to help reduce the search space. Generalized arc consistency (GAC) algorithms remove, from the domain of a variable, any value that supports no compatible value in the domain of a connected variable. GAC algorithms are generally a key component of many CSP solvers due to the effectiveness and long research history of the approach.

Simple Tabular Reduction (STR)[Ull07], in particular in its optimized form STR2 [Lec11], is a state-of-the-art GAC algorithm for table constraints. Before the development of STR, conventional GAC algorithms on table constraints shared two important traits. First, compatible tuples were actively sought for each domain value in order to establish that the value was arc consistent. Second, the table was static throughout the search; the only changes to the CSP occurred in variable domains.

STR differs from these algorithms by not trying to systematically search for a specific tuple, but by processing the whole table to maintain the property that every tuple must be valid (that is, each tuple involves only values that are currently present in the domains at the time the algorithm is invoked.) Tuples that are found to be invalid are immediately removed. Domain values are collected from the remaining tuples and any value that is not produced during this collection process must therefore be inconsistent. STR2 improves on STR by skipping over some columns during the traversal of the table.

Algorithm 1: High-level structure of STR2

```
while somecondition do
    ...
    ...
    ...
...
if somecondition then exit with flag1
...
if somecondition then exit with flag2
...
exit with flag3
```

Because a value's presence in its domain must be re-checked every time STR2 is invoked on a constraint, STR2 has a fixed lower bound equal to the sum of the domain size of all the variables involved in the constraint. The cost of this task takes up more of STR2's execution time as domains become larger or become more resistant to reduction during backtracking search.

The lower bound can be recognized simply by looking at STR2's structure, which is shown in Algorithm 1 (irrelevant details are hidden). There are three points where STR2 can exit (the lines containing *flag1*, *flag2*, *flag3* in the algorithm), all of which come after the while loop at the beginning of the algorithm. It is clear that the work done by the loop is unavoidable and thus makes up STR2's lower bound. Chapter 6 will show that the while loop can be broken up into three smaller loops and then rearranged into inter-dependent phases so that the algorithm can exit even when the original loop is only partially finished, thereby improving the lower bound of the algorithm. Algorithm 2 illustrates.

Algorithm 2: Result after breaking up the while loop of Algorithm 1

```
while somecondition do
    ...
if somecondition then exit with flag1
while somecondition do
    ...
if somecondition then exit with flag2
while somecondition do
    ...
exit with flag3
```

	X	Y	Z
1	a	f	l
2	b	f	m
3	e	g	m
4	a	f	m
5	b	g	o
6	a	h	o
7	d	h	o
8	b	i	n
9	c	j	k

(1) Standard table

X		Y		Z	
a	{1,4,6}	f	{1,2,4}	k	{9}
b	{2,5,8}	g	{3,5}	l	{1}
c	{9}	h	{6,7}	m	{2,3,4}
d	{7}	i	{8}	n	{8}
e	{3}	j	{9}	o	{5,6,7}

(2) Dual table

Figure 1.5: Standard and dual representations of a ternary constraint involving three variables X, Y, and Z.

1.4 Dual representation for table constraints

Chapter 7 proposes dual representation for table constraints and introduces STR3, an optimal GAC algorithm that works on this representation.

As stated in Section 1.3, the idea behind Simple Tabular Reduction is to remove invalid tuples from tables as soon as possible in a systematic fashion. STR3 is based on the same principle as STR [Ull07] and STR2 [Lec11] but employs a different representation of table constraints. Similar to a few other algorithms (e.g., GAC-allowed [BR97] and GAC-va [LS06]), STR3 provides an index for each constraint table, enabling a tuple sought with respect to a domain value to be found without visiting irrelevant tuples, thus reducing time complexity. Figure 1.5 shows an example for a ternary constraint. Importantly, for each constraint, STR3 maintains some specific data structures designed so that no constraint tuple is processed more than once along any path through the search tree, going from the root to a leaf.

Most of the GAC algorithms for table constraints previously introduced in the literature suffer from repeatedly traversing the same tables or related data structures during search [Lec11, CY10]. In contrast, STR3 avoids such repetition and is path-optimal: each element of a table is examined at most once along any path of the search tree. An important feature of STR3 is that it is designed specifically to be interleaved with backtracking search, where the goal is to maintain the consistency while minimizing the cost of backtracking. As such, unlike most other GAC algorithms, STR3 is only applicable within the context of search: STR3 maintains GAC, but before commencement of search, GAC must be enforced by some other algorithm, such as STR2 for example.

1.5 Outline of this dissertation

In summary, this dissertation is organized as follows:

- Chapter 1: Introduction
- Chapter 2: Domain Transmutation in Constraint Satisfaction Problems.
  This chapter is reproduced from the following publication:

  "Domain Transmutation in Constraint Satisfaction Problems",
  James Bowen and Chavalit Likitvivatanavong,
  Proceedings of AAAI-04, 2004.
- Chapter 3: Extracting Microstructure in Binary Constraint Networks.
  This chapter is reproduced from the following publication:

  "Extracting Microstructure in Binary Constraint Networks",
  Chavalit Likitvivatanavong and Roland H. C. Yap,
  LNAI 4651 Recent Advances in Constraints, 2007, Springer-Verlag.
- Chapter 4: A Refutation Approach to Neighborhood Interchangeability in CSPs.
  This chapter is reproduced from the following publication:

  "A Refutation Approach to Neighborhood Interchangeability in CSPs",
  Chavalit Likitvivatanavong and Roland H. C. Yap,
  Proceedings of the 21st Australasian Joint Conference on Artificial Intelligence (AI-08), December, 2008.
- Chapter 5: Eliminating Redundancy in CSPs Through Merging and Subsumption of Domain Values.
  This chapter is a revision of the following publication:

  "Eliminating Redundancy in CSPs Through Merging and Subsumption of Domain Values",
  Chavalit Likitvivatanavong and Roland H. C. Yap,
  ACM SIGAPP Applied Computing Review, Volume 13, Issue 2, 2013.
- Chapter 6: Improving the Lower Bound of Simple Tabular Reduction.
  This chapter is reproduced from the following publication:

  "Improving the Lower Bound of Simple Tabular Reduction",
  Christophe Lecoutre, Chavalit Likitvivatanavong, and Roland H. C. Yap,
  Constraints, Volume 20, Issue 1, Pages 100–108, January 2015.

- Chapter 7: STR3: A path-optimal filtering algorithm for table constraints.
  This chapter is reproduced from the following publication:

  "STR3: A path-optimal filtering algorithm for table constraints", Christophe Lecoutre, Chavalit Likitvivatanavong, and Roland H. C. Yap, *Artificial Intelligence*, Volume 220, Pages 1–27, March 2015.

Chapter 2

Domain Transmutation in Constraint Satisfaction Problems

2.1 Abstract

We study local interchangeability of values in constraint networks based on a new approach where a single value in the domain of a variable can be treated as a combination of "sub-values". We present an algorithm for breaking up values and combining identical fragments. Experimental results show that the transformed problems take less time to solve for all solutions and yield more compactly-representable, but equivalent, solution sets. We obtain new theoretical results on context dependent interchangeability and full interchangeability, and suggest some other applications.

2.2 Introduction

Interchangeability of domain values was first reported by [Fre91], where neighborhood interchangeability (NI) and neighborhood substitutability (NS) are identified as local properties that can be exploited to remove redundant values, either as a preprocessing step or during search [BF92, Has93]. NI was later extended to cover interchangeable values in different variables [CN98].

We present a new approach to local interchangeability in this chapter. Normally, each domain value is treated as atomic. Our idea is to "split the atom" — a domain value can be split into several "sub-values" (Figure 2.1) so

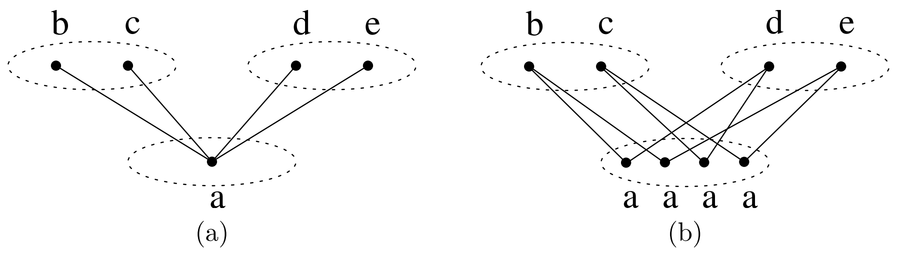


Figure 2.1: Value *a* can be split into four fragments.

that interchangeable fragments from other values can then be merged to avoid duplicate search effort during the solving process.

Figure 2.2 illustrate. The problems depicted are in microstructure form: each edge connects compatible values in the domains of two variables. Figure 2.2a shows the original problem, while 2.2b shows the result of splitting and merging sub-atomic value fragments in the domain of X (a process which, by analogy with Nuclear Chemistry,[1] we call *domain transmutation*). Figure 2.2c shows the result of subsequently transmutating the domain of Y.

Consider, for example, the values a, b and c shown in 2.2a. Value a supports the following tuples in $Y{\times}Z{\times}W$: (d,h,j), (d,h,k), (d,i,j), (d,i,k), (e,h,j), (e,h,k), (e,i,j) and (e,i,k). We denote this support relationship concisely by $(\{\boldsymbol{a}\}, \{d,e\}\times\{h,i\}\times\{j,k\})$. The supports of b and c are $(\{\boldsymbol{b}\}, \{d,e\}\times\{h,i\}\times\{j\})$ and $(\{\boldsymbol{c}\}, \{d,e\}\times\{h,i\}\times\{k\})$. No two of the values a, b and c are NI. However, a is neighborhood-substitutable for b and also for c.

The value a can be split into two fragments: $(\{\boldsymbol{a}\}, \{d,e\}\times\{h,i\}\times\{j\})$ and $(\{\boldsymbol{a}\}, \{d,e\}\times\{h,i\}\times\{k\})$. The first fragment of a is interchangeable with b, while the second fragment of a is interchangeable with c. Instead of eliminating one of the interchangeable fragments, we merge them together and use their combined labels as the new value's label, to indicate its origin. Thus, after merging we have two values: $(\{\boldsymbol{a,b}\}, \{d,e\}\times\{h,i\}\times\{j\})$ and $(\{\boldsymbol{a,c}\}, \{d,e\}\times\{h,i\}\times\{k\})$, depicted as ab and ac in 2.2b.

Similarly, consider the values d and e in 2.2b. The supports of d and e are $(\{\boldsymbol{d}\}, \{ab,ac\}\times\{h,i\})$ and $(\{\boldsymbol{e}\}, \{ab,ac\}\times\{h\})$. However, d can be split into two fragments: $(\{\boldsymbol{d}\}, \{ab,ac\}\times\{h\})$ and $(\{\boldsymbol{d}\}, \{ab,ac\}\times\{i\})$. The first fragment is interchangeable with e. The result is 2.2c. No two values from the same domain intersect in this problem.

[1]Transmutation — a change in the number of protons in the nucleus of an atom, producing an atom with a different atomic number.

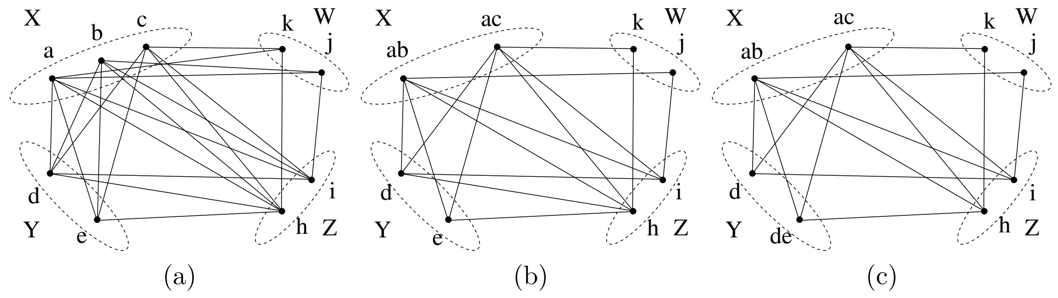


Figure 2.2: A CSP and its two transformations.

As demonstrated, our approach subsumes both NI and NS, but it does so by applying the NI principle to value-*fragments*. Indeed, while standard NS cannot preserve the solution set of a CSP, our approach can, precisely because it treats value-substitutability as fragment-interchangeability. For example, the solution *ac*-*de*-*h*k in 2.2c corresponds to four solutions in the original problem: (*a*,*d*,*h*,*k*), (*a*,*e*,*h*,*k*) (*c*,*d*,*h*,*k*), and (*c*,*e*,*h*,*k*).

We will show empirically that domain transmutation can reduce the amount of time needed to find all solutions, as well as enable more compact representations of solution sets. This is essential in certain interactive constraint-based applications, such as interactive design-advice systems [O'S02]: sometimes, to make his next interactive decision, a design engineer needs to be able to examine *all* consistent assignments to the variables involved in a sub-region of the overall CSP, and to see how similar they are to each other. Another application involves compiling CSPs [WF99]: all solutions to a sub-problem are required in order to create a meta-variable, after which NI is employed to reduce domain size.

2.3 Background

We focus on binary CSPs since non-binary constraints can be converted into binary constraints.

Definition 1 (Binary CSP) *A* binary CSP $\mathcal{P}$ *is a triplet* $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ *where* $\mathcal{V}$ *is the finite set of variables,* $\mathcal{D} = \bigcup_{V \in \mathcal{V}} D_V$ *where* D_V *is the finite set of possible values for* V *(sometimes we denote* D_V *by* $\mathcal{D}(V)$*), and* $\mathcal{C}$ *is a finite set of constraints such that each* $C_{XY} \in \mathcal{C}$ *is a subset of* $D_X \times D_Y$ *indicating the compatible pairs of values for* X *and* Y*, where* $X \neq Y$*. If* $C_{XY} \in \mathcal{C}$*, then* $C_{YX} = \{(y, x) \mid (x, y) \in C_{XY}\}$ *is also in* $\mathcal{C}$*. The* neighborhood *of variable* V*,* *denoted by N_V, is the set $\{W \mid C_{VW} \in \mathcal{C}\}$.*

A partial assignment *is a function $\pi : \mathcal{W} \subseteq \mathcal{V} \to \mathcal{D}$ such that $\pi(W) \in D_W$ for all $W \in \mathcal{W}$; we denote the function domain by* $\mathrm{dom}(\pi)$. *π is* consistent *iff for all $C_{XY} \in \mathcal{C}$ such that $X, Y \in \mathrm{dom}(\pi)$, $(\pi(X), \pi(Y)) \in C_{XY}$; we denote consistency of π by* $\mathrm{cons}(\pi)$. *π is a* solution *iff* $\mathrm{dom}(\pi) = \mathcal{V}$ *and* $\mathrm{cons}(\pi)$.*The set of all solutions for $\mathcal{P}$ is denoted by* $\mathrm{Sols}(\mathcal{P})$. *We use $\mathcal{P}|_{\mathcal{W}}$ to denote the CSP induced by $\mathcal{W} \subseteq \mathcal{V}$.*

Given a partial assignment π, we define $\pi[a/b]$ to be the partial assignment where $\pi[a/b](V) = b$ if $\pi(V) = a$; otherwise $\pi[a/b](V) = \pi(V)$.

At this point we extend the usual definition of a value to a 2-tuple in order to clearly distinguish between its syntax (label) and semantics (support). This lower-level detail is only of theoretical concern and can be safely ignored in other contexts.

Definition 2 (Values) *A* value *a is a 2-tuple (L, σ) where L is a set of* labels, *while σ, called* support, *is a function $\sigma : N_V \to 2^{\mathcal{D}}$ such that $\sigma(W) \subseteq D_W$ for any $W \in N_V$. We use L_a to denote the set of labels of a and use σ_a to denote the support of a. A value a in $\mathcal{D}$ of the CSP $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ must be* valid, *that is, $\sigma_a(W) = \{b \in D_W \mid (a, b) \in C_{VW}\}$ for any $W \in N_V$.*

Let $\mathcal{L} = \bigcup_{a \in \mathcal{D}} L_a$ be the set of all labels. A partial label assignment *is a function $\lambda : \mathcal{W} \subseteq \mathcal{V} \to \mathcal{L}$ such that $\lambda(W) \in \bigcup_{a \in D_W} L_a$ for all $W \in \mathcal{W}$. Given a partial assignment π, we denote π_{label} to be the set of partial label assignments $\{\lambda \mid \lambda(V) \in L_{\pi(V)}\}$. For any set of partial assignments $\mathcal{X}$, we use $\mathcal{X}_{\mathrm{label}}$ to denote $\bigcup_{\pi \in \mathcal{X}} \pi_{\mathrm{label}}$.*

The new definition allows a variable domain to contain values having the same label but differrent supports (e.g. Figure 2.1b). Note that NI is still allowed: it is possible to have values with the same supports but different labels. However, values having both the same labels and supports are not permitted. In practice, when structure can no longer be used to differentiate values, each label can still be made unique and yet indicate its origin simply by appending to it some suffix. (e.g. r_1 and r_2 in Figure 2.3b). A solution can be converted back to the original format in time $O(|\mathcal{V}|)$.

Each value can have multiple labels so that we can combine NI values without losing any solution (unlike the standard practice where only one value is kept). For example $(\{\boldsymbol{a}\}, \{x\}{\times}\{y\})$ and $(\{\boldsymbol{b}\}, \{x\}{\times}\{y\})$ can be replaced by a single value ($\{\boldsymbol{a,b}\}$, $\{x\}\times\{y\}$). Any solution π involving ($\{\boldsymbol{a,b}\}$, $\{x\}\times\{y\}$) can be converted using π_{label}. Throughout this chapter, we always assume CSPs with no NI values since NI values are trivially handled by this approach.

Given a CSP we define the following operations on values, analogous to the usual operations on sets.

Definition 3 (Operations on Values) *Let a and b be two values in D_V.*

The intersection *of a and b (denoted by $a \odot b$) is a value c where $L_c = L_a \cup L_b$ and $\sigma_c(W) = \sigma_a(W) \cap \sigma_b(W)$ for all $W \in N_V$. The intersection is undefined (denoted by $a \odot b = \emptyset$) if there exists a variable $X \in N_V$ such that $\sigma_a(X) \cap \sigma_b(X) = \emptyset$. Two values a and b, are said to be* disjoint *if $a \odot b = \emptyset$. A set of values is disjoint if its members are pairwise disjoint.*

The union *of a and b (denoted by $a \oplus b$) is a value c where $L_c = L_a$ and $\sigma_c(W) = \sigma_a(W) \cup \sigma_b(W)$ for all $W \in N_V$. The union is undefined (denoted by $a \oplus b = \emptyset$) if $L_a \neq L_b$ or there exist two or more variables $X \in N_V$ such that $\sigma_a(X) \neq \sigma_b(X)$. The* subtraction *of b from a (denoted by $a \ominus b$) is the smallest set[2] of disjoint values $\mathscr{C}$ such that $\bigoplus(\mathscr{C} \cup \{a \odot b\}) = a$.*

Value a is subsumed[3] *by b (denoted by $a \sqsubseteq b$) if $\sigma_a(W) \subseteq \sigma_b(W)$ for all $W \in N_V$.*

We will continue to use cross-product representation to denote supports for a value. For instance, consider values ($\{\boldsymbol{a}\}$, $\{d,e\}\times\{h,i\}\times\{j,k\}$) and ($\{\boldsymbol{b}\}$, $\{d,e\}\times\{h,i\}\times\{j\}$), which are depicted by their labels a and b in Figure 2.2a. Their intersection is ($\{\boldsymbol{a,b}\}$, $\{d,e\}\times\{h,i\}\times\{j\}$). Another example: ($\{\boldsymbol{x,y}\}$, $\{a,b\}\times\{c\}$) $\odot$ ($\{\boldsymbol{y,z}\}$, $\{a\}\times\{c,d\}$) = ($\{\boldsymbol{x,y,z}\}$, $\{a\}\times\{c\}$).

It should be emphasized that the result of subtraction may be a *set* of values. Consider ($\{\boldsymbol{a}\}$, $\{d,e\}\times\{f,g\}$) $\ominus$ ($\{\boldsymbol{b}\}$, $\{d\}\times\{f\}$). The result is a set $\{$($\{\boldsymbol{a}\}$, $\{e\}\times\{f,g\}$), ($\{\boldsymbol{a}\}$, $\{d\}\times\{g\}$)$\}$. (This kind of fragmentation is not uncommon. On a larger scale, extracting subproblems can also fracture the remainder of a CSP [FH95].)

Lemma 1 *Let a and b be values in D_V and let $W \in N_V$.*

(1) Let $S \subseteq D_W$, then $S \subseteq \sigma_a(W)$ iff $a \in \bigcap_{d \in S} \sigma_d(V)$.

(2) $\mathscr{C} = \{a \odot b\} \cup (a \ominus b) \cup (b \ominus a)$ is disjoint[4]. Moreover,

[2]Subtraction is unique, although we do not have enough space to provide the proof.
[3]a is neighborhood substitutable for b if and only if $b \sqsubseteq a$.
[4]Recall that the subtraction of two values may result in a collection of values.

$\{\pi \mid \mathit{cons}(\pi) \wedge (\mathit{dom}(\pi) = \{V\} \cup N_V) \wedge (\pi(V) \in \{a, b\})\}_{\text{label}} =$
$\{\pi \mid \mathit{cons}(\pi) \wedge (\mathit{dom}(\pi) = \{V\} \cup N_V) \wedge (\pi(V) \in \mathscr{C})\}_{\text{label}}.$

(3) $\odot$ *and* $\oplus$ *are idempotent, commutative, and associative.*

We now formally define domain transmutation.

Definition 4 (Transmutation) *Given a CSP* $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$*. For any* $V \in \mathcal{V}$*, a* transmutation *of* D_V *(denoted by* $\text{trans}(D_V)$*) is a variable domain* D'_V *such that,*

(1) D'_V *is disjoint.*

(2) $\text{Sols}(\mathcal{P}|_{\{V\}\cup N_V})_{\text{label}} = \text{Sols}(\mathcal{P}'|_{\{V\}\cup N_V})_{\text{label}}$ *, where* $\mathcal{P}'$ *is the CSP in which* D_V *is replaced by* D'_V *and all constraints involving* V *have been updated*[5] *by values in* D_V*. We denote* $\mathcal{P}'$ *by* $\text{trans}(\mathcal{P}, V)$*.*

In a domain transmutation, no two values overlap and all the solutions involving the original domain are preserved. An algorithm for transmutating a domain will be given in the next section. Effects of domain transmutation are shown in the following theorem, whose proof is omitted.

Theorem 1 (Transmutation Effects) *Given a CSP* $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$*, let* $\text{trans}(\mathcal{P}, V) = (\mathcal{V}, \mathcal{D}', \mathcal{C}')$*,*

(1) If $b \in \mathcal{D}'(V)$*, then there exists some* $a \in \mathcal{D}(V)$ *such that* $b \sqsubseteq a$ *and such that, for any solution* π *for* $\text{trans}(\mathcal{P}, V)$ *where* $\pi(V) = b$*, there also exists the solution* $\pi[b/a]$ *for* $\mathcal{P}$*.*

(2) If π *is a solution for* $\mathcal{P}$ *such that* $\pi(V) = a$*, then there is* exactly one $c \in \mathcal{D}'(V)$ *,* $c \sqsubseteq a$*, such that* $\pi[a/c]$ *is a solution for* $\text{trans}(\mathcal{P}, V)$*.*

Proposition 1 (Upper Bound on Search Space) *Given a CSP* $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$*, we denote* $\text{maxassign}(\mathcal{P})$ *to be the number of possible partial assignments* π *such that* $\text{dom}(\pi) = \mathcal{V}$*. For any binary CSP* $\mathcal{P}$*,* $\text{maxassign}(\text{trans}(\mathcal{P}, V)) \leqslant \text{maxassign}(\mathcal{P})$ *for all* $V \in \mathcal{V}$*.*

In the rest of this section, we will obtain new results on other types of interchangeability based on transmutation.

Context dependent interchangeability (CDI) was proposed in [WFC96] in an attempt to capture interchangeability in a limited situation. CDI is rephrased as follows.

[5]To make values in D'_V valid.

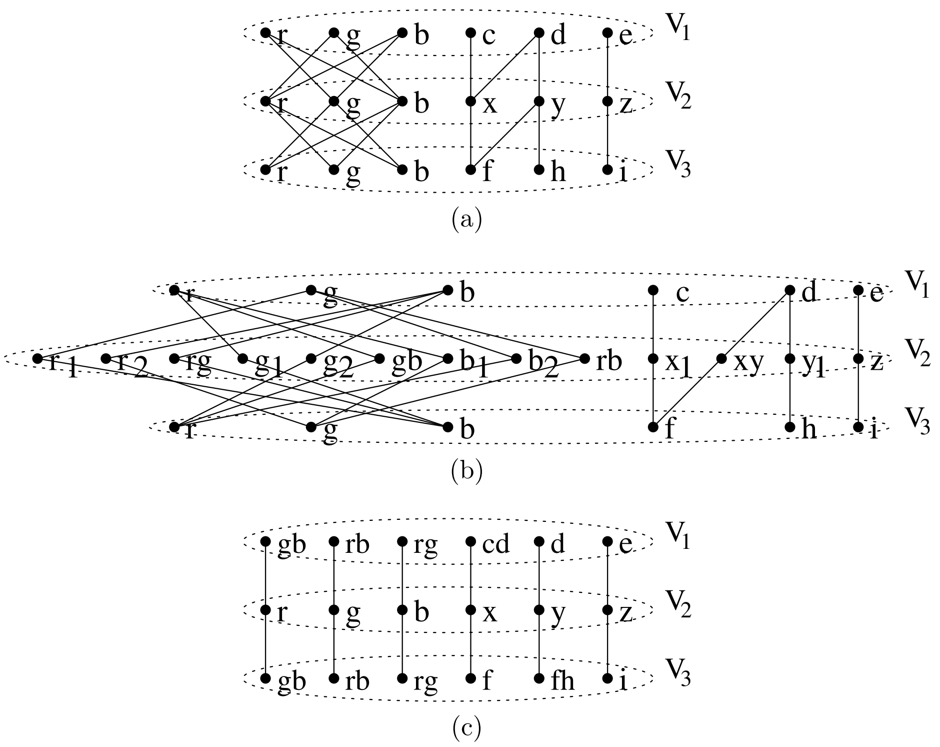


Figure 2.3: An example of the transmutation process.

Definition 5 (Context Dependent Interchangeability) *Values a and b in D_V are* context dependent interchangeable *iff there exist two solutions π_1 and π_2 such that $\pi_1(V) = a$ and $\pi_2 = \pi_1[a/b]$.*

Theorem 2 (Intersection of CDI Values) *Values a and b in D_V are CDI for a CSP $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ iff in* trans$(\mathcal{P}, V) = (\mathcal{V}, \mathcal{D}', \mathcal{C}')$ *there is* exactly one *$c \in \mathcal{D}'(V)$ such that $c \sqsubseteq a$ and $c \sqsubseteq b$ and there exists a solution π such that $\pi(V) = c$.*

Proof: ($\Rightarrow$) From Theorem 1(2) we have c_1 and c_2 and two solutions $\pi[a/c_1]$ and $\pi[a/c_2]$ ($= \pi[a/b][b/c_2]$) for *trans*$(\mathcal{P}, V)$. But c_1 and c_2 must be the same value because otherwise $c_1 \odot c_2 \neq \emptyset$ ($\pi(W) \in c_1(W) \cap c_2(W)$ for all $W \in N_V$), contradicting Definition 4(1).

($\Leftarrow$) Given a solution π in *trans*$(\mathcal{P}, V)$ such that $\pi(V) = c$ where $c \sqsubseteq a$ and $c \sqsubseteq b$, then $\pi[c/a]$ and $\pi[c/b]$ are solutions of $\mathcal{P}$ according to Theorem 1(1). □

Collorary 1 (Necessary Condition for CDI) *If values a and b in D_V are CDI then $a \odot b \neq \emptyset$.*

Theorem 2 tells us that two values are CDI iff both share the same sub-value and that sub-value is part of a solution. As a result, CDI can be seen as a local condition followed by a global condition. In [WFC96], however, CDI is viewed as a single global condition and has to be detected indirectly by a non-CSP method.

We can exploit this property to find CDI values by transmutating a domain and then solving the resulting problem *independently using any CSP algorithm.* E.g., to check whether a and b in Figure 2.2a are CDI, one only needs to check whether ab in 2.2c is involved in any solution. If two values do not intersect, such as h and i in 2.2a, we can tell immediately that they are not CDI according to Corollary 1.

We next consider the relation between full interchangeability (FI) [Fre91] and transmutation. [Fre91] showed that NI implies FI. But what happens when values are FI but not NI? Theorem 3 gives the answer.

Definition 6 (Full Interchangeability) *Two values a and b in D_V are FI iff for any solution π,*

(1) if $\pi(V) = a$ then $\pi[a/b]$ is a solution.

(2) if $\pi(V) = b$ then $\pi[b/a]$ is a solution.

In contrast to FI, CDI has a weaker requirement in that values do not need to be interchangeable in all solutions — just some of them. On the other hand, FI values do not need to be involved in any solution, whereas it is necessary for CDI.

Theorem 3 (FI Pruning) *Two values a and b in D_V are FI iff for any solution π, $(\pi(V) \in \{a, b\}) \Rightarrow (\pi(W) \in \sigma_{a\odot b}(W)$ for all $W \in N_V)$.*

Sketch of Proof: Contrapositive of the fact: there exists a solution π such that $\pi(V) = a$ but $\pi[a/b]$ is not a solution iff there exists $W \in N_V$ such that $\pi(W) \notin \sigma_{a\odot b}(W)$. □

Collorary 2 (Necessary Condition for FI) *If $a \odot b = \emptyset$ then a and b are FI iff neither takes part in a solution.*

The common way of finding FI values is to look for two values that can be interchangeable in the solution sets. We will show how to use Theorem 3 to reduce the search effort.

Consider the CSP in 2.3a; suppose that there is an extra constraint between V_1 and V_3, whose only nogoods are $\{(c,f),(d,h)\}$. Suppose we are interested in finding FI values in V_2, whose domain is $\{r,g,b,x,y,z\}$, and after transmutation becomes what is shown in 2.3b. The potential FI pairs of values are r-g, g-b,

r-b, and x-y. Although any pair of values that do not intersect can still trivially be FI, this is uninteresting since according to Corollary 2 both of them must not take part in any solution; thus interchangeability of z can be ruled out because it does not intersect with other values.

We can remove rg, gb, and rb from the domain since their involvement in solutions does not affect FI. The domain is now $\{r_1,r_2,g_1,g_2,b_1,b_2,x_1,y_1\}$.

To find all the solutions, suppose we try to instantiate V_2 in the order $(r_1,r_2,g_1,g_2,b_1,b_2,x_1,y_1)$. The first solution involves r_1. According to Theorem 3, r cannot be FI with any other values and so we can rule out r-g and r-b. Moreover, r_2 can be removed from the domain since its involvement in a solution does not give us any new information.

Next we try g_1, and it too takes part in a solution. By the same reasoning, we infer that g cannot be FI with any other values, and so we can prune g_2. We are now done with FI regarding $\{r,g,b\}$ — no FI value involving one of them exists — and so values b_1 and b_2 can be removed.

We carry on with values x_1 and y_1 that are left in the domain; neither is involved in a solution. There is no value left and by Theorem 3, we conclude that x and y are FI.

2.4 Domain Transmutation Procedure

The procedure for transmutating the domain of a single variable is stated as pseudo-code below.

Theorem 4 (Correctness) *Algorithm* `transmutate` *produces a domain transmutation.*

Proof: At the end of the algorithm `output` is a disjoint set of values by induction on the size of `output` and Lemma 1(2). Since the algorithm depends only on $\odot$ and $\ominus$ to split values, all solutions are preserved by Lemma 1(2). Therefore `output` satisfies Definition 4 $\square$

The following theorem shows that each variable needs to be transmutated by the algorithm at most once.

Theorem 5 (Transmutation Quiescence) *If a variable domain D_V is a transmutation, any subsequent transmutation attempt using the algorithm will*

Algorithm 3: transmutate(V)

```
1  input ← $D_V$
2  output ← $\emptyset$
3  while input $\neq \emptyset$ do
4      Select and delete any $a$ from input
5      match ← false
6      while (not match) and (next $b$ in output exists ) do
7          if $a \odot b \neq \emptyset$ then
8              match ← true
9              $\mathscr{A} \leftarrow a \ominus b$
10             $\mathscr{B} \leftarrow b \ominus a$
11             Replace $b$ in output with $a \odot b$ and all values in $\mathscr{B}$
12             Put all values in $\mathscr{A}$ back to input
13     if (not match) then
14         Insert $a$ to output
15 Where possible, union values in output
16 $D_V$ ← output
17 Update all constraints connecting $V$
```

result in no change.

Proof: This is obviously true if no variable domain in N_V is altered. Otherwise, we will show that after D_V has been transmutated, it remains disjoint regardless of any domain transmutation in N_V.

Let $\mathcal{P}$ be the CSP ($\mathcal{V}$,$\mathcal{D}$,$\mathcal{C}$) just after D_V has been transmutated. Let a and b be values in $\mathcal{D}(V)$. Both are disjoint according to Definition 4(1) and so there must be a variable $W \in N_V$ such that $\sigma_a(W) \cap \sigma_b(W) = \emptyset$. Consider *trans(*$\mathcal{P}, W$*)* $=$ $(\mathcal{V}, \mathcal{D}', \mathcal{C}')$ and let us suppose that $c \in \sigma_a(W) \cap \sigma_b(W) \neq \emptyset$ as a result of transmutating W, for some value $c \in \mathcal{D}'(W)$.

By Lemma 1(1), $\{a, b\} \subseteq \sigma_c(V)$. Since $c \in \mathcal{D}'(W)$, by Theorem 1(1) $c \sqsubseteq d$ for some $d \in \mathcal{D}(W)$. That means $\{a, b\} \subseteq \sigma_c(V) \subseteq \sigma_d(V)$. By Lemma 1(1), $d \in \sigma_a(W) \cap \sigma_b(W)$ — contradicting $\sigma_a(W) \cap \sigma_b(W) = \emptyset$ for $\mathcal{P}$.

Hence the assumption is false and $\sigma_a(W) \cap \sigma_b(W)$ remains empty after W is transmutated. □

Note that union in line 15 is used in re-combining remaining fragmented values from the same source, although the smallest possible domain transmutation is not required by Definition 4.

During transmutation the size of `output` could be very large if there many fragments of values left after intersection. If so, it is very unlikely that the values inside can be combined so that the resulting domain is smaller than the original. Therefore we use the *transmutation cutoff heuristic* which imposes an upper bound on the size of `output`; if it exceeds the upper bound, transmutation is aborted.

The overall transmutation process involves transmutating each variable domain, one by one, in some order. The order in which variables are processed affects the result. Figure 2.3 shows sample transmutations: 2.3a is the original, 2.3b shows the result of processing in the order (V_2, V_1, V_3), while the CSP in 2.3c is obtained with either (V_1, V_3, V_2) or (V_3, V_1, V_2). (In fact, the CSP in 2.3b results from the transmutation of V_2 alone and subsequent attempts to transmutate V_1 and V_3 result in no change.)

The CSP in 2.3b has larger domains than the original CSP in 2.3a, so it can be expected that it would take more time to solve. As a heuristic, we have chosen never to accept a domain transmutation if domain size is increased; we call this the *transmutation acceptance heuristic*. While variable domains can, in principle, be transmutated in any order, we always pick the variable whose transmutation leads to the maximum domain size reduction. To achieve this, we tentatively consider the transmutation of each variable in the network; if there is no variable whose domain size is reduced by the transmutation procedure, we terminate the process. The same approach is applied for subsequent variable transmutations. We need only recompute tentative transmutations for the domains of the variables that are immediate neighbors of the variable whose domain has been transmutated.

2.5 Experimental Results

We tested the `transmutate` algorithm and solved the transmutated problems for all solutions over randomly generated CSPs with ten variables[6] and six values in each domain, varying density from 0.2 to 0.8 with 0.05 increment step, while tightness ranges from 0.2 to 0.8 with the same increment. We used model B random problem generator, with “flawless” constraint generation [GMP+01].

[6]These problems may appear small but, as will be discussed later, many real-world uses of the approach involve only small regions of larger network.

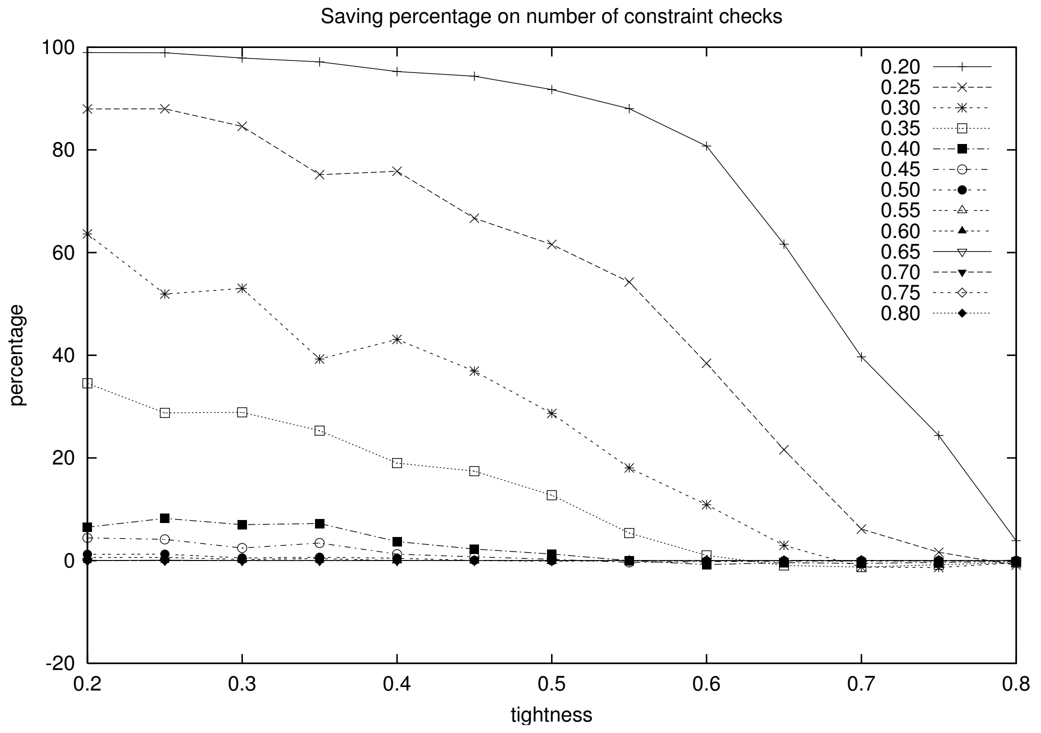


Figure 2.4: Experimental results: saving percentage on the number of constraint checks. The legends refer to density.

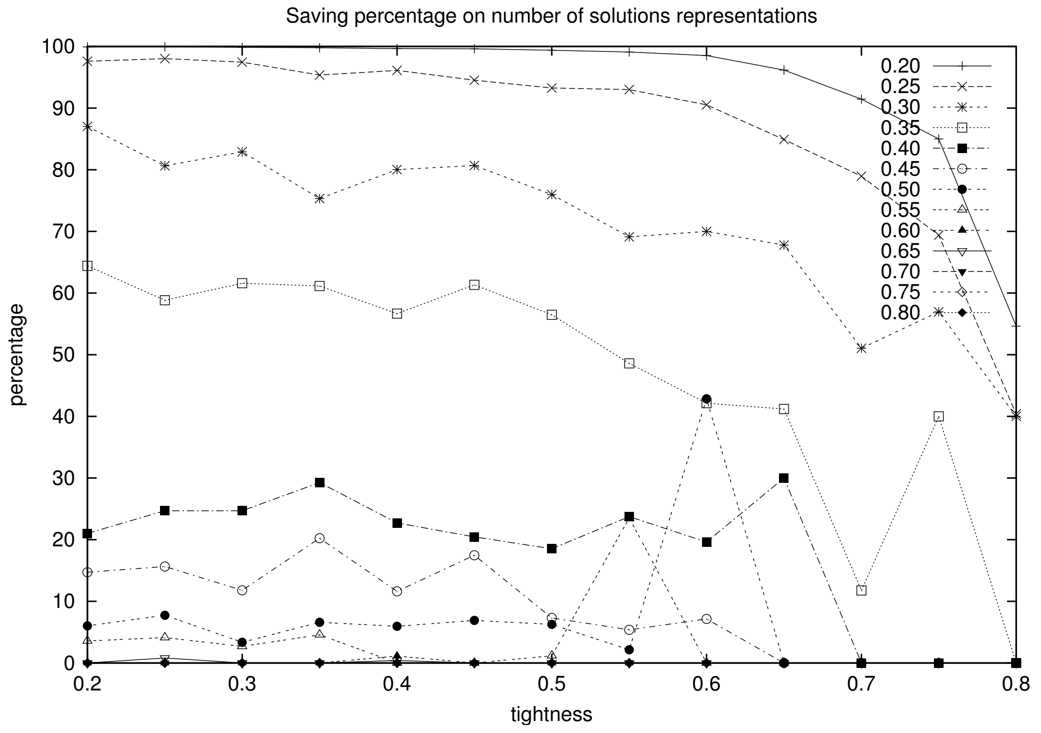


Figure 2.5: Experimental results: saving percentage on the number of solution representations. The legends refer to density.

For each data-point, the number of constraint checks required to compute all solutions and the number of representations needed to express these solutions are computed and averaged over 100 problem instances. We gathered the data for both transmutated and original problems. The solver is MAC-3 with dom/deg variable ordering. For the transmutation cutoff heuristic, we fixed the upper bound of the `output` queue to be ten times the size of the domain (60).

It may appear that our experimental problems are too small, but remember that *all* solutions are to be computed. Since the solving time increases exponentially with the problem size, finding all solutions in larger networks therefore has few practical uses, and is not studied here. Indeed, as explained in the introduction, all solutions are needed only for relatively small subnetworks in larger CSPs — for creating meta-variables or for user perspicuity of subnetworks. Furthermore, since we tested the algorithm over a wide range of density and tightness, we had to settle on a relatively small problem size to keep the experimental time in check. However we believe the trend seen in the experiments in this chapter can be extrapolated to larger problems without significant loss of accuracy.

Because transmutation time is always an insignificant fraction of the time needed to compute all the solutions of a CSP, it is always worthwhile trying the transmutation. In our setting, it takes just a few seconds to transmutate a (50, 10, 0.5, 0.5)[7] instance.

The results are shown in the graphs, in which the legends refer to density. While the transmutation cannot *guarantee* to reduce the number of constraint checks needed to compute all solutions to a CSP, it can produce huge savings. From the graphs, we see that more domain transmutation occurred for problems with lower density or lower tightness, as either case will lead to a lot of value combination; a similar reduction holds for the solution representation. On the other hand, in the worst case we have seen in the experimental data, which occurred only at a few data-points, the number of extra constraint checks for a transmutated CSP is no more than one or two percent of the untransmutated CSP.

[7]A tuple (n, m, *density*, *tightness*) denotes a CSP where n is the number of variables and m is the domain size for each variable.

2.6 Other Applications

Domain transmutation can be used for other purposes as well. Three applications are described in this section.

2.6.1 Robustness

We can exploit the combination of values after transmutation to achieve robustness for a single solution simply by ordering values during search based on the size of their labels. For example, suppose (*a*, *abc*, *b*, *bc*, *c*) is the transmutated domain of some variable. During search, it would be ordered as (*abc*, *bc*, *a*, *b*, *c*). The first solution obtained will be the most robust in this variable. That is, if a solution found involves *abc*, we know that it corresponds to three solutions in the original problem and should one of the three values be unavailable, it can be immediately replaced with either of the other two without any change to the rest of the solution. The advantage of this approach is that the transmutation process is independent of the solving process and thus no specialized solver is required. We can also order the variables for search based on which variables should be more robust to changes.

But, since variables are instantiated in order, there may be a case where the solver would prefer a solution which is very robust at one variable and brittle at another, rather than a uniformly robust solution: e.g., a solution tuple (*abcd*, *efg*, *hi*, *j*) may be found before (*ac*, *ef*, *hi*, *jk*). [GPR98] provides formalization and more details on solution robustness. In contrast, [Les94] propose an algorithm that gives a single maximal bundling of solutions.

2.6.2 Decision Problems

In Definition 3, union is restricted to fragments that come from the same value so as to preserve the solutions. However, satisfiability remains unchanged when the restriction is lifted, although a solution cannot easily be converted back.

Definition 7 (Extended Union) *Let a and b be two values in D_V. The* extended union *of a and b (denoted by $a \otimes b$) is a value c where $L_c = L_a \cup L_b$ and $\sigma_c(W) = \sigma_a(W) \cup \sigma_b(W)$ for all $W \in N_V$. The extended union is undefined (denoted by $a \otimes b = \emptyset$) if there exist two or more variables $X \in N_V$ such that $\sigma_a(X) \neq \sigma_b(X)$.*

Theorem 6 (Satisfaction of Extended Union) *Consider two values a and b in CSP $\mathcal{P}$ such that $a \otimes b \neq \emptyset$, and the resulting CSP $\mathcal{P}'$ where a and b are replaced by $a \otimes b$. $\mathcal{P}$ is satisfiable iff $\mathcal{P}'$ is satisfiable.*

For example, consider $(\{\boldsymbol{a}\}, \{x\}\times\{y\})$ and $(\{\boldsymbol{b}\}, \{x\}\times\{z\})$ in D_V, where y and z are in D_W, $W \in N_V$. Both can be combined into $(\{\boldsymbol{a},\boldsymbol{b}\}, \{x\}\times\{y,z\})$. Given a solution π involving $(\{\boldsymbol{a},\boldsymbol{b}\}, \{x\}\times\{y,z\})$, there must be another solution involving either $(\{\boldsymbol{a}\}, \{x\}\times\{y\})$ or $(\{\boldsymbol{b}\}, \{x\}\times\{z\})$, although we cannot tell which one without inspecting $\pi(W)$.

In a situation where satisfiability of problems is the only concern, extended union can be used on its own to prune values during preprocessing or during search, similar to NI. And, unlike basic domain transmutation, the resulting domain is *guaranteed* to be smaller than the original.

Using extended union in domain transmutation can sometimes dramatically reduce the domain size. For example, the domain of V_2 in Figure 2.3b can be reduced from 13 values to 4 should we use extended union in the algorithm.

2.6.3 Finding a Single Solution

Even though Proposition 1 tells us that the upper bound on the search space for a transmutated CSP is smaller than its original, this is not in any way an indication of the actual search effort. In fact, in our preliminary, unreported experiments, the results are mixed. This is simply a consequence of conventional CSP solvers, where each value's support is established as the search progresses, and discarded afterward during backtracking. As the problem gets harder (more backtracking) and the domain size gets larger[8], finding a support for each value becomes the main bottleneck as it has to be re-discovered anew.

In this situation, the benefit of AC-4-like data structures [MH86] in which supports for each values are remembered could offset its overhead. Although AC-4 was empirically shown to be inferior than AC-3 — a forgetful algorithm with non-optimal worst-case time-complexity — as far as we know there is no report on its effectiveness compared to other AC algorithms when used during search to solve *hard* problems. ([SF94b] compared MAC-4 with only forward checking.)

[8]Larger domains in transmutated CSP are not uncommon, despite using the best variable ordering.

2.7 Related Works

Cross product representation (CPR) of partial solutions generated during search has been studied in [HF92]. In extending a CPR to complete solutions, a new value is combined into existing CPR if possible; otherwise a new CPR is created. Domain transmutation could be seen as a "dynamic programming" version of the same concept.

Preliminary work on value transformation was reported in [BL03]. Subsequently and independently [CS03] defined Generalized Neighborhood Substitutability, which corresponds exactly to the intersection of values. Rather than extracting identical parts from various values to form new ones, extra constraints are inserted to prevent the same search space from being explored again. However, this approach is undemonstrated, and no other benefits such as compact representation of solutions can be obtained.

2.8 Conclusions

We present a new perspective on domain values in CSP where each value is viewed as a combination of smaller values. This allows us to freely manipulate variable domains so as to remove redundant partial assignments. We present an inexpensive method for merging duplicate local solutions in a constraint network, which subsumes NI and NS. The benefits are two-fold: reducing time spent on finding consistent assignments and reducing the complexity of enumerating the members of the complete solution set. Experimental results show that the savings are considerable, especially on loose problems with a low density of constraints. We use the new extended CSP definition, in which values are composed of labels and structures, to obtain new result on CDI and FI and to prove some properties of the algorithm presented. We also suggest how it could be used to find robust solutions, to solve decision problems, and to speed up the search for a single solution.

Chapter 3

Extracting Microstructure in Binary Constraint Networks

3.1 Abstract

We present algorithms that perform the extraction of partial assignments from binary Constraint Satisfaction Problems without introducing new constraints. They are based on a new perspective on domain values: we view a value not as a single, indivisible unit, but as a combination of value fragments. Applications include removing nogoods while maintaining constraint arity, learning nogoods in the constraint network, enforcing on neighborhood inverse consistency and removal of unsolvable sub-problems from the constraint network.

3.2 Introduction

Constraint Satisfaction Problems (CSP) is one of the most important modeling tools in AI with far-reaching applications. There are many variations on the standard CSP, such as weighted constraints, partial, or distributed CSP, but they are all based on the same notion: assigning values to variables in order to satisfy constraints among them.

Normally, values that can be assigned to a variable are treated as being inherently different from each other. In [BL04] however, a new viewpoint on domain values is proposed which takes into account the microstructure (a low-level graphical representation of a CSP based on compatibility of values).

Basically, a value is composed of *label* and *support structure* and can be broken into several values as long as they correspond to the original one. Solutions involving values with the same label are indistinguishable from one another given the rest of the solutions are the same.

This definition of a domain value has been shown effective in reducing interchangeability within a variable domain by recombining value fragments to eliminate identical parts [BL04]. This chapter also uses the microstructure but differs in that we only break values apart and do not form new values from the value fragments. Furthermore, it deals with a partition of the microstructure that spans many connected variables, as opposed to a single variable in [BL04].

We illustrate the main idea and one of its applications in Figure 3.1(a). There are three variables and eight values, and each line denotes a compatible pair of values. Suppose we want to designate the tuple (b, d, g) as a nogood (a partial assignment that cannot be extended to a full solution) the usual approach would be to add a new 3-ary constraint involving the three variables. It is unsafe to directly delete one of b, d, or g since solutions involving them may be inadvertently eliminated in the process.

Figure 3.1(b) depicts an equivalent CSP in which values are split into fragments as shown. Values having the same label (e.g. the three values with label "b") are differentiated by their structures. Solutions of (a) are the same as those of (b) with respect to their support labels. The tuple (b, d, g) is isolated (by the process we will later describe) and thus we can simply delete one of the values involved to mark it as a nogood; the rest would be removed by simple arc consistency processing. Should this network be part of a larger one, the links to the external part would need to be modified so that they connect to the newly-created values instead (e.g a single link to b in (a) corresponds to three links to the three b in (b)).

Our contribution includes a novel process of extracting partial assignments without introducing new constraints, along with the following applications:

- *Maintaining constraint arity.* Because no new constraint is introduced, the original constraint arity would remain unchanged. Algorithms that work only for binary CSPs [BCvBW02] for instance would be usable even after higher-order nogoods are incorporated.

- *Learning nogoods by pruning.* Nogoods can be extracted and pruned from the microstructure, as shown in Figure 3.1(b), rather than recorded

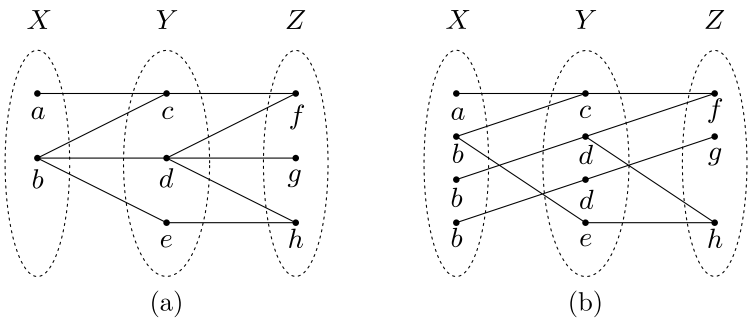


Figure 3.1: Marking (b, d, g) in (a) as a nogood by eliminating one of b, d, or g, could remove some solutions, while doing so in (b), an equivalent network, is completely safe.

[FD94]. This gives an alternative approach to deal with the problem of increasing memory requirements for recording nogoods.

- *Enforcing Neighborhood Inverse Consistency in one pass.* Like many other consistency algorithms, NIC [FE96] requires more than a single pass of propagation to fully achieve consistency. We show how to do it in a single pass.
- *Extracting unsolvable sub-problems.* The common approach taken is to decompose a CSP into several sub-problems [FH95]. Our method is more efficient since only the target sub-problem is extracted while everything else remains intact in the original CSP.

The chapter is organized as follows. In Section 3.3 we cover the CSP preliminary and recall the formalism on domain values. In Section 3.4 we provide two algorithms, one for extracting simple nogoods and the other for extracting complex ones, along with their proofs of correctness. In Section 3.5, the process is generalized so that more complex microstructures can be separated. Applications are described in Section 3.6. We conclude in Section 3.7.

3.3 Background

Definition 8 (Binary Constraint Network) *A* binary constraint network $\mathcal{P}$ *is a triplet* $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ *where* $\mathcal{V}$ *is a finite set of variables,* $\mathcal{D} = \bigcup_{V \in \mathcal{V}} D_V$ *where* D_V *is a finite set of possible values for* V*, and* $\mathcal{C}$ *is a finite set of constraints such that each* $C_{XY} \in \mathcal{C}$ *is a subset of* $D_X \times D_Y$ *indicating the possible pairs of values for* X *and* Y*, where* $X \neq Y$*. Without loss of generality, we assume that* *two values in different domains are distinct (i.e. if* $a \in D_X$ *then* $a \notin D_Y$ *for all* $Y \neq X$*). If* $a \in D_X$ *then we use* var(a) *to denote* X*. If* $C_{XY} \in \mathcal{C}$*, then* $C_{YX} = \{(y, x) \mid (x, y) \in C_{XY}\}$ *is also in* $\mathcal{C}$*.*

A binary constraint network is k-consistent *if given* $k-1$ *variables and* $k-1$ *values that satisfy the constraints on these variables, we are able to find a value (called a* support*) for any* k*th variable such that all the constraints on the* k *variables will be satisfied by the* k *values taken together. When* $k = 2$ *and 3 it is called Arc Consistency (AC) and Path Consistency (PC) respectively.*

The neighborhood *of variable* V *(*N_V*) is the set* $\{W \mid C_{VW} \in \mathcal{C}\}$*. We use* n*,* e*, and* d *to denote the number of variables, the number of constraints, and the maximum domain size. An* assignment *is a function* $\pi : \mathcal{W} \subseteq \mathcal{V} \to \mathcal{D}$ *such that* $\pi(W) \in D_W$ *for all* $W \in \mathcal{W}$*; we denote the function domain by* dom(π)*.* π *is* consistent *if and only if for all* $C_{XY} \in \mathcal{C}$ *such that* $X, Y \in$ dom(π)*,* $(\pi(X), \pi(Y)) \in C_{XY}$*.* π *is a* solution *if and only if* dom$(\pi) = \mathcal{V}$ *and* π *is consistent. We use* $\mathcal{P}\mid_{\mathcal{W}}$ *to denote the binary constraint network induced by* $\mathcal{W} \subseteq \mathcal{V}$*. A* Constraint Satisfaction Problem *involves finding one or more solutions to an associated constraint network or declaring it unsatisfiable.*

We extend the usual definition of a value to a 2-tuple in order to clearly distinguish between the syntax (value of label) and the semantics (the support corresponding to the label).

Definition 9 (Values) *A* value $a \in D_V$ *is a 2-tuple* (L, σ) *where* L *is a set of* labels*, while* σ*, called* support structure*, is a function* $\sigma : N_V \to 2^{\mathcal{D}}$ *such that* $\sigma(W) \subseteq D_W$ *for any* $W \in N_V$*. We use* L_a *to denote the set of labels of* a *and use* σ_a *to denote the support structure of* a*. A value* a *must be* valid*, that is,* $\sigma_a(W) = \{b \in D_W \mid (a, b) \in C_{VW}\}$ *for any* $W \in N_V$*. The* local solutions *of* a *is the set* $\{(s_1, \ldots, s_{|N_V|}) \in D_{W_1} \times D_{W_2} \times \ldots \times D_{W_{|N_V|}} | s_i \in \sigma_a(W_i), W_i \in N_V\}$*, the enumeration of* a*'s supports. The* size *of* a *(*size*(*a*)) is* $\prod_{W \in N_V} |\sigma_a(W)|$*, which equals the number of local solutions of* a*.*

Let $\mathcal{L} = \bigcup_{a \in \mathcal{D}} L_a$ *be the set of all labels. A* label-assignment *is a function* $\lambda : \mathcal{W} \subseteq \mathcal{V} \to \mathcal{L}$ *such that* $\lambda(W) \in \bigcup_{a \in D_W} L_a$ *for all* $W \in \mathcal{W}$*. Given an assignment* π*, we denote* π_{label} *to be the set of label-assignments* $\{\lambda \mid \lambda(V) \in L_{\pi(V)}\}$*.*

The new definition allows a domain to contain values having the same label but different support structures; values having both the same label and the support structure are not permitted. In some example, we append a suffix to labels.

This is not actually necessary and is used only to better differentiate between values with the same label (e.g. c_1, c_2 and c_3 in Figure 3.2(b) all have the same label c). For simplicity, we use a form of cross-product representation to denote the support structure of a value. For example, value d in Figure 3.1(a) is represented by ($\{\boldsymbol{d}\}$, $\{b\}\times\{f,g,h\}$). The new definition also allows multiple labels so that we can combine neighborhood interchangeable values [Fre91] without losing any solution (unlike the standard practice where only one value is kept). For example ($\{\boldsymbol{a}\}$, $\{x\}\times\{y\}$) and ($\{\boldsymbol{b}\}$, $\{x\}\times\{y\}$) can be replaced by a single value ($\{\boldsymbol{a,b}\}$, $\{x\}\times\{y\}$). Any solution involving ($\{\boldsymbol{a,b}\}$, $\{x\}\times\{y\}$) can be converted back using label-assignment.

We define the following operations on values, analogous to the usual operations on sets. For these operations to be correct, we require constraint networks to be AC. Given the same requirement for various algorithms — for instance Maintaining Arc Consistency — and the efficiency of recent AC algorithms this property is not taxing to presume.

Definition 10 (Operations on Values) *Let a and b be two values in D_V.*

The intersection *of a and b ($a \odot b$) is a value c where $L_c = L_a \cup L_b$ and $\sigma_c(W) = \sigma_a(W) \cap \sigma_b(W)$ for all $W \in N_V$. Two values a and b are* disjoint *($a \odot b = \emptyset$) if there exists a variable $X \in N_V$ such that $\sigma_a(X) \cap \sigma_b(X) = \emptyset$. A set of values is disjoint if its members are pairwise disjoint.*

The union *of a and b ($a \oplus b$) is a value c where $L_c = L_a = L_b$ and $\sigma_c(W) = \sigma_a(W) \cup \sigma_b(W)$ for all $W \in N_V$. Union is undefined*[1] *($a \oplus b = \emptyset$) if $L_a \neq L_b$ or there exist $X, Y \in N_V$ such that $\sigma_a(X) \neq \sigma_b(X)$ and $\sigma_a(Y) \neq \sigma_b(Y)$. A* subtraction *of b from a ($a \ominus b$) is a minimal set of disjoint values $\mathscr{C}$ such that $\bigoplus(\mathscr{C} \cup \{a \odot b\}) = a$. Value a is* subsumed *by b ($a \sqsubseteq b$) if $\sigma_a(W) \subseteq \sigma_b(W)$ for all $W \in N_V$.*

For example, ($\{\boldsymbol{x,y}\}$, $\{a,b\}\times\{c\}$) $\odot$ ($\{\boldsymbol{y,z}\}$, $\{a\}\times\{c,d\}$) = ($\{\boldsymbol{x,y,z}\}$, $\{a\}\times\{c\}$), ($\{\boldsymbol{x}\}$, $\{a\}\times\{b\}\times\{c\}$) $\oplus$ ($\{\boldsymbol{x}\}$, $\{a\}\times\{b\}\times\{d\}$) = ($\{\boldsymbol{x}\}$, $\{a\}\times\{b\}\times\{c,d\}$). For subtraction, we stress that the result is a *set* of values. Consider ($\{\boldsymbol{a}\}$, $\{d,e\}\times\{f,g\}$) $\ominus$ ($\{\boldsymbol{b}\}$, $\{d\}\times\{f\}$): two possible results[2] are $\{$($\{\boldsymbol{a}\}$, $\{e\}\times\{f,g\}$), ($\{\boldsymbol{a}\}$, $\{d\}\times\{g\}$)$\}$ and $\{$($\{\boldsymbol{a}\}$, $\{e\}\times\{f\}$), ($\{\boldsymbol{a}\}$, $\{d\}\times\{e,g\}$)$\}$. These operations preserve

[1]In other words, supports are only allowed to differ on one variable in the neighborhood. Union imposes this restriction so that a new value can be formed without introducing spurious solutions. Union was used in [BL04] but not in the algorithms here.

[2]Algorithms in this chapter do not require the result of a subtraction to be unique, although it would lead to different networks. We can enforce uniqueness by imposing some ordering on the support structure.

the local solutions of values involved.

3.4 Extracting Assignments

We define the following definitions in order to deal with the extraction process precisely.

Definition 11 (Unit) *A value $a \in D_X$ is called a* unit value *if and only if a has exactly one support in the domain of each variable in N_X. A consistent assignment $\pi = (t_1, \ldots, t_m)$ is called a* unit assignment *if and only if each t_i is a unit value in $\mathcal{P}|_{dom(\pi)}$.*

Given a consistent assignment π, the goal is to make it a unit assignment. There are many ways to transform the domains involved so that π becomes unit. The simplest method is to solve $\mathcal{P}|_{dom(\pi)}$ for all solutions and modify domains and constraints so that a solution is represented by a single strand in the microstructure. The target unit assignment would be one of the solutions. This process is not practical as all solutions are needed and the sub-problems involved are completely replaced. We propose a dynamic method that gradually changes the network by subtracting from t_i in π the *unit value of t_i with respect to π* (defined below) until the whole assignment becomes unit.

Definition 12 (Unit Value with Respect to Assignment) *Given a consistent assignment $\pi = (t_1, \ldots, t_m)$ and a value $a = t_i$ for some $1 \le i \le m$, the* unit value of a with respect to π *(denoted by* $\mathrm{unit}(a, \pi)$*) is a value u such that $L_u = L_a$ and for any X in the neighborhood of* $\mathrm{var}(a)$

$$\sigma_u(X) = \begin{cases} \{t_j\} & \textit{if } X = \mathrm{var}(t_j), i \neq j \textit{ and } 1 \le j \le m \\ \sigma_a(X) & \textit{otherwise} \end{cases}$$

We explain the process using Figure 3.2(a)–(h). Figure (a) depicts the initial microstructure. Suppose we want to mark the assignment $\pi = (b, c, e, f, h)$ as a nogood. This can be done by separating it from the rest of the network. The result is shown in (h); both (a) and (h) are equivalent in term of solution set.

The entire process could be thought of as "untangling" a thread by splitting it off one segment at a time. In (b) for instance, the value ($\{\boldsymbol{c}\}$, $\{a,b\}\times\{d,e\}$) in (a) is split into three values with the same label: ($\{\boldsymbol{c}\}$, $\{b\}\times\{e\}$), ($\{\boldsymbol{c}\}$, $\{b\}\times\{d\}$), and ($\{\boldsymbol{c}\}$, $\{a\}\times\{d,e\}$), where ($\{\boldsymbol{c}\}$, $\{b\}\times\{e\}$) is *unit*(c, π) for both (a) and (b).

These three values are represented as c_1, c_2, and c_3 in the figures. In this example D_Y is chosen first; the complete order is (Y, W, Z, Y, W, U, X) (the order is chosen at random). Note that a domain value could be transformed more than once, and a different ordering results in a different network, although the target unit assignment is always identical.

3.4.1 Basic algorithm

```
Algorithm 4: extractLine($\pi$)
input: A consistent assignment $\pi = (t_1, \ldots, t_m)$
1 repeat
2     Pick $t_k$ such that $t_k \neq unit(t_k, \pi)$
3     $R \longleftarrow t_k \ominus unit(t_k, \pi)$ /* $R$ is a set */
4     Replace $t_k$ in its respective domain with $unit(t_k, \pi)$ and value(s) in $R$
5     Update constraints involved with variable $var(t_k)$
6 until $t_i = unit(t_i, \pi)$ for all $1 \leq i \leq m$
```

A basic template of the extraction process is given in Algorithm 4. We will first consider a restricted version, which requires that π is non-cyclic (defined below). We call this algorithm *extractLine()*.

Definition 13 (Cyclic Assignment) *A consistent assignment π for a binary constraint network $\mathcal{P}$ is* cyclic *if and only if the constraint network for $\mathcal{P}\mid_{dom(\pi)}$ contains a cycle.*

From the example in Figure 3.2(a)–(h) it is not clear whether the algorithm terminates in general, since a variable domain could be repeatedly transformed. For instance, value e in (a) is unit but loses the property after its neighboring value c is split. We will prove that *extractLine()* is correct and terminates.

Definition 14 (Neighborhood Arc Consistency) *A value is* neighborhood arc consistent *(NAC) if and only if its supports are arc-consistent with one another.*

Lemma 2 *Given a constraint network and a value a, an upper-bound on the number of NAC values having the same label as a in an equivalent constraint network is d^n.*

Proof: The maximum number of solutions for the network is d^n. The network can be rearranged so that each solution is a unit assignment. Each value a in the original network participates in no more than d^n of them. Since a unit value

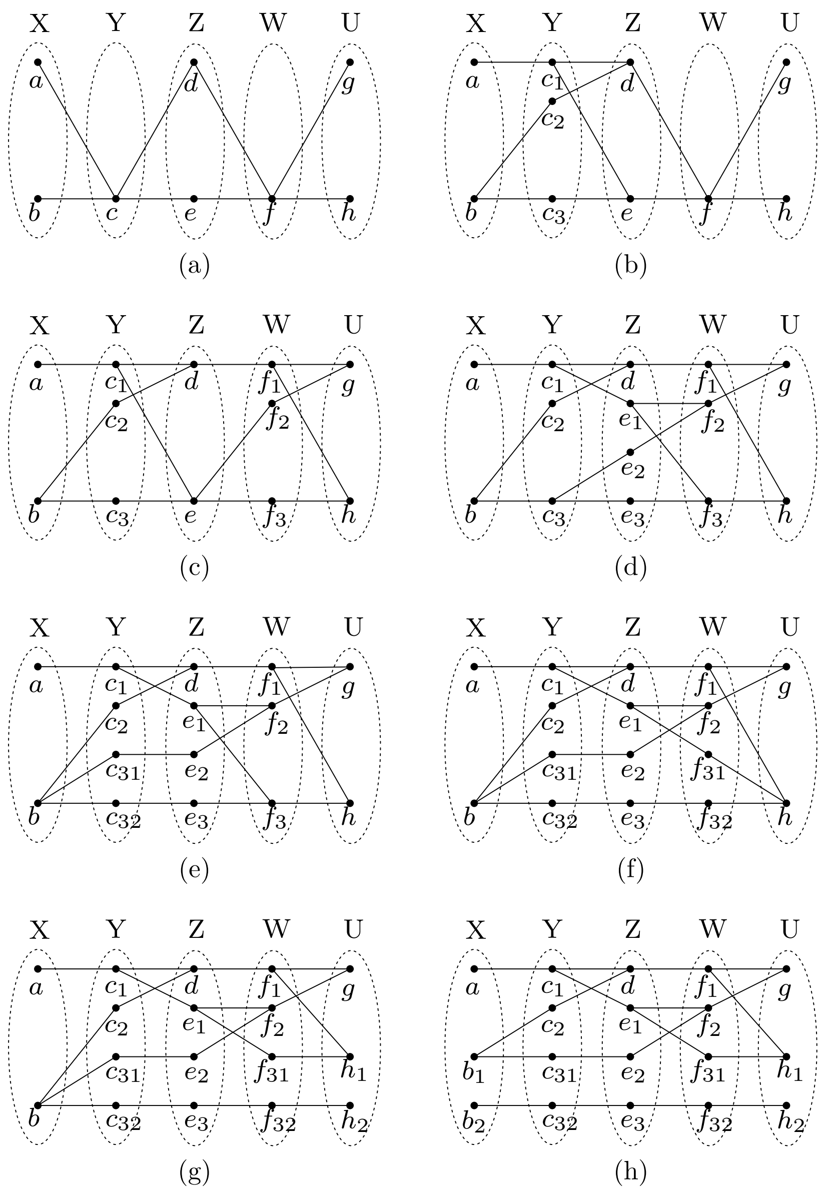


Figure 3.2: Extracting a non-cyclic assignment. Figures (a)–(h) illustrate the process of extracting (b, c, e, f, h) using the order (Y, W, Z, Y, W, U, X). In short, given $a \ominus b$ a subtraction algorithm will try to shed part of a by going through each variable in the neighborhood until b emerges. For instance, consider ($\{\boldsymbol{c}\}$, $\{a,b\}\times\{d,e\}$) $\ominus$ ($\{\boldsymbol{c}\}$, $\{b\}\times\{e\}$) (from (a) to (b)). Since $c \in D_Y$, we need to consider $N_Y = \{X, Z\}$. Suppose X is chosen first; the algorithm would divide ($\{\boldsymbol{c}\}$, $\{a,b\}\times\{d,e\}$) into ($\{\boldsymbol{c}\}$, $\{b\}\times\{d,e\}$) and ($\{\boldsymbol{c}\}$, $\{a\}\times\{d,e\}$). ($\{\boldsymbol{c}\}$, $\{b\}\times\{d,e\}$) is further divided into ($\{\boldsymbol{c}\}$, $\{b\}\times\{d\}$) and ($\{\boldsymbol{c}\}$, $\{b\}\times\{e\}$). Therefore,($\{\boldsymbol{c}\}$, $\{a,b\}\times\{d,e\}$) $\ominus$ ($\{\boldsymbol{c}\}$, $\{b\}\times\{e\}$) = $\{$($\{\boldsymbol{c}\}$, $\{a\}\times\{d,e\}$), ($\{\boldsymbol{c}\}$, $\{b\}\times\{d\}$), ($\{\boldsymbol{c}\}$, $\{b\}\times\{e\}$)$\}$ using the order (X, Z) (this ordering has nothing to do with the selection of t_k from line 2 of Algorithm 4). Note that ($\{\boldsymbol{c}\}$, $\{a\}\times\{d,e\}$) $\oplus$ ((($\{\boldsymbol{c}\}$, $\{b\}\times\{d\}$) $\oplus$ ($\{\boldsymbol{c}\}$, $\{b\}\times\{e\}$)) = ($\{\boldsymbol{c}\}$, $\{a,b\}\times\{d,e\}$).

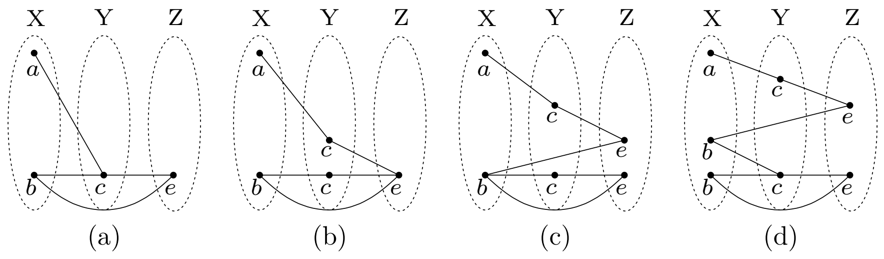


Figure 3.3: Figure (a) – (d) illustrate non-terminating transformation. Figure (a) depicts the partial microstructure induced by (b, c, e). In Figure (b) – (d), the processing order is (Y, Z, X). Extraction continues indefinitely for the order $(Y, Z, X, Y, Z, X, \ldots)$.

of a unit assignment is NAC, an equivalent network can contain at most d^n NAC unit values having the same label as a. □

Theorem 7 extractLine() *is correct and terminates on non-cycle input.*

Proof: The algorithm changes the constraint network incrementally by splitting each t_k one by one. Since operations on values, including subtraction ($\ominus$), preserve the local solutions with respect to their labels, the resulting network admits the same solutions as the original.

After the subtraction in line 3, $unit(t_k, \pi)$ takes place of t_k in the repeat loop, making t_k a unit value in the future passes. When the condition in line 6 becomes true, each component of π must be a unit value and thus π is a unit assignment according to Definition 11.

To show that the algorithm terminates, we note that there are at most $d^{|dom(\pi)|}$ NAC values having the same label as t_k according to Lemma 2. Since $t_k \neq unit(t_k, \pi)$ (in fact t_k subsumes $unit(t_k, \pi)$), each subtraction produces at least one new value r in R with the same label as t_k. The acyclic restriction on π implies there is no constraint among variables in the neighborhood of $var(t_k)$, thus r is automatically NAC. Therefore, each subtraction produces at least one NAC value. As the upper-bound on the number of NAC values having the same label as t_k and the number of components of π (which is m) are finite, the algorithm cannot keep on producing new NAC values and it must terminate within a finite number of steps. □

Space complexity of *extractLine()* is $O(dg)$ where g is the maximum degree of all variables in the network. This is due to the fact that the algorithm works on two values at a time (a value chosen and its unit value) and modifies only part of connecting constraint.

The algorithm may not terminate on input containing a cycle. An example of non-terminating transformation is shown in Figure 3.3(a)—(d).

3.4.2 Extracting cyclic assignments

In a non-terminating transformation that involves a cyclic assignment, values that are repeatedly split off are not part of any solution. This stems from the fact that the definition of unit value does not take into account the constraints among the neighborhood. Since *extractLine()* operates on one segment at a time, intuitively speaking there is a chance that the segments split off will not be joined properly when the target assignment forms a cycle.

A solution to this problem is to enforce PC [Mon74] on new values created after each subtraction. PC ensures that the segments that are split off form a connected path along the cycle. To make the algorithm terminate on cyclic input, we add the following line in the algorithm between line 5 and 6: *Enforce PC on* $\mathcal{P}\mid_{dom(\pi)}$ *only on arcs involving values in R.* Figure 3.4 demonstrates the extraction of a cycle. We call this algorithm *extractCycle().*

It is worth noting that having a constraint network that is already path consistent beforehand does not eliminate the need for path consistency processing in *extractCycle().* As an example, consider ($\{\boldsymbol{d}\}$, $\{a, b_2\}\times\{e,f\}$) in Figures 3.4(b), where both (a) and (b) are path consistent. After subtraction, d is split into three values, whose edges (b_2, d_2), (d_2, e), and (d_1, f) are path inconsistent. The reason is due to the multiplicative effect of the number of local solutions involving d ($size(d)$), while PC among its neighborhood only has the additive effect on some of those local solutions.

In order to prove that *extractCycle()* terminates, we need to define the following notion of neighborhood path consistency.

Definition 15 (Path Consistent with Respect to Cycle) *Given a cycle involving an arc* (X, Y)*, a tuple* $(a, b) \in C_{XY}$ *is* path consistent with respect to the cycle *if and only if* (a, b) *can be extended to the whole cycle by finding values that satisfy all the constraints in the cycle.*

Definition 16 (Neighborhood Path Consistency) *A value* $a \in D_X$ *is* neighborhood path consistent (NPC) with respect to a cycle C *if and only if* (a, b) *is path consistent with respect to* C *for all* $b \in N_X$*.*

The proof of the following lemma is similar to that of of Lemma 2.

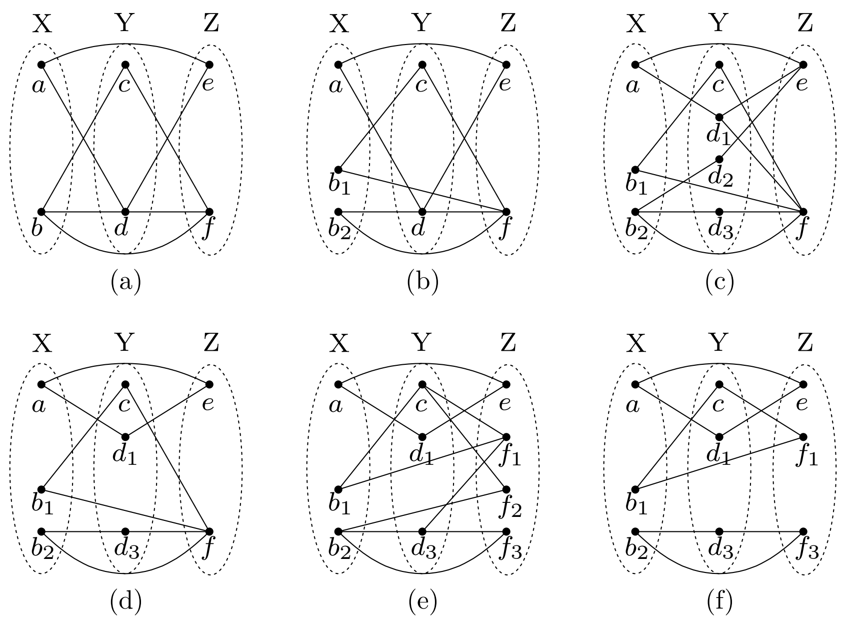


Figure 3.4: Extracting (b, d, f). Figure (a) is the initial microstructure which contains three entangled solutions. In (b) D_X is transformed; enforcing PC has no effect on the result. Figure (c) depicts the network after D_Y is transformed; PC is later enforced, resulting in (d). Next, D_Z is transformed as shown in (e). The final result after PC processing is shown in (vi). Notice that the three solutions are now disconnected.

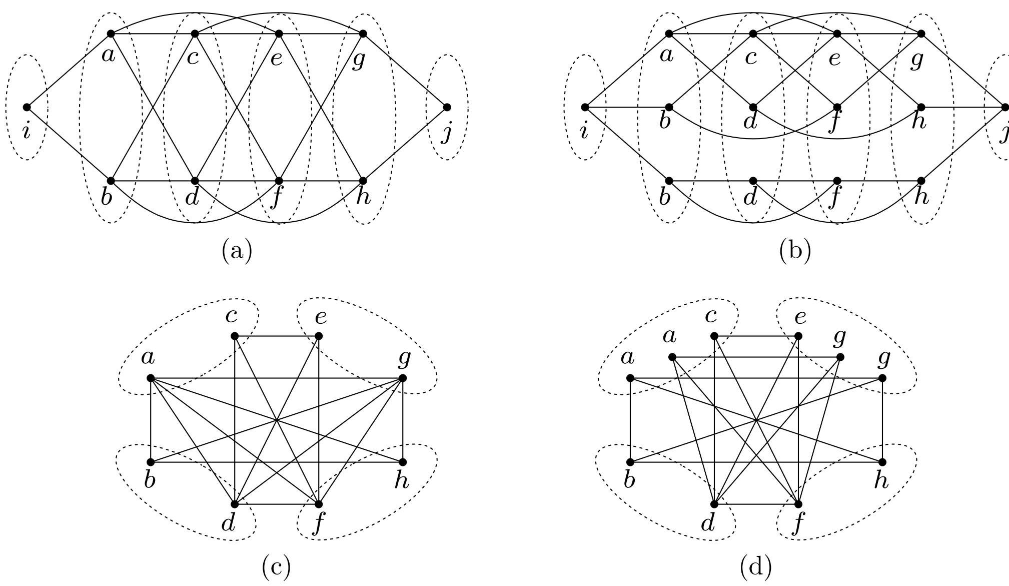


Figure 3.5: More complex extraction. (a) and (c) are the original networks while (b) and (d) are the results after extracting (b, d, f, h) and (a, b, g, h) respectively. Note that if (a, d, f, g) were to be extracted in (c) instead the result in (d) would contain three disconnected solutions.

Lemma 3 *Given a value $a \in D_X$ and a cycle C involving X, an upper-bound on the number of NPC values with respect to C having the same label as a in an equivalent constraint network is d^n.*

In the rest of the chapter when we say a value $a \in D_X$ is NPC that means it is NPC with respect to the all the cycles involving X in the input π. Non-terminating transformation involves generating non-NPC values, which can be removed by PC processing. Note that NPC implies NAC but not vice versa. We will use NPC instead of NAC in the termination proof for *extractCycle()*.

Theorem 8 extractCycle() *is correct and terminates.*

Proof: The proof is similar to that of *extractLine()* except for the algorithm termination. Like the proof of *extractLine()*, we will show that each subtraction will bring a certain measure closer to a finite bound. However, with no restriction on π, not all values in R of the algorithm are NPC and there is no guarantee that a subtraction will produce at least an NPC value. We will instead use the bound on the number of NPC values together with a bound on a new measure involving π.

We define the following measure $size(\pi) := \sum_{1 \le i \le m} \sum_{W \in N_{var(t_i)}} |\sigma_{t_i}(W)|$. Observe that after the algorithm is finished, $size(\pi)$ must be less than that of the original input assignment by a finite amount. We denote ΔL to be the value of that amount and we will use this bound along with the upper-bound in Lemma 3 (denoted by U) to prove algorithm termination. Specifically, we will show that: (1) a subtraction either produces an NPC value having the same label as t_k or decreases $size(\pi)$ by at least one. (2) a subtraction does not increase $size(\pi)$. (3) a subtraction that decreases $size(\pi)$ by one will decrease the number of NPC values by at most d^n. (In contrast, the number of NAC values in the proof of *extractLine()* does not decrease.) As a result, $size(\pi)$ decreases no more than ΔL times and the number of NPC values increases no more than $U + d^n \Delta L$ times. The algorithm terminates since these bounds are finite and either the decrease or the increase must happen after each subtraction.

We prove the three conditions as follows:

(1) Assume a subtraction does not produce an NPC value having the same label as t_k. Since t_k subsumes $unit(t_k, \pi)$, we will focus on the arc (t_k, s) where $s \neq t_i$ for any i and $s \in \sigma_{t_k}(W)$ for some W. (t_k, s) will be removed during the PC processing that follows the subtraction; otherwise it will form part of an NPC sub-value, contradicting the assumption. Thus the value of $size(\pi)$ will be

decreased by the number of such arcs. (E.g (d, e) and (d, a) in Figure 3.4(b) are deleted in (d), thereby reducing the measure by 2. We emphasize that these arcs are not path inconsistent by themselves — indeed, both arcs are PC in (b). They are removed due to the *combination* of subtraction and PC processing.)

(2) Since subtraction preserves local solutions of t_k, if an arc is removed it would be replaced by an equivalent arc that leads to the same local solutions. (E.g. (c, d) in Figure 3.2(a) is deleted in (b), but it is replaced by (b, c_2). Both arcs link b to d via c.) An increase in the number of arcs means an increase in the number of local solutions, which is not possible.

(3) If an arc is deleted, a number of values could lose the NPC property. We simply take the maximum d^n.

Remark: we cannot rely on the reduction of the measure $\mathit{size}(\pi)$ (or other similar measures based on the number of links involving π) alone to prove the algorithm termination since it does not always decrease after subtraction. For instance, $\mathit{size}((b, c_3, e, f, h))$ in Figure 3.2(b) is 12, the same as $\mathit{size}((b, c_3, e, f_3, h))$ in (c). □

Space complexity of *extractCycle()* is the same as that of *extractLine()* for the same reason. Time complexity is exponential in the worst case as a result of the upper bound used in the proof. The bounds in both proofs are admittedly very loose but they are in no way an indication of the actual number of passes. Our concern is to show that the algorithms terminate. Nevertheless, we expect the algorithm to be used in some specific context as an auxiliary routine to other algorithms, to be used occasionally, so that even if the time complexity is higher that would not render it impractical. An algorithm which is more expensive can still be applied as a preprocessing step for some applications. We will discuss further applications in Section 3.6.

Another example of the process is given in Figure 3.6. Since the input is assumed to be consistent, we only need to enforce PC on new arcs. That is, after subtracting $\mathit{unit}(b, \pi)$ we need to find a support for (a, b_1), (b_1, c), (b_1, f), (e, b_2), (b_2, c), and $\{(i, j) \mid i \in \{b_1, b_2\}, j \in \{g, d\}\}$. Moreover, since PC is enforced just to eliminate non-NPC values, we do not need to record a new constraint. In this example we do not need to add C_{YW}.

Although we have shown in the proof that PC is enough for extracting cycle of any length, one might think that extracting an assignment involving k-clique would require k-consistency. This is not true since PC is used only to eliminate

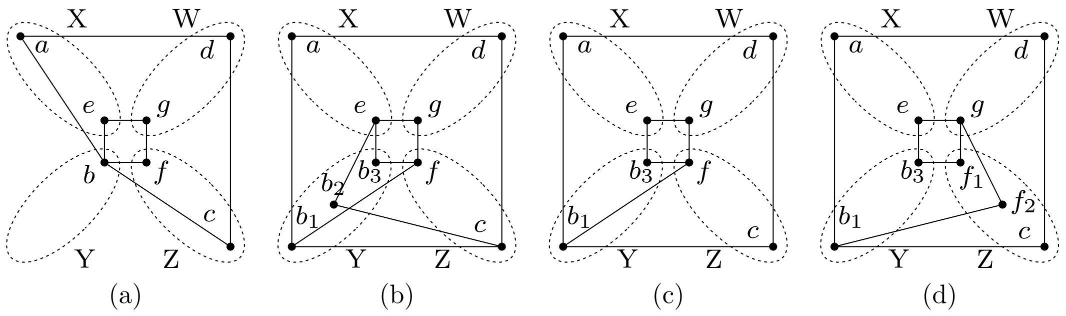


Figure 3.6: Extracting (b, e, g, f). (b) shows the result after the first subtraction. At this point PC is enforced. Since no arc from b_2 to any value in D_W is PC, b_2 is removed ((b_2, g) has no support in Z and (b_2, d) has no support in X.) The result is shown in (c). (d) depicts the network after the next subtraction. After enforcing PC, f_2 is removed.

values that will not form proper cycles, not to solve any sub-problem. Because a k-clique can be decomposed into smaller cycles, the subtraction operation together with PC suffice in extracting any type of assignment, as long as it is consistent. Examples of more complex extraction are given in Figure 3.5, which includes a 4-clique and two overlapping cycles.

3.5 Extracting Microstructure

We can generalize the subtraction process to cover the case when the input is a microstructure rather than an assignment. We formally define a microstructure as follows.

Definition 17 (Microstructure) *A* microstructure *of a binary constraint network $\mathcal{P}$ is a graph $\mathcal{M} = (V, E)$ in which $V \subseteq \mathcal{D}$, and if an edge (a, b) is in E then values a and b are compatible in $\mathcal{P}$. We use* $\text{dom}(\mathcal{M})$ *to indicate* $\bigcup_{a \in V} \text{var}(a)$. *A microstructure $\mathcal{M} = (V, E)$ is* arc consistent *if and only if given $a \in V$, $X = \text{var}(a)$, and $C_{XY} \in \mathcal{C}$, there exists $(a, b) \in E$ for some $b \in D_Y$.*

Since we define the set of vertices as a subset of domain values, the support structure of a value in a microstructure follows that of the whole network. However, we need to define a value whose support structure conforms with only edges in the microstructure.

Definition 18 (Unit Value with Respect to Microstructure) *Given an arc consistent microstructure $\mathcal{M} = (V, E)$ and a value $a \in V$, the* unit value *of a* with respect to *$\mathcal{M}$ (denoted by* unit*$(a, \mathcal{M})$) is a value u such that $L_u = L_a$, and*

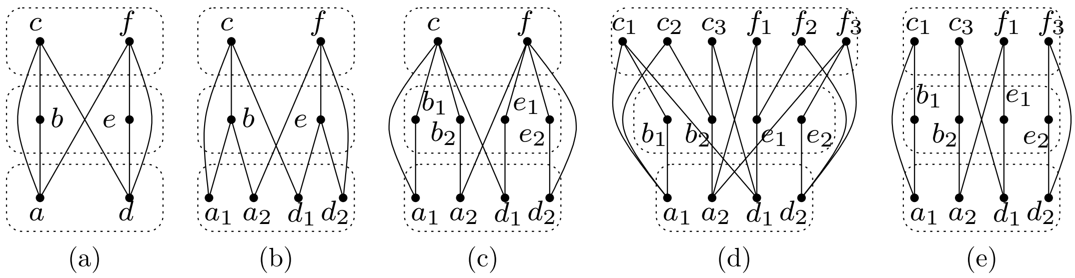


Figure 3.7: Extracting cycle (a, b, c, d, e, f). (a) is the original network; (e) is the result. Notice that the cycle (a, b, c, d, e, f) is arc consistent but contains no solution (arcs (a, c) and (d, f) are not part of the cycle). From (d) to (e) we only enforce PC on values in R of the algorithm so that arcs involving c_3 and f_1 are not checked for PC.

for any X in the neighborhood of var*(a)*

$$\sigma_u(X) = \begin{cases} \{b \in V | X = \text{var(b)} \ \textit{and} \ (a, b) \in E\} & \textit{if } X \in \text{dom}(\mathcal{M}) \\ \sigma_a(X) & \textit{otherwise} \end{cases}$$

The algorithm for extracting microstructure (*extractStructure()*) is similar to *extractCycle()* except the input is an arc consistent microstructure $\mathcal{M}$ and we use the unit value according to Definition 18 rather than Definition 12. Notice that if a microstructure $\mathcal{M}$ contains single value per domain then it is also an assignment. An arc consistent microstructure $\mathcal{M}$ according to Definition 17 is equivalent to the consistent assignment $\mathcal{M}$ according to Definition 8. This means an input $\mathcal{M}$ for *extractStructure()* need not contain a solution for *dom*$(\mathcal{M})$.

The correctness and termination proof is similar to that of *extractCycle()*. *extractStructure()* is strictly more powerful than *extractCycle()* since its input may involve more than one value from the same domain. An example of microstructure extraction is given in Figure 3.7.

3.6 Applications

Maintaining constraint arity. A common approach taken to characterize a tuple as a nogood is to update the constraint involved so that the tuple would not be tried in the future. When constraints are table-based, this can be done by simple record-keeping. If the constraint does not exist it must be created. When the tuple is of size k, the resulting network will have at least one constraint of arity k. This change in topology is problematic for algorithms that presume the maximum arity of constraints to be bounded. Algorithms that work only for binary CSP [BCvBW02] would be rendered useless when the constraint arity increases, even though the network starts out as binary. This is especially significant for distributed CSPs in which agents perform two-way communication. We resolve this problem by first extracting the target tuple and removing it by deleting one of the values involved from its domain. The extraction is done along the existing constraints and no new, higher-arity constraint is required.

Adaptive consistency [DP87] is one example of algorithms that produce non-binary nogoods. By using the extraction process, an initially binary constraint network will remain binary after applying adaptive consistency. Indeed, the arcs in the *ordered constraint graph* would be identical to the constraint network itself. Adaptive consistency has been used in the context of real-time constraint satisfaction [BCFR04], whose authors chose to delete domain values involved in nogoods. However, some solutions may be lost. In contrast, the removal of nogoods by extraction would allow complete solution retention without the need for extra storage space for new constraints.

As our algorithms work only for binary CSP at the moment, an interesting future research is to extend the process to directly cover non-binary constraints (though conversion to binary CSP is possible with good performance [SS05]). This will allow the original arity of the network to be maintained regardless of its initial value; algorithms that work only on k-ary constraints but produce nogoods involving g-ary constraint as a side-effect where $g > k$ would continue to work.

Learning nogoods by pruning. While the common approach used in nogood learning is to record each nogood as it arises [FD94], we can extract and discard a nogood instead. Nogoods of higher order can then be avoided without requiring a large amount of space. Although in the worst case, the resulting microstructure can be quite large, this is offset by the fact that we “learn” a nogood by pruning part of the microstructure away — in effect embedding the knowledge in the network itself — rather than by explicit memorization; the dynamic would balance the overall size as search progresses. This kind of learning is independent of variable ordering and is especially useful for non-systematic search whose completeness relies on nogood store, which becomes very large over time, such as weak-commitment search [Yok94].

Enforcing NIC in one pass. NIC is a powerful technique and has shown to be stronger than many other types of consistency [DB01]. NIC requires that a value participates in a solution of its neighborhood; otherwise, the value is removed and the effect is propagated. Solving a sub-problem for a solution is a relatively expensive task however. The propagation would further increase the total cost, as every single neighborhood-inverse-inconsistent value must be removed to fully achieve NIC. In practice, NIC is rarely used and/or limited to a single pass.

We suggest a way to fully enforce NIC in one pass as follows. When a solution in the neighborhood is found, we extract it out together with the value it supports. This ensures each NIC value has only a single solution as its support after the first pass. Afterward, no sub-problem needs to be solved and further neighborhood-inverse-inconsistent values are deleted by AC propagation alone.

It remains to be seen whether it is possible to apply the idea to other higher-order consistency techniques that require multiple passes, such as Singleton Arc Consistency [BD05, LC05].

Extracting unsolvable sub-problems. In contrast to the previous use for NIC, we can extract sub-problems that are known to be unsolvable and discard them to reduce the search space. This method (using *extractStructure()*) is strictly more powerful than nogood recording since each unsolvable sub-problem may involve more than one value from the same domain. In [FH95], a constraint network is decomposed into disconnected sub-problems by recursively splitting *variable domains*. The result is a collection of independent constraint networks with redundant variables. Microstructure extraction is more efficient since only the target sub-problem is isolated while the rest of the network remains intact; everything is contained in just one CSP.

We can use this technique to partially enforce k-consistency by preprocessing a CSP so that any pattern in the microstructure matching known k-inconsistent subproblems in the portfolio — for instance a pigeonhole problem — is extracted and eliminated, without the usual time complexity associated with k-consistency.

Remark It has been shown in [vBD95] that a path consistent row-convex constraint network is globally consistent. However, this theorem does not imply that enforcing path consistency on a row-convex constraint network would yield a solution since the result may not remain row-convex. Given the way the subtraction operation on value is used in this chapter, one might hope to fix the constraints by splitting problematic values in order to transform a row with non-consecutive 1's into several rows with consecutive 1's. Unfortunately, this method does not preserve the convexity of related columns (if the column in a constraint C_{XY} is non-convex, so is the corresponding row in C_{YX}).

3.7 Conclusions

We have introduced a novel process based on value-splitting that is able to extract the target tuple/microstructure while preserving all the solutions. A number of applications are suggested. As the process involves only the most basic CSP model and is not restricted to any specialized problem or constraint, we believe that once this process is recognized more wide-ranging applications will appear. Since the extraction process increases the size of domains and constraints, it is most suitable for dynamic algorithms that are able to prune parts of the microstructure away as they run, so that the increase and the decrease in size would cancel each other out.

Although the algorithms have yet to be evaluated empirically, in this chapter we place more emphasis on demonstrating how a different view of domain values can give new approaches to existing problems. We plan to investigate implementation issues, such as ordering heuristics (that is, how best to pick a value in line 2 of Algorithm 4) and how to efficiently implement the subtraction operator, and carry out experimental studies in future work.

Chapter 4

A Refutation Approach to Neighborhood Interchangeability in CSPs

4.1 Abstract

The concept of Interchangeability was developed to deal with redundancy of values in the same domain. Conventional algorithms for detecting Neighborhood Interchangeability work by gradually establishing relationships between values from scratch. We propose the opposite strategy: start by assuming everything is interchangeable and disprove certain relations as more information arises. Our refutation-based algorithms have much better lower bounds whereas the lower bound and the upper bound of the traditional algorithms are asymptotically identical.

4.2 Introduction

Interchangeability was introduced in [Fre91] in order to deal with redundancy of values in the same variable domain. Removing or grouping interchangeable values together has proved useful in reducing search space and solving time [BCZ01, LCF05, WF99, Has93].

Neighborhood Interchangeability (NI) can be detected in quadratic time by the Discrimination Tree algorithm (DT) [Fre91]. DT works by assuming zero knowledge and build up relationships between values. The disadvantage is that determining whether a value is NI with another one requires all values to be checked.

We propose a different method that is able to detect values that are not NI early on, without checking all the values. Initially we assume values are identical. That is, they are NI with each other. For each value in the neighboring variables, we test its consistency against these values and update our assumption about their relationships. When enough is known so that a value is certain not to be NI with any other value, it can be removed from future consideration.

In this chapter, we will study algorithms that can efficiently identify neighborhood interchangeability using this approach. The chapter is organized as follows. Section 4.3 gives the background for CSPs and Interchangeability. Section 4.4 gives the overall flow of the algorithms, with concrete algorithms for identifying NI provided in Section 4.6. We show that these algorithms have much better lower bounds than DT, which is explained in Section 4.5. We conclude in Section 4.8.

4.3 Preliminaries

A finite constraint network P is a pair $(\mathcal{X}, \mathcal{C})$ where $\mathcal{X}$ is a finite set of n variables and $\mathcal{C}$ a finite set of e constraints. Each variable $X \in \mathcal{X}$ has an associated domain containing the set of values allowed for X. The initial domain of X is denoted by $dom^{init}(X)$; the current one by $dom(X)$. Each constraint $C \in \mathcal{C}$ involves an ordered subset of variables of $\mathcal{X}$ called scope (denoted by $scp(C)$), and an associated relation (denoted by $rel(C)$). For each k-ary constraint C with $scp(C) = \{X_1, \ldots, X_k\}$, $rel(C) \subseteq \prod_{i=1}^{k} dom^{init}(X_i)$. For any $t = (a_1, \ldots, a_k)$ of $rel(C)$, called a tuple, $t[X_i]$ denotes a_i. A constraint's relation can be described by a formula (called *intensional form*) or by exhaustively listing all the tuples (called *extensional form* or *positive table constraint*). Alternately, a relation may describe tuples *not* allowed; if the relation is in extensional form it is also called *negative table constraint.* The maximum number of tuples in the network is denoted by s. We denote the maximum size of a domain by d, the maximum arity of all constraints by r and the maximum number of constraints involving a single variable with g.

Let C be a k-ary constraint and $scp(C) = \{X_1, \ldots, X_k\}$, a k-tuple t of $\prod_{i=1}^{k}$ $dom^{init}(X_i)$ is said to be: (1) *allowed* by C if and only if $t \in rel(C)$, (2) *valid* if and only if $\forall X_i \in scp(C)$, $t[X_i] \in dom(X_i)$, (3) *a support* in C if and only if it is allowed by C and valid, and (4) *a conflict* if and only if it is not allowed by C and valid. A tuple t is a *support of* (X_i, a) in C when t is a support in C and $t[X_i] = a$. A *constraint check* determines if a tuple is allowed. A *validity check* determines if a tuple is valid. A *solution* to a constraint network is an assignment of values to all the variables such that all the constraints are satisfied. A constraint network is *satisfiable* if it has at least one solution. A Constraint Satisfaction Problem (CSP) involves determining whether a given constraint network is satisfiable.

We assume that values in different domains are different, so that $a \in dom(X)$ and $a \in dom(Y)$ are different values. The domain must be mentioned to distinguish which a is referred to, unless it is clear from the context.

Some of the Interchangeability concepts introduced in [Fre91] are review below.

Definition 19 (FI) *A value $a \in dom(X)$ is* fully interchangeable *with a value $b \in dom(X)$ if and only if (1) every solution which contains a remains a solution when a is replaced with b (2) every solution which contains b remains a solution when b is replaced with a.*

Since identifying FI values amounts to finding all solutions to a constraint network, the process is intractable because the general CSP itself is NP-complete. A weaker but sufficient condition for FI is Neighborhood Interchangeability (NI).

Definition 20 (NI) *Two values $a, b \in dom(X)$ are* neighborhood interchangeable *if and only if for every constraint C such that $X \in scp(C)$,*

$$\{t \in \bar{D} \mid (a,t) \text{ satisfies } C\} = \{t \in \bar{D} \mid (b,t) \text{ satisfies } C\}$$

where $\bar{D} = \prod_{Y \in scp(C) \setminus \{X\}} dom(Y)$.

4.4 Overall Process

We give general routines for identifying and eliminating neighborhood redundant values in Algorithm 5 and 6. A value is redundant if its removal from the satisfiable constraint network does not render it unsatisfiable. Algorithm 5 can be instantiated appropriately to get the different specific algorithms.

The routine BuildStruct creates a data structure that allows FilterStruct to eliminate redundant values efficiently. If FilterStruct has detected and removed some redundant value, it puts the remaining domain in the data structure filStruct and propagates the result. BuildDom rebuilds the domain.

CreateTupleList collects the tuples in the neighborhood. In the worst case, the running time of CreateTupleList as well as the size of tupleList is $O(gd^{r-1})$. If constraints are in extensional form, the running time becomes $O(gs)$, whereas the size of tupleList becomes $O(g.\min(s, d^{r-1}))$. In the rest of the chapter we will use l to denote the size of tupleList.

Algorithm 5: RedundancyCheck($\mathcal{X}$,$\mathcal{C}$)

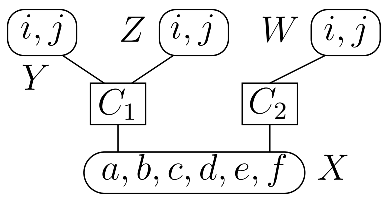
```
1 $Q \leftarrow \{X \mid X \in \mathcal{X}\}$
2 while $Q \neq \emptyset$ do
3     extract $X$ from $Q$
4     tupleList ← CreateTupleList($X$)
5     valStruct ← BuildStruct($X$, tupleList)
6     if FilterStruct(valStruct, filStruct) then
7         $dom(X)$ ← BuildDom($X$, filStruct)
8         for $C \in \mathcal{C}$ such that $X \in scp(\mathcal{C})$ do
9             $Q \leftarrow Q \cup \text{scp(C)} \setminus \{X\}$
```

Algorithm 6: CreateTupleList(X)

```
1 tupleList ← $\emptyset$
2 for $C \in \mathcal{C}$ such that $X \in scp(\mathcal{C})$ do
3     if $rel(C)$ is in intensional form then
4         tupleList ← tupleList $\cup \prod_{Y \in \text{scp}(C) \setminus \{X\}} dom(Y)$
5     else
6         for tuple $t \in rel(C)$ do
7             $t'$ ← the resulting tuple after removing $t[X]$ from $t$
8             tupleList ← tupleList $\cup \{t'\}$
9 return tupleList
```

4.5 Identifying NI by Discrimination Tree

An efficient algorithm for detecting NI values, called the Discrimination Tree algorithm, was introduced in [Fre91]. The idea is to focus on a single value



C_1		
X	Y	Z
a	i	j
b	i	i
b	i	j
c	i	i
e	i	i

C_2	
X	W
c	i
c	j
d	j
e	i
e	j

(a)

C_1		
X	Y	Z
b	i	i
c	i	i
e	i	i
a	i	j
b	i	j

C_2	
X	W
c	i
e	i
c	j
d	j
e	j

(b)

Figure 4.1: (a) NI example. Positive tables. (b) C_1 and C_2 when sorted on (Y, Z) and W.

$v \in dom(X)$ and go through values (or tuples for non-binary constraints) in the neighborhood in some fixed order and build a tree based on their consistency with v. Subsequent checks for other values in $dom(X)$ would begin from the root of the existing tree and follow the path in the tree having the same consistency until reaching the node where consistency differs, at which point a new branch is created. After the algorithm is finished, values that are NI will be grouped in the same leaf. The Discrimination Tree algorithm is shown in Algorithm 7. We remark that the algorithm here is different from the ones in [Fre91, LCF05] but is still based on a discrimination tree with the same time complexity.

For an example of DT, consider the constraint network in Figure 4.1a. The tree of this network is shown in Figure 4.2a. The algorithm requires 24 constraint checks and reports $\{c, e\}$ as the only set of NI values.

The worst-case running time of DT is $O(dl)$. We emphasize that the lower bound of DT is $\Omega(dl)$, leaving the algorithm with no room for improvement.

Algorithm 7: BuildStruct<DT>(X, tupleList)

```
1 create a root node
2 for $a \in dom(X)$ do
3     move to the root node
4     for $t \in$ tupleList do
5         if $a$ is consistent with $t$ then
6             if node corresponding with $t$ existed then
7                 move to that node
8             else construct node corresponding with $t$
```

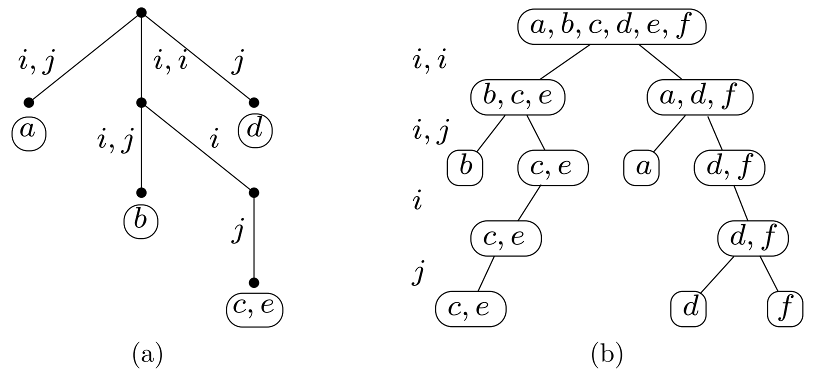


Figure 4.2: Two approaches to establish NI: (a) Discrimination Tree (b) Refutation Tree.

4.6 Identifying NI by Refutation

We propose an opposite approach to DT in identifying NI values. Instead of starting with zero knowledge about value interchangeability, we assume in the beginning that all values are NI and update the assumption as more data becomes available. We call this algorithm Refutation Tree (RT). Details are shown in Algorithm 8.

Algorithm 8: BuildStruct<RT>(X, tupleList)

```
1  thisC ← $\{\{v \in dom(X)\}\}$
2  for $t$ ∈ tupleList do
3      nxtC ← $\emptyset$
4      for $T$ ∈ thisC do
5          lft ← $\emptyset$
6          rgt ← $\emptyset$
7          for $a \in T$ do
8              if $a$ is consistent with $t$ then
9                      lft ← lft ∪ $\{a\}$
10             else  rgt ← rgt ∪ $\{a\}$
11         if  |lft| $\geq 2$ then  nxtC ← nxtC ∪ {lft}
12         if |rgt| $\geq 2$ then  nxtC ← nxtC ∪ {rgt}
13         if  nxtC = $\emptyset$ then  return $\emptyset$
14         else thisC ← nxtC
15 return thisC
```

We describe the algorithm as follows. The set thisC consists of sets of values, which correspond to the nodes in the refutation tree. The RT algorithm works by traversing the refutation tree in a breath-first fashion. For each tuple in tupleList, the algorithm checks whether it is compatible with the values from

Algorithm 9: FilterStruct<RT>(V,F)

```
1 revise ← false
2 result ← ∅
3 for $T \in V$ do
4     if $|T| > 1$ then
5         revise ← true
6         pick $v \in T$
7         result ← result ∪ $(T \setminus \{v\})$;
8 F ← result
9 return revise
```

Algorithm 10: BuildDom<RT>(X,F)

```
1 return $dom(X) \setminus$ F
```

each set in thisC (line 4, 7, 8). A set is split into lft set (consistent values) and rgt set (inconsistent values) for each tuple checked. The result represents the current state of knowledge about NI. A singleton indicates that the value in this set is different from the rest of the domain. The value is discarded (line 11 and 12) since no further data would conflict with what we have learned so far. That is, once it is known that v is inconsistent with t, we will not find out later that v is consistent with t.

For example, consider the tree in Figure 4.2b. In the beginning, all values are assumed to be NI. We then check whether they in fact are consistent with the first tuple (i, i) according to C_1 (a value $x \in dom(X)$ is consistent with (i, i) according to C_1 iff $(x, i, i) \in rel(C_1)$). Only b, c, and e are consistent so they are split off to form a new set. We repeat the process with the next tuple until tupleList is exhausted. It takes 20 constraint checks for RT on this example, compared to 24 for DT.

The worst-case complexity for RT is the same as that for DT. However, RT improves upon DT in lower bound. Because each set in the collection thisC is partitioned into at most two sets, the trace of this process forms a binary tree. The lower bound is achieved when the height of the tree is the smallest possible — that is, when the tree is a complete binary tree. Hence, the lower bound is $\Omega(d. \min(l, \lg d))$.

The efficiency of the RT algorithm depends on the number of NI values and the variable ordering when refutation trees are created. In contrast, the cost of DT is fixed regardless of the number of NI values in a domain. RT requires equal or fewer number of constraint checks than DT in all cases.

4.6.1 Exploiting table constraints

The table constraint, when sorted, is recognized as a simple yet effective way to reduce the cost of CSP algorithms [GJMN07, LS06]. We will show how to exploit sorted table constraints in RT.

Consider the example in Figure 4.1a. For tuple (i, i), Algorithm 8 must check the tuple against each value in $\{a$, b, c, d, e, $f\}$ to partition the set. If the constraints are sorted as shown in Figure 4.1b, this is a simple matter of traversing the table from the first row that contains (i, i) to the first row that contains tuple other than (i, i) and collects values in the X columns while traversing. Once (i, j) is encountered, it is clear that no other values except $\{b, c, e\}$ are consistent with (i, i) so there is no need to check the rest of the table.

The running time for RT with sorted table constraints is $O(s)$. Algorithm 8, however, only provides a high-level concept of the refutation approach. It does not suit for exploiting table constraints. Since we do not know in advance which values would be encountered during the traversal of tables for a given tuple, this requires searching in the collection of sets (thisC) for the right sets that contain the same values as in the corresponding section of the table. The cost incurred for the search makes the running time asymptotically higher than $O(s)$.

It is not a trivial task to revise RT so that it is able to exploit sorted table constraints while maintaining its lower bound. In the next section we will give a new algorithm that can process values in any arbitrary order.

Note that DT can also exploit sorted table constraints. In fact, Figure 4.2a can be created just by traversing the tables in Figure 4.1a. We will show later that RT can exploit mixed positive and negative table constraints so that the running time can be decreased to even less than $O(s)$.

4.6.2 Implementation

We provide a detailed algorithm (called RTS) in Algorithm 11. It follows the idea laid out in Algorithm 8 but differs in that it iterates through values in a domain rather than through sets of values in a collection. This is done to facilitate arbitrary orders of values in sorted table constraints. We will explain how sorted table constraints are used in the algorithm later in this section.

We use array bin to partition NI values. Partitions are numbered from 0 to curSize $-$ 1. Initially, all values are assigned to bin[0]. For each tuple in the tupleList the algorithm checks whether it is consistent with values in valList. If a value is consistent, it is taken off the current bin and put into a new bin, whose number is determined by array split. Afterward, the size of the original bin (denoted by array size) and the size of the new bin is updated. The value of split is obtained from a linked-list of available bins (nxtOf and next) and is fixed for a given bin until the next tuple is considered (if-block in line 22).

When all values in the same bin are consistent with the tuple t, they are moved to the new bin, leaving the original bin empty. To avoid having the number of bins exceeds the number of values in the domain, we reuse the empty bin and put it in the linked-list of available bins (if-block in line 26).

If the size of a bin is exactly one, the singleton value will be removed from the valList. This is done in the if-block in line 16. We reduce the size of valList by one and swap the singleton value with the value at the end of valList (line 19). We use array label as another abstract layer of values for this purpose.

We use array rec to keep track of the bin assigned involving a given tuple. It serves two purposes. First, it prevents incorrect reuse of bin. Because the value of split[bin[v]] is correct only for a given t (line 8), as soon as a new tuple is considered, the old value of split[bin[v]] is incorrect, since the bin involving the previous tuple become a separate and independent bin. We enforce this condition by comparing the current tuple with the tuple recorded (line 22). Second, we use rec to prevents premature elimination of a singleton value. To be certain that a partition with a single value will not increase in size later, the algorithm must already finish checking all the values against the current tuple. We enforce this condition by eliminating singleton partitions only after the next tuple is considered (line 16).

Example Let us reconsider the example in Figure 4.1a and the tree in Figure 4.2b. The first tuple is (i, i). After it is checked against values from 0 to 5 (representing $\{a, b, c, d, e, f\}$), we have: split[0]=1, bin[1]=1, bin[2]=1, bin[4]=1 (i.e. bin[1] for b, c, e), bin[0]=0 bin[3]=0 bin[5]=0 (i.e. bin[0] for a, d, f). After (i, j) is checked we have split[0]=2, split[1]=3, bin[0]=2 (for a), bin[1]=3 (for b), bin[2]=1 and bin[4]=1 (for c and e), bin[3]=0 and bin[5]=0 (for d and f). The

Algorithm 11: BuildStruct<RTS>(X, tupleList)

```
for $i \leftarrow 0$ to $|dom(X)| - 1$ do
    size[$i$] $\leftarrow 0$ ;     bin[$i$] $\leftarrow 0$
    rec[$i$] $\leftarrow \emptyset$ ;    split[$i$] $\leftarrow 0$
    label[$i$] $\leftarrow i$ ;   nxtOf[$i$] $\leftarrow i + 1$ ;   del[$i$] $\leftarrow$ false
size[0] $\leftarrow |dom(X)|$
curSize $\leftarrow |dom(X)|$
next $\leftarrow 1$
while curSize $> 0$ and tupleList $\neq \emptyset$ do
    extract $t$ from tupleList
    if sortedT then valList $\leftarrow$ CreateValList($X, t$)
               else valList $\leftarrow \{0, \ldots,$ curSize $- 1\}$
    while valList $\neq \emptyset$ do
        extract $i$ from valList
        if sortedT(and) del[$i$] then continue
        $v \leftarrow$ label[$i$]
        if size[bin[$v$]]=1 $\wedge$ rec[bin[$v$]]$\neq t$ then
            curSize $\leftarrow$ curSize $- 1$
            if sortedT then del[$v$] $\leftarrow$ true
                       else label[$i$] $\leftrightarrow$ label[curSize $- 1$]
        else
            if sortedTor $v$ is consistent with $t$ then
                if split[bin[$v$]]= $\emptyset$ or rec[bin[$v$]]$\neq t$ then
                    split[bin[$v$]] $\leftarrow$ next
                    next $\leftarrow$ nxtOf[next]
                size[bin[$v$]] $\leftarrow$ size[bin[$v$]] $-1$
                if size[bin[$v$]] $= 0$ then
                    nxtOf[bin[$v$]] $\leftarrow$ next
                    next $\leftarrow$ bin[$v$]
                rec[bin[$v$]] $\leftarrow t$
                bin[$v$] $\leftarrow$ split[bin[$v$]]
                size[bin[$v$]] $\leftarrow$ size[bin[$v$]] $+1$
                rec[bin[$v$]] $\leftarrow t$
```

linked list of available bin (indicated by next together with nxtOf) is 4→5→6. After tuple i is checked, two singletons a and b are removed, and we have curSize=4 and label[0]=5, label[5]=0, label[1]=4, label[4]=1, while the rest of label remains unchanged. For values 0 to 3 in valList, bin[label[0]]=bin[5]=0 (for f), bin[label[1]]=bin[4]=4 (for e), bin[label[2]]=bin[2]=4 (for c), bin[label[3]]=bin[3]=0 (for d). The linked list of available bin is 1→5→6 (bin 1 and 5 were previously used for a and b, but are available for reuse now that a and b are ignored). □

Sorted table constraints can be exploited by setting flag sortedT to true. We assume the sorted tables are positive. Negative tables can be accommodated simply by changing "consistent" in line 21 to "inconsistent". The algorithm for sorted table constraints differs from the general algorithm in three places. First, valList is set by routine CreateValList, which traverses the corresponding sorted table and collects the relevant values. For instance, using example in Figure 4.1, valList for (i, i) is $\{b,\ c,\ e\}$. Second, we do not perform constraint check for values gathered in this way from sorted table constraints because we already know their consistency just from their existence in the table (line 21). Third, singleton values are skipped in a different way. The same technique for general constraints cannot be used because valList changes from one tuple to another. Instead, we use boolean array del to indicate whether a value should be ignored (line 14 and 18). Notice that this does not decrease the complexity of the algorithm because all the values in valList must be iterated anyway. The lower bound of RTS is $\Omega(s)$.

It is interesting to note that the algorithm is applicable to mixed constraint of both consistency types. The lower bound of $\Omega(s)$ can be made lower if the shorter sections in either positive or negative relations are combined. For instance, suppose we have $dom(X){=}dom(Y){=}\{1, 2, 3, 4\}$. $rel(C_1) = \{(1, 1),$ $(1, 2),\ (2, 2),\ (3, 2),\ (4, 2),\ (2, 3),\ (4, 3)\}$ lists compatible tuples for X and Y, $rel(C_2){=}\{(2, 1),\ (3, 1),\ (4, 1),\ (1, 3),\ (3, 3),\ (1, 4),\ (2, 4),\ (3, 4),\ (4, 4)\}$ lists incompatible tuples for X and Y. We can gather shorter parts from both constraints to create a mixed constraint C_3, $rel(C_3){=}\{P(1, 1),\ N(1, 3),\ N(3, 3)\}$, where P and N denotes positive and negative tuples. The size of the combined constraint can be much smaller than the size of the original constraints; for this example, $|rel(C_1)| = 7$, $|rel(C_2)| = 9$, $|rel(C_3)| = 3$. We can use C_3 in the algorithm by switching the consistency in line 21 depending on the tuple. While it is rare to have both positive and negative tables for the same constraint, the mixed constraint can be created from only one of them: if the size of the section is less than half the number of all possible tuples then we retain the tuples. If the size is more than half, the tuples in the opposite consistency type would be created by inference.

4.7 Related Works

NI has been shown to improve search in a number of works [LCF05, BCZ01, CD02, Has93]. Although DT was introduced only in the context of binary CSPs, it has been extended to cover non-binary CSPs in [LCF05]. The authors pointed to a case where DT provides incorrect results for non-binary CSPs. To avoid this problem, they suggest performing DT on each neighboring constraint and intersecting the results. Our DT is derived from the binary version in a slightly different way and it does not cause the incorrect results.

In [LCF05], it has also been recognized that singleton partitions can be removed. However, these singletons can be eliminated only after DT is finished for each constraint. This makes the lower bound higher than that of refutation-based algorithms, which can ignore singletons much earlier.

4.8 Conclusions

We introduce a refutation method to local interchangeability and study it in detail for NI. Rather than starting with no prior knowledge, we assume the opposite — that everything is interchangeable with one another — and update our assumption as new information comes along. While the algorithms presented have the same upper bound on their running time as those of the standard algorithms, the refutation approach allows some values that are not NI with others to be detected early and removed from further consideration by the algorithms, thus decreasing their lower bounds. We also show how these algorithms can take advantage of table constraints while still achieving the lower bound described.

Note that Algorithm 11 is provided in low-level details because it is not clear how one can take advantage of sorted constraints while maintaining the same lower bound of the generic RT. Without proper care the lower bound could increase, defeating the whole purpose. Otherwise, implementing RT is straightforward.

Another form of local interchangeability called Neighborhood Substitutability [Fre91] also benefits from the refutation approach. NS has more pruning power but received much less attention than NI due to its higher cost. A direct NS algorithm was given in [BCH^{+}94] but no experimental result was reported.

Our main objective for this chapter is to explore theoretical possibilities that come with this new approach. While the better lower bound does not imply actual performance, it gives practitioners considerable room to maneuver. The practicality of this approach will largely depend on how the average running time can be pushed closer to the lower bound. Since the actual performance of these algorithms are affected by ordering heuristics for the list of tuples in the neighborhood, this is an interesting aspect to explore further. On the other hand, direct algorithms such as DT have the same (worst-case) running time on every input.

Chapter 5

Eliminating Redundancy in CSPs Through Merging and Subsumption of Domain Values[1]

5.1 Abstract

Onto-substitutability has been shown to be intrinsic to how a domain value is considered redundant in Constraint Satisfaction Problems (CSPs). A value is onto-substitutable if any solution involving that value remains a solution when that value is replaced by some other value. We redefine onto-substitutability to accommodate binary relationships and study its implication. Joint interchangeability, an extension of onto-substitutability to its interchangeability counterpart, emerges as one of the results. We propose a new way of removing interchangeable values by constructing a new value as an intermediate step, as well as introduce virtual interchangeability, a local reasoning that leads to joint interchangeability and allows values to be merged together. Algorithms for removing onto-substitutable values are also proposed.

[1]This chapter is a revision of [LY13]. Theorem 4 in [LY13] is incorrect because it alludes to the conventional concept of network equivalence despite the fact that the paper expropriates the term "equivalent" to describe something else. The idea of Theorem 4 still holds and it is restated into what is now Proposition 6. Some definitions and concepts are also modified to improve legibility. The changes are limited to Section 5.4 and Section 5.5 and do not affect the contents of subsequent sections.

5.2 Introduction

An important indicator of the hardness of a constraint satisfaction problem is the size of the search space. Eliminating interchangeable values was introduced in [Fre91] as a way of reducing complexity of a problem by removing portions of the search space that are essentially identical. Recent focus on interchangeability has been on onto-substitutability [BCM08, Fre11] : a domain value is onto-substitutable if any solution involving that value remains a solution when that value is replaced by some other value. Standard substitutability, by contrast, is a binary relation between two fixed elements.

In this chapter, we redefine onto-substitutability as a binary relation and study its consequences. We propose joint interchangeability: two sets are joint-interchangeable iff any solution involving a value in one set remains a solution when that value is replaced by some value in the other set. Joint interchangeability is more practical since any one of the two sets can be eliminated; for onto-substitutability only a single value can be removed. We then propose virtual interchangeability: values are virtually interchangeable if they support the same values in every constraint but one. A set S of virtually interchangeable values can be compactly represented by a value s, in effect making S joint-interchangeable with s. Hence, virtual interchangeability leads indirectly to joint interchangeability.

To make sure virtually interchangeable set of values can be merged into a single value while retaining all the solutions, we expand the definition of domain value to accommodate the notion of label. A value may have more than one label, and labels are what actually appear as part of solutions. We introduce several new ways of comparing networks based on this concept. As a result, we can solve a CSP by transforming it using virtual interchangeability into a more compact network, solve the derived network, and convert it back — all without losing any solution. Moreover, this method works in the context of the hidden transformation, which is a way of transforming non-binary constraints into binary equivalents, by treating table constraints as a form of hidden variables. Preliminary results show that compressing all virtually interchangeable values is a promising approach to simplify table constraints in structured problems.

5.3 Joint Interchangeability

Interchangeability and related ideas were first described in Freuder [Fre91]. Bordeaux et al. [BCM08] provided a formal framework that demonstrates connections among structural properties of CSPs. Removability, to which these properties reduce, is regarded by the authors to be the basis of how a value can be removed without affecting satisfiability of the problem. Freuder [Fre11] later proposed the concept of dispensability: a value is dispensable if removing it will not remove all solutions to the problem. A value can then be dispensable without being removable (or onto-substitutable as called in [Fre11]). Dispensability would therefore appear to be a more fundamental property than onto-substitutability. A survey of interchangeability concepts is reported in [KWC+10].

Onto-substitutability underscores the notion that a value's attributes can be broken down and subsumed by other values. An onto-substitutable value can be removed without affecting satisfiability of the problem *precisely because it is semantically redundant.* For this reason, onto-substitutability is arguably key to understand many interchangeability concepts, but not as fundamental as dispensability when it comes to determine why a value can be removed.

Given the prospects of onto-substitutability, we will focus on this property and its derivations. First, we give the formal definition of CSP and redefine substitutability so as to make onto-substitutability a binary relation as follows. Any set of values mentioned in this section must be nonempty.

Definition 21 (CSP) *A finite constraint network $\mathcal{P}$ is a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where $\mathcal{X}$ is a finite set of n variables and $\mathcal{C}$ a finite set of e constraints. $D(X) \in \mathcal{D}$ represents the set of values allowed for variable $X \in \mathcal{X}$. Each constraint $C \in \mathcal{C}$ involves an ordered subset of variables in $\mathcal{X}$ called scope (denoted by* $\mathrm{scp}(C)$*), and an associated relation (denoted by* $\mathrm{rel}(C)$*). For any k-ary constraint C with* $\mathrm{scp}(C) = \{X_1, \ldots, X_k\}$, $\mathrm{rel}(C) \subseteq \prod_{i=1}^{k} D(X_i)$. *For any k-tuple $t = (a_1, \ldots, a_k)$ over $\mathcal{T} = \{X_1, \ldots, X_k\}$ such that $X_i \in \mathcal{X}$ and $a_i \in D(X_i)$, $t[X_i]$ denotes a_i, and* $\mathrm{scp}(t)$ *denotes $\mathcal{T}$. If $\mathcal{S} \subseteq \mathcal{T}$ then $t[\mathcal{S}]$ denotes the tuple over $\mathcal{S}$ obtained by restricting t to the variables in $\mathcal{S}$. A* solution *of $\mathcal{P}$ is a member of* $\mathrm{rel}(\mathcal{P}) = \{t \mid scp(t) = \mathcal{X} \wedge \forall C \in \mathcal{C} \,.\, t[scp(C)] \in rel(C)\}$. *$\mathcal{P}$ is* satisfiable *iff $rel(\mathcal{P}) \neq \emptyset$.*

We present new definitions and re-define substitutability and onto-substitutability as follows.

Definition 22 *Given value $v \in D(X)$, the* maximally substitutable *set of v is maxsub(v) = $\{b \in D(X) \mid$ there exists a solution involving v which remains a solution when v is substituted by $b\}$.*

Definition 23 *A value $v \in D(X)$ is* substitutable by *a set of value S iff $S \subseteq$ maxsub(v) and any solution involving v remains a solution when v is substituted by some $b \in S$. S is called a* substitutable set *of v.*

We simply say v is *substitutable* if there exists S such that v is substitutable by S. When $|S| > 1$, v is *onto-substitutable* and that S is an onto-substitutable set of v. A substitutable set of v is *minimal* if there is no strictly smaller substitutable set of v. Distinct minimally substitutable sets may exist for any given value. We also extend substitutability so that a set of value S is substitutable by T iff for any value $v \in S$, v is substitutable by T.

Interchangeability can be redefined to cover many-to-many relationship in a similar fashion.

Definition 24 *A set of value $S \subseteq D(X)$ is said to be* joint-interchangeable (JI) with *a set of value $T \subseteq D(X)$ iff S is substitutable by T and T is substitutable by S.*

When $|S| = 1 = |T|$, JI reduces to ordinary interchangeability. JI is symmetric and transitive. S is *minimally joint-interchangeable* with T if there exist no $S' \subseteq S$ and $T' \subseteq T$ such that S' is JI with T' and $S' \neq S \vee T' \neq T$.

Example 1 *Consider a network involving two variables with solutions $\{(1, a)$, $(2, a)$, $(3, b)$, $(4, b)$, $(5, c)$, $(6, c)$, $(1, d)$, $(3, d)$, $(5, d)$, $(2, e)$, $(4, e)$, $(6, e)\}$. $\{a, b, c\}$ is minimally joint-interchangeable with $\{d, e\}$.*

Take note that the definitions of substitutability and JI allow the same value to appear on both sides of the relations. This helps us identify JI sets that would not otherwise be recognized. The following example illustrates.

Example 2 *Consider a set of solutions $\{(1, a)$, $(2, a)$, $(1, b)$, $(2, c)$, $(3, b)$, $(3, c)\}$. a is not interchangeable with b, but $\{a, c\}$ is minimally joint-interchangeable with $\{b, c\}$.*

Proposition 2 *If S is joint-interchangeable with T, then either $S \setminus T$ or $T \setminus S$ can be eliminated without affecting the network's satisfiability.*

In [BCM08] the authors claim that local reasoning is not sound for onto-substitutability. Freuder [Fre11] shows that this is not the case if the scope

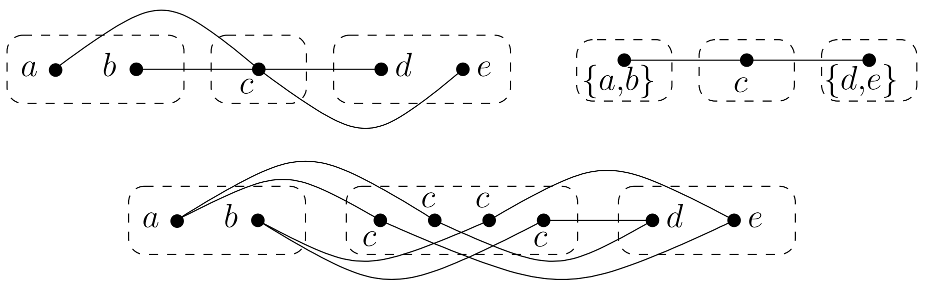


Figure 5.1: Three synonymous networks. Dots represent domain values, while lines connect values that are compatible. Dash lines enclose values from the same domain. a, b, and c are labels. Two values have more than one label: one tagged with $\{a, b\}$ and the other $\{d, e\}$.

of "local reasoning" is broadened to include closure on sub-problems. The same rationale can be applied to joint interchangeability.

JI is a stronger than onto-substitutability but proves to be more useful. The latter lets us eliminate objects only from one side of the relation, whereas JI allows either side to be removed. Since a CSP with more values generally translates to longer search, once it is known S is JI with T an easy way to simplify the problem is to eliminate the larger set. Conversely, we want to identify two JI sets such that their size difference is as large as possible. We introduce and study a local reasoning which takes advantage of this fact called virtual interchangeability in Section 5.5.

Onto-substitutability can be weakened further by considering substitutability in only some solution.

Definition 25 *A value v is* nominally substitutable *by b iff there exists a solution involving v and it remains a solution when v is substituted by b. A value v is simply said to be* nominally substitutable *when there exists such b, that is, when* maxsub*$(v) \neq \emptyset$.*

Nominal substitutability and dispensability depend on the existence of a solution. Local reasoning such as closure is thus ineffective, because extending some solution in a bounded area to the whole problem is just as computationally difficult as finding a new solution from scratch. Nominal substitutability is equivalent to "minimal substitutability" in [Fre11] and "context dependent interchangeability" in [WFC96].

Recombination of values in the smallest closure was studied in [BL04] and it was shown that two values are nominally substitutable iff their structure in the closure overlap and a fragment of their intersection involved is in a solution.

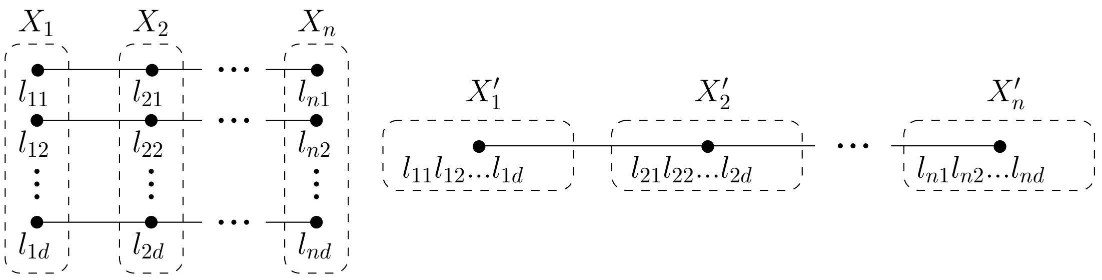


Figure 5.2: $\mathcal{Q}$ on the right subsumes $\mathcal{P}$ on the left.

5.4 Enhancing Domain Values

Domain values serve two main purposes: structurally and semantically. These two aspects are intertwined in most CSP models, yet we would have more flexibility in manipulating a network when they are decoupled. To this end, we extend the definition of a value in this section. Combined with virtual interchangeability, this allows us to merge values while preserving all solutions at the same time.

Definition 26 (Labels) *A value v of a constraint network $\mathcal{P}$ is a tuple (uid, lab) where uid is an identifier unique to this value in $\mathcal{P}$ (also denoted by* uid*(v)) and lab is a set, which we also call the value's* labels*. (also denoted by* lab*(v)). A value must have at least one label.*

This enhanced definition allows a domain value to possess multiple labels while at the same time the same label can be associated to different domain values — a many-to-many relationship, in other words. Through labels, a single tuple can represent multiple tuples of the standard kind.

New definitions based on labels are given as follows.

Definition 27 *Given networks $\mathcal{P}$ and $\mathcal{Q}$ such that $X_1, \ldots, X_k$ are variables of both $\mathcal{P}$ and $\mathcal{Q}$, and tuple t where* scp*(t)* $= \{X_1, \ldots, X_k\}$*, then*

- sol*(t) denotes* $\prod_{i=1}^{k}$ *lab*$(t[X_i])$*.*
- *A member of sol(t) is called a rendition of t.*
- *A member of sol$(\mathcal{P}) := \bigcup_{t\in rel(\mathcal{P})}$ sol(t) is called a* rendition *of $\mathcal{P}$.*
- *A rendition of t* agrees *with $\mathcal{P}$ if it is also a rendition of $\mathcal{P}$.*
- *A label of value $t[X_i]$, for some X_i, is* valid with respect to *t and $\mathcal{P}$ if it is involved in some rendition of t that agrees with $\mathcal{P}$. It is* invalid *otherwise.*

- $\mathcal{Q}$ subsumes $\mathcal{P}$ *if* $sol(\mathcal{Q}) \supseteq sol(\mathcal{P})$*.*
- $\mathcal{P}$ *and* $\mathcal{Q}$ *are* synonymous *if* $sol(\mathcal{P}) = sol(\mathcal{Q})$*.*

An example of synonymous networks is shown in Figure 5.1.

We are interested in the type of network transformation that offers a second pathway to solve the original network. Subsumption in Definition 27 provides a necessary condition for any such transformation: a transformed network must, at the very least, preserve all solutions of the original. That is, suppose we are given a network $\mathcal{P}$ and its transformation $\mathcal{Q}$ such that $\mathcal{Q}$ subsumes $\mathcal{P}$, then we can find all solutions of $\mathcal{P}$ by solving $\mathcal{Q}$. Still, this method may in fact entail as much work as solving $\mathcal{P}$ itself.

For now, we remark on another issue with subsumption. Although the solutions of the original are preserved, they are not guaranteed to be properly distributed in the transformed network $\mathcal{Q}$. For example, given some solution t of $\mathcal{Q}$, it is possible that $sol(t) \cap sol(\mathcal{P}) = \emptyset$, which means t covers no solution of $\mathcal{P}$ at all. The effort spent finding t is therefore wasted. It is more useful to have a transformation such that any solution of the transformed network can always be converted to a solution of the original. This condition is what we call *conformity* in the following definition.

Definition 28 *Given networks* $\mathcal{P}$ *and* $\mathcal{Q}$*,* $\mathcal{Q}$ conforms to $\mathcal{P}$ *if any solution of* $\mathcal{Q}$ *yields a rendition that agrees with* $\mathcal{P}$*.*

According to this definition, an unsatisfiable network conforms (trivially) to any network.

Example 3 *Consider network* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$*, where* $\mathcal{X} = \{X_1, \ldots, X_n\}$*,* $|D(X_i)| = d$ *and* $D(X_i) = \{a_{i1}, \ldots, a_{id}\}$ *for* $1 \leq i \leq n$*,* $lab(a_{ij}) = \{l_{ij}\}$*,* $\mathcal{C} = \{C_1, \ldots, C_{n-1}\}$*,* $scp(C_i) = \{X_i, X_{i+1}\}$ *and* $rel(C_i) = \{(a_{ij}, a_{(i+1)j}) \mid 1 \leq j \leq d\}$ *for* $1 \leq i < n$*, and* $\mathcal{Q} = (\mathcal{X}', \mathcal{D}', \mathcal{C}')$*, where* $\mathcal{X}' = \{X'_1, \ldots, X'_n\}$*,* $D(X'_i) = \{a'_i\}$*,* $lab(a'_i) = \{l_{i1}, \ldots, l_{id}\}$*,* $\mathcal{C}' = \{C'_1, \ldots, C'_{n-1}\}$*,* $scp(C'_i) = \{X'_i, X'_{i+1}\}$ *and* $rel(C'_i) = \{(a'_i, a'_{i+1})\}$*. The networks are illustrated in Figure 5.2.*

The following holds: (1) $\mathcal{Q}$ *subsumes* $\mathcal{P}$*. (2)* $\mathcal{Q}$ *conforms to* $\mathcal{P}$*. (3)* $\mathcal{P}$ *conforms to* $\mathcal{Q}$*. (4)* $\mathcal{P}$ *is satisfiable iff* $\mathcal{Q}$ *is satisfiable. (5)* $\mathcal{P}$ *is not synonymous with* $\mathcal{Q}$*.*

Example 3 shows that it is trivial to find the one solution of the transformed network $\mathcal{Q}$, but obtaining a rendition that agrees with $\mathcal{P}$ amounts to solving the original network itself.

The process of obtaining a rendition from a given solution that agrees with the original network will be discussed in more detail in the next section after we introduce some concrete transformation process.

The following proposition gives some basic properties of conformity.

Proposition 3 *Given networks $\mathcal{P}$ and $\mathcal{Q}$,*

1. *if $\mathcal{Q}$ subsumes $\mathcal{P}$ then $\mathcal{P}$ conforms to $\mathcal{Q}$.*
2. *if $\mathcal{P}$ conforms to $\mathcal{Q}$ then $\mathcal{P}$ is satisfiable implies that $\mathcal{Q}$ is satisfiable.*

Proof: (1) Because a value must have at least one label, any solution of $\mathcal{P}$ must have at least one rendition. Because $\mathcal{Q}$ subsumes $\mathcal{P}$, that rendition agrees with $\mathcal{Q}$. (2) Suppose $\mathcal{P}$ is satisfiable. Let t be a solution of $\mathcal{P}$. Because $\mathcal{P}$ conforms to $\mathcal{Q}$, t yields a rendition r that agrees with $\mathcal{Q}$. That is, r must be a rendition of some solution t' of $\mathcal{Q}$, which means $\mathcal{Q}$ is satisfiable. □

The following proposition is a direct consequence of Proposition 3.

Proposition 4 *Given networks $\mathcal{P}$ and $\mathcal{Q}$,*

1. *if $\mathcal{Q}$ subsumes $\mathcal{P}$ then $\mathcal{P}$ is satisfiable implies that $\mathcal{Q}$ is satisfiable.*
2. *if $\mathcal{P}$ conforms to $\mathcal{Q}$ and $\mathcal{Q}$ conforms to $\mathcal{P}$ then $\mathcal{P}$ is satisfiable iff $\mathcal{Q}$ is satisfiable.*
3. *if $\mathcal{P}$ and $\mathcal{Q}$ are synonymous then $\mathcal{P}$ is satisfiable iff $\mathcal{Q}$ is satisfiable.*

The converse of (2) and (3) do not hold.

Example 3 also provides a counterexample of Proposition 4(3).

5.5 Virtual Interchangeability

Identification followed by elimination has been an established method for dealing with redundant values in CSPs. This practice is simple and straightforward. As a result, much of the attention has been devoted to finding a new kind of interchangeability and developing more efficient algorithms that recognize these properties.

We present a proactive strategy with regard to properties that are binary relations. Given a relation R, rather than the usual passive approach of finding two objects x and y such that xRy and eliminating one of the two, we need only identify x, then create y from x so that xRy, and finally eliminate x. Central to this approach, however, is whether a "better" object y can be created from x. Concretely for JI, instead of identifying two sets that are JI with each other, we focus on finding a set of values that possess a certain property, then by exploiting that property create an equivalent value that represents that set more concisely, and only after the new value is added do we remove the original values.

In this section we introduce the concept of virtual interchangeability, a local reasoning that can be used as outlined above. We begin by recalling necessary definitions.

Definition 29 *The* projection *of constraint C to $\mathcal{S} \subseteq \mathrm{scp}(C)$ is a constraint $\pi_{\mathcal{S}}(C)$ where $\mathrm{scp}(\pi_{\mathcal{S}}(C)) = \mathcal{S}$ and $\mathrm{rel}(\pi_{\mathcal{S}}(C)) = \{t[\mathcal{S}] \mid t \in \mathrm{rel}(C)\}$. The projection of a tuple is defined in the same fashion. The* concatenation *of $t_1 \in C_1$ and $t_2 \in C_2$ ($\mathrm{con}(t_1, t_2)$) is the tuple t resulting from the concatenation of t_1 and t_2 followed by rearrangement so that $\mathrm{scp}(t) = \mathrm{scp}(t_1) \cup \mathrm{scp}(t_2)$.*

Definition 30 *Given two values $a, b \in D(X)$ and constraint C such that $X \in scp(C)$, values a and b are* neighborhood interchangeable with respect to C *if and only if*

$$\{t \in \bar{D} \mid con(a,t) \in rel(C)\} = \{t \in \bar{D} \mid con(b,t) \in rel(C)\}$$

where $\bar{D} = \pi_{scp(C)\setminus\{X\}}(C)$.

Definition 31 *Two values $a, b \in D(X)$ are* neighborhood interchangeable *(NI) [Fre91, LY08] if and only if they are neighborhood interchangeable with respect to C for every constraint C such that $X \in scp(C)$,*

Given NI values, we can combine them into a single value without losing any solution simply by merging their labels into those of the representative value while discarding the remaining values. As a result, the initial network and the network after the NI values are merged are equivalent.

Definition 32 *Two values $a, b \in D(X)$ are* virtually interchangeable *(VI) (with respect to C) iff there is at most one constraint C such that $X \in scp(C)$ and a and b are not neighborhood interchangeable with respect to C. A set of values are* virtually interchangeable *if any two values are virtually interchangeable with each other.*

VI and NI are almost the same except for the difference of supports in a single constraint. NI implies VI but VI does not imply NI. Neighborhood Substitutability (NS) [Fre91] is incomparable to VI. A network that contains no VI values is called VI-free.

The VI-merging process and its implications will be discussed in this section. We begin by describing a generic merging process that can be used to combine any pair of values.

Proposition 5 *Given $a, b \in D(X)$ in network $\mathcal{P}$, a new network $\mathcal{Q}$ can be derived from $\mathcal{P}$ by merging b into a as follows*

- *updating $rel(C)$ for any C involving X by altering any tuple $t \in rel(C)$ where $t[X] = b$ so that $t[X] = a$*
- *setting $lab(a) \cup lab(b)$ to be the new value of $lab(a)$*
- *removing b from $D(X)$*

Consequently,

1. *$\mathcal{Q}$ subsumes $\mathcal{P}$*
2. *$\mathcal{P}$ is satisfiable $\Rightarrow$ $\mathcal{Q}$ is satisfiable.*

Proof: (1) Let $\mathcal{S}$ be the sub-network of $\mathcal{P}$ whose constraint graph forms a star-graph with X as the center and neighbors of X the leaves. Let $\mathcal{S}'$ denotes $\mathcal{S}$ right after b is merged into a. Any solution of $\mathcal{S}$ involving a or b can be made a solution of $\mathcal{S}'$ by replacing b with a since the new a supports any value that either a or b in $\mathcal{S}$ supports. And because the label set of a in $\mathcal{S}'$ incorporates that of a and b in $\mathcal{S}$, it is clear that $sol(\mathcal{S}') \supseteq sol(\mathcal{S})$ (the set $sol(\mathcal{S}') \setminus sol(\mathcal{S})$ may contain spurious tuples that stem from the cross product of the new a's supports). Hence, $sol(\mathcal{Q}) \supseteq sol(\mathcal{P})$. (2) comes from (1) and Proposition 4. □

Observe that although in Proposition 5 the existing designation "a" is chosen to represent the new value, it can be called anything as long as relations involved are correctly amended. The naming has no effect on $sol(\mathcal{P})$ and $sol(\mathcal{Q})$ because both of them concern with values' labels rather than the values themselves.

Proposition 5 provides a concrete procedure for merging any pair of values, with no restriction whatsoever. The trade-off comes in the form of spurious renditions that do not agree with the original network. Nevertheless, with this kind of value-merging it is possible to solve a constraint network indirectly as follows. First, transform the original network $\mathcal{P}$ by merging values and updating the network according to the method described by Proposition 5. Second, solve the resulting network to obtain a solution t. (If it is unsatisfiable, so is the original.) We call this stage the *solving stage.* Third, find a rendition of t that agrees with the original network. For convenience, we also say t is to be *interpreted* (into $\mathcal{P}$) and call this stage the *interpretation* stage. If the interpretation fails, we go back to the second stage and find another solution to interpret, repeating the second and the third stage until either the interpretation succeeds or no more solution is found.

Because Proposition 5 imposes no restriction on the values to be merged, nothing prevents us from repeatedly merging every pair of values until all domains become singletons (e.g. merging everything in $\mathcal{P}$ from Example 3 results in $\mathcal{Q}$.) In this instance, the solving stage is instantaneous (because there is only one solution, provided no domain in the original network is empty) while interpreting that solution is essentially the same as solving the original network.

We now consider what happens when only VI values are allowed to be merged.

Proposition 6 *If network $\mathcal{P}$ is transformed into $\mathcal{Q}$ by VI-merging then $\mathcal{P}$ subsumes $\mathcal{Q}$ and interpreting a solution of $\mathcal{Q}$ into $\mathcal{P}$ is search-free.*

Sketch of Proof: Like the proof of Proposition 5(1), it suffices to focus on the star-graph $\mathcal{S}$ and $\mathcal{S}'$ in which $a, b \in D(X)$ are merged into new value c. Let us now consider $s' \in rel(\mathcal{S}')$ such that $s'[X] = c$. We assume a and b are VI and not NI (otherwise the proposition is trivial), which means there exists variable Y where the supports of a and b are different. Let s be the tuple obtained from s' by replacing c with either a or b such that the replacement is compatible with $s'[Y]$. The definition of VI promises that such s is a solution of $\mathcal{S}$ and that testing a and b can be done in linear time through conventional constraint checks. Because s' is any arbitrary solution of $\mathcal{S}'$ and the labels of c come strictly from those of a and b , it follows that $sol(\mathcal{S}) \supseteq sol(\mathcal{S}')$ and thus $sol(\mathcal{P}) \supseteq sol(\mathcal{Q})$. Moreover, $lab(s[X])$ contains no invalid label. □

Proposition 4, 5, and 6 together tell us that if $\mathcal{P}$ is VI-merged into $\mathcal{Q}$ then $\mathcal{P}$ is satisfiable iff $\mathcal{P}$ is satisfiable. In addition, we now learn that VI-merging produces synonymous networks in a way that allows us to efficiently reconstruct solutions of the original networks from those of the transformed networks. The following example shows how.

Example 4 *Consider $D(X)$ of the network in Figure 5.3 (top). Value labeled 0 is VI with value labeled 1, while value labeled 2 is VI with value labeled 3. The network at the bottom is derived from the top by merging 0 with 1 and 2 with 3.*

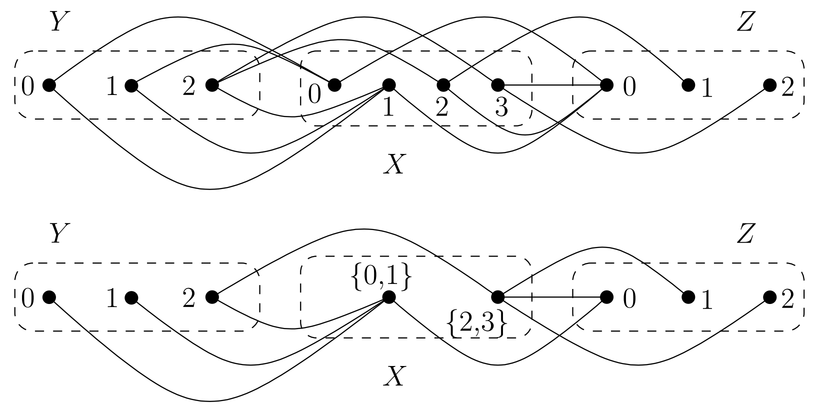


Figure 5.3: An example of VI-merging.

As a result, both networks are synonymous. To interpret solutions of the bottom network efficiently, it is best to make inferences rather than performing generate-and-test. For instance, consider solution $t = (2, \{0, 1\}, 0)$ *of the bottom network. To find valid labels of* $t[X]$*, we consider the common supports of* $t[Y] = 2$ *and* $t[Z] = 0$ *on* X *in the top network. We take the supports of 2 and 0 in* $D(X)$ *({1,2,3} and {0,1,2,3}) and compare them with the labels of* $t[X]$ *to filter out invalid labels. As a result, the valid label set of* $t[X]$ *is* $\{1, 2, 3\} \cap \{0, 1, 2, 3\} \cap \{0, 1\} = \{1\}$.

We can make a constraint network more compact by repeatedly merging VI values where possible. This involves going through each variable one by one and merging all VI values in the domain, then propagating the results to adjacent variables. The process terminates when no more VI values are detected. Propagation is necessary because values that are not initially VI may become so, once their neighbors are modified. Consider Example 3 for instance. In $\mathcal{Q}$ each domain has only one value, which is the best compression possible. Initially however, only the two variables at both ends (X_1 and X_n) can be compressed. No VI values are detected in the middle variables. After X_1 and X_n are compressed, the adjacent variables X_2 and X_{n-1} have to be re-examined in light of the change. Values of $D(X_2)$ and $D(X_{n-1})$ then all become VI and will be merged as a result. The process continues until the propagation converges.

It is important to note that merging all VI values of a variable domain with respect to different constraints produces different results. In fact, it was shown in [CDE15] that finding an optimal sequence of VI-merging operation is NP-complete. Because a variable may contain different VI values that are VI with respect to different constraints, to reduce the size of the search space one heuristic is to pick some ordering so that the domain size becomes as small as

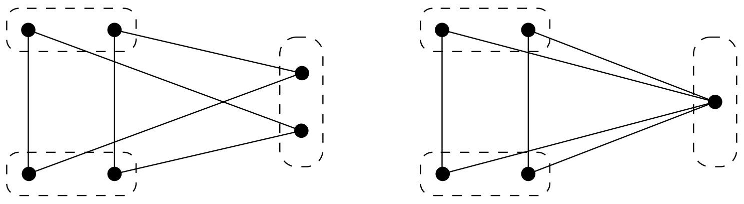


Figure 5.4: The network on the left is VI-free but does not satisfy the BTP, while the one on the right satisfies the BTP but is not VI-free.

possible when no VI values can no longer be found. In this chapter, we consider only this greedy heuristic. Given a domain, we detect VI values by considering each involved constraint one by one and calculate the possible reduction in domain size. The constraint that gives the best reduction is chosen first for the application of Proposition 5. The reduction for each constraint is re-computed and the process is repeated until all VI values for this variable are merged.

The broken-triangle property (BTP) [CJS10] has a similar condition to VI in that it also forbids values having different sets of supports in two adjacent variables. Figure 5.4 shows that the BTP is not comparable to the VI-free property.

5.6 Algorithms for Merging VI values

We now present algorithms for finding and merging VI values of a single variable domain. Given a variable, all involving constraints are initially joined together into a single table constraint. The result is then compressed by considering each constraint in turn whether values are VI with regard to this constraint. We consider only variables involving binary constraints. Later, we describe how to compress non-binary constraints.

Given variable X, Algorithm 12 collects all supports of values in $D(X)$ and tabulates the results. Each row represents supports of a single value in $D(X)$ from various constraints, and each cell contains the supports' labels, in effect making the row, a cartesian product representation (CPR). The table can be viewed as partially compressed from the start.

Optionally, after supports are collected the table can be enumerated first so that each cell contains only a single label and each row becomes a simple tuple rather than a CPR. This may yield a better compression later although the side-effect on involving constraints can be more extensive. More importantly, however, Proposition 5 do not apply since both handle only the merging of domain values, not splitting them. Splitting and recombining domain values are beyond the scope of this chapter, but it suffices to say that the resulting network does not necessarily subsume the original and thus some renditions may be lost, although conformity and satisfiability are unaffected. A case of missing renditions will be given in Example 5.

Algorithm 12: Collect(X)

```
1 $c \leftarrow |C \in \mathcal{C}$ such that $X \in scp(C)|$
2 let $T$ be an empty table of $c$ columns
3 foreach $C_j$ such that $X \in scp(C_j)$ do
4     foreach $t \in rel(C_j)$ s.t. $scp(C_j) = \{X, Y\}$ for some $Y$ do
5         add $lab(t[Y])$ to cell $(t[X], j)$ of $T$
```

Algorithm 13 details the greedy compression process. To compress a table, we hypothetically evaluate whether values are VI with respect to each column. The actual compression is performed with regard to the column that yields the best reduction (stopping at this point is denoted as the *single-best* compression process). The process is repeated until every column is committed . Constraints are updated using the finished table as the final step.

Proposition 7 *After all VI values are merged with respect to a certain constraint, any further merging of VI values in this domain with respect to the same constraint is not possible so long as neighboring domains are not altered.*

Proof: At time t_1 after all VI values are merged with respect to constraint C, no two values are NI with respect to all constraints except C. If at some later time t_2, there exist two values that are NI in all but C, these two values must differ in at least one other constraint C' beside C. Consequently, the difference in C' must be eliminated at some point between t_1 and t_2. The change in C' affects the neighboring domains, contradicting the assumption. □

Theorem 9 *The cost of* Compress*(C) is $O(m^3 n \lg n)$, where C is a table constraint of m columns and n rows.*

Proof: Line 5 takes $O(n)$. It takes $O(mn \lg n)$ in line 6 to sort the projection from line 5. After sorting, counting the duplicate in line 7 is just a matter of going through each member of A while comparing the current and the previous member, costing $O(mn)$. The whole process then takes $O(\sum_{i=1}^{m} imn \lg n) = O(m^3 n \lg n)$. □

Algorithm 13: COMPRESS(C)

```
1  remainder ← $scp(C)$
2  **repeat**
3      (var,max) ← $(\emptyset, 0)$
4      **foreach** $X \in$ *remainder* **do**
5          $A \leftarrow \pi_{scp(C)\setminus\{X\}}(C)$
6          sort $A$
7          count ← the number of duplicate elements in $A$
8          **if** *count* > *max* **then**  (var,max) ← ($X$, count)
9      **if** *max* > *0* **then**
10         remove var from remainder
11         merge *lab* of any two rows that differ only at col. var
12 **until** *max* = *0* **or** *rem* = $\emptyset$
```

Example 5 *We consider merging VI values of X in the network in Figure 5.3 (top). After supports are collected, the table is shown in Table 5.1(a). Tentative reduction size with respect to C_{XY} and C_{XZ} are both 2. The table is then compressed by merging VI values with respect to C_{XY}. The result is shown in Table 5.1(b). Next, VI values with respect to C_{XZ} are combined, resulting in Table 5.1(c). No more VI values exist and this table will be used to update C_{XY} and C_{XZ}, as depicted in Figure 5.3 (bottom). There are two values in the new $D(X)$: one with labels $\{0, 1\}$, and the other with $\{2, 3\}$.*

Alternatively, one can choose to expand Table 5.1(a) first so that it would contain 9 tuples, each component of each tuple having a single label. After compressing the table with respect to C_{XY} and C_{XZ} the result is shown in Table 5.1(d).

The original network has 9 rendered solutions whereas the compressed network in Table 5.1(c) has 2 solutions and 12 rendered solutions total, and the one in Table 5.1(d) has 2 solutions and 11 rendered solutions total. The network in Table 5.1(c) subsumes and conforms to the original. The network in Table 5.1(d) conforms to but does not subsume the original network. The tuple $t = (2, 2, 0)$, where $scp(t) = (X, Y, Z)$, exists in the original network but not in Table 5.1(d).

In addition to merging VI values in binary CSPs as discussed above, we can also use Algorithm 13 to compress tables in non-binary networks directly since we can think of a table as a dual variable in the hidden transformation [BCvBW02], where each tuple in the table stands for a value in the dual variable's domain. There is no need to invoke Algorithm 12.

Table 5.1: Joined tables for constraints involving X.

(a)

X	Y	Z
0	{0,1}	0
1	{0,1,2}	0
2	2	{0,1}
3	2	{0,2}

(b)

X	Y	Z
{0,1}	{0,1,2}	0
2	2	{0,1}
3	2	{0,2}

(c)

X	Y	Z
{0,1}	{0,1,2}	0
{2,3}	2	{0,1,2}

(d)

X	Y	Z
{0,1}	{0,1,2}	0
{2,3}	2	{1,2}

Proposition 8 *Given a non-binary constraint network $\mathcal{P}$ where each value has a single label initially. Assume the network is converted into a binary network using the hidden transformation. Assume further that we merge all possible VI values in every variable domain, including dual and ordinary variables, and propagate as necessary, resulting in network $\mathcal{Q}$. Interpreting a solution of $\mathcal{Q}$ into $\mathcal{P}$ is search-free.*

Proof: Because dual variables do not exist in the original network, there is no need to deal with their labels. Instead, instantiation of the dual variables determines labels of the ordinary ones. Note that this reasoning is not valid for a mixed binary and non-binary network since there would be no intermediary dual variable between two original variables if there exists a binary constraint involving them. □

Example 6 *Consider the constraint in Table 5.2(a). Let us consider merging VI values of H, the hidden variable for this constraint. Tentative reduction size with respect to each constraint are: $C_{HX} = 3$, $C_{HY} = 2$, $C_{HZ} = 1$, $C_{HW} = 0$. Since the maximum reduction size is 3, we then proceed with merging VI values with respect to C_{HX}. Table 5.2(b) shows the result. At this point, we recompute the reduction size for each remaining constraint: $C_{HY} = 1$, $C_{HZ} = 0$, $C_{HW} = 0$. Table 5.2(c) shows the result after further merging with respect to C_{HY}. No more compression is possible. Since instantiation of hidden variables has no effect on actual solution, there is no need to have multiple labels for values in H. We can re-label aceg, bf and d to 1, 2, and 3, for instance. Table 5.2(c) will be used to construct the hidden constraints C_{HX}, C_{HY}, C_{HZ}, and C_{HW}.*

We note that although one can hypothetically view a table as a dual variable,

Table 5.2: Merging of a 4-ary constraint. Labels in the H column correspond to domain values of the hidden variable.

(a)

H	X	Y	Z	W
a	0	0	1	1
b	0	1	0	0
c	0	1	1	1
d	1	0	0	1
e	1	0	1	1
f	1	1	0	0
g	1	1	1	1

(b)

H	X	Y	Z	W
ae	{0,1}	0	1	1
bf	{0,1}	1	0	0
cg	{0,1}	1	1	1
d	1	0	0	1

(c)

H	X	Y	Z	W
$aceg$	{0,1}	{0,1}	1	1
bf	{0,1}	1	0	0
d	1	0	0	1

but without actual dual variables present, the structural information of the combined tuples is lost. The CPR thus represents both syntax and semantic of the constraints at the same time. Consistency algorithms that normally work on tuples therefore must be modified to handle the CPR. On the other hand, the network that is explicitly transformed by the hidden transformation and compressed afterwards does not require any specialized algorithm.

5.6.1 Experimental Results

We now present some preliminary results on domain and table compression. We first compare results on random CSPs, which we expect to be unstructured, followed by structured problems. Results for series generated according to model RB are shown in Table 5.3. The processing time for each instance is negligible — a small fraction of a second at the most. Enumerating each table before the compression yields almost no improvement. Despite the fact that only a few percentage of all values in a single instance are found to be VI, we do not find even a single pair of NI values in any instance tested. Interestingly, the number of affected variables is significant though the reduction is small, i.e. up to 27% of variables but with only 1.63% values merged. Since detecting NI values dynamically [Has93, LCF05] can improve their numbers, we can expect even more improvement for VI.

NS is not considered here because finding all NS values is too expensive and only two values can be checked at the same time. The number of pairs grows exponentially along with domain size. By contrast, algorithms for NI and VI can tackle the whole domain at once in polynomial time.

Table 5.4 displays the results for crossword puzzle `ukVG`[2], which involves non-binary constraints of non-random structure: a constraint of arity k contains all the words of length k. In contrast to the randomly generated binary problems, we see that compressing all VI values yields remarkable reduction rates, as high as 70% for the arity-4 constraint. At low arity, there is not much difference between the single-best compression and the greedy compression, but as the arity increases the reduction rate for the single-best drops sharply while the greedy compression maintains its rate well. The greedy compression can merge twice more values than the single-best compression at high arity.

We have also conducted some experiments to measure the running time of compressed instances from the series `rand-3-20-20` and `rand-5-30-5` using Abscon[3] as a blackbox solver. Because newer solvers such as Abscon have implemented advanced GAC algorithms, we expect them to perform well on table constraints and relatively poorly on the hidden-transformed binary encoding, which require more variables, more constraints, and much larger domains. We therefore compare the running time of the original problems against their hidden transformation problems that are simplified by the compression. The reduction rate ranges from 60% to 86%. Overall, the running time of the transformed problems is very competitive with that of the original — winning over half of all the instances from `rand-3-20-20` and losing slightly on `rand-5-30-5` for most instances — despite the obvious disadvantages of the hidden transformation. These results however are limited by the fact that the solver is not modified to take advantage of the CPR of the compressed tables or the topology of the hidden-transformed networks [SS05]. With either of them taken into account, the VI compression is likely to boost solver's performance much further in practice.

[2]All benchmarks are from `http://www.cril.univ-artois.fr/CSC09`.

[3]From `http://www.cril.univ-artois.fr/~lecoutre/software.html`.

Table 5.3: Results for random problems. The results for each series are averaged over 100 instances. #v is the no. of variables in each instance, #d the domain size, #c the no. of constraints, #t the tightness of each constraint, #vi the percentage of values merged via VI, and #va is the percentage of variable affected ($\geq$ 2 VI values merged.)

Instance	#v	#d	#c	#t	#vi	#va
2-10-5-15-658	10	5	15	0.658	2.58%	10.90%
2-20-3-30-519	20	3	30	0.519	4.38%	12.35%
2-20-20-25-909	20	20	250	0.909	1.58%	18.60%
2-50-3-120-367	50	3	120	0.367	2.54%	6.96%
2-50-5-70-683	50	5	70	0.683	2.59%	11.08%
2-100-4-200-500	100	4	200	0.500	2.05%	7.36%
2-100-10-110-877	100	10	110	0.877	1.95%	14.96%
2-200-100-220-985	200	100	220	0.985	1.63%	27.20%

Table 5.4: Results for crossword puzzles. Column #tuple gives the original no. of tuples. The third col. shows the no. of tuples left after the single-best compression, while the fourth displays the reduction percentage. `maxrate` gives the reduction rate for the greedy compression.

arity	#tuples	#tuples left	reduction rate	maxrate
4	4947	1727	65.09%	70.91%
5	10935	6203	43.27%	54.60%
6	18806	13759	26.84%	41.34%
7	27087	22156	18.20%	32.93%
8	32387	29558	8.73%	23.84%
9	32865	30420	7.44%	18.09%
10	29784	27699	7.00%	14.48%
11	23333	21897	6.15%	12.78%
12	16917	15944	5.75%	11.67%
13	11246	10749	4.42%	10.33%
14	6998	6745	3.62%	10.02%
15	3962	3839	3.10%	9.36%
16	2009	1962	2.34%	7.72%

5.7 Identifying Redundant Values

We turn our attention to onto-substitutability in this section. An onto-substitutable value is redundant in the sense that every solution it participates in is also covered by some other value. We consider only algorithms that compute all onto-substitutable values in the smallest closure of a given variable in a network with extensional constraints.

To remove all onto-substitutable values from the domain of variable X, we do

Table 5.5: Tables for constraints involving A.

(a)

A	B	C
4	1	1
0	0	0
0	1	1
1	1	1
2	0	0
3	0	0
5	1	1

(b)

A	D	E
0	2	2
0	3	3
2	2	2
2	3	3
1	0	1
3	3	3
4	2	2
5	2	2
5	3	3
5	0	1

(c)

A	B	C	D	E
{0,2}	0	0	2	2
{0,2,3}	0	0	3	3
{1,5}	1	1	0	1
{0,4,5}	1	1	2	2
{0,5}	1	1	3	3

(d)

A	B	C
{0,2,3}	0	0
{0,1,4,5}	1	1

(e)

A	D	E
{1}	0	1
{0,2,4,5}	2	2
{0,2,3,5}	3	3

the following,

1. let D be the table created as a result of joining of all table constraints C involving X.
2. sort D in lexicographic order while ignoring column X.
3. merge cells of any two rows that differ only at col. X.
4. while there exists a value $v \in D(X)$ such that every cell containing v also contains another value v', remove v from these cells and from $D(X)$.

An example is demonstrated in Table 5.5. Variable A is involved in two ternary constraints given as positive tables ((a) and (b)). After the two tables are joined and rows are merged, the result is shown in (c). Let us first consider value 0. In column A, 0 is never contained in a singleton set. Therefore 0 is onto-substitutable and we can remove it from the $D(A)$ and from all the sets in column A. Value 1 does not appear as a singleton as well; hence 1 is onto-substitutable. The set containing value 2 in the first row has become a singleton after the removal of 0, so 2 is not onto-substitutable. So are values 3 and 4 for the same reason. Therefore, values 0, 1, 3, and 4 are onto-substitutable and they can be removed from $D(A)$.

Note that the concept of onto-substitutability is tied to the structure of the network, therefore algorithms may produce different outcomes depending on the sequence of value removal. For instance, suppose value 5 is considered first, rather than value 0 as done in the previous example. Value 5 is onto-substitutable and it is removed. Next, values 2, 3, and 4 are examined and found to be onto-substitutable and removed as well. As a result, values 5, 2, 3, and 4 are found to be onto-substitutable in that order.

While this algorithm is simple and straightforward it has serious inefficiency in the process of joining of the constraints involved, which amounts to joining every constraint in a complete-graph constraint network for instance. We will consider an improved algorithm which places more computation on each table and operates on joined sub-tables only when necessary. Details are given in Algorithm 14.

We assume tables are initially merged as done in Table 5.5 (d) and (e) (from (a) and (b)). The algorithm decides whether $a \in D(X)$ is onto-substitutable. It first checks if there exist values that do not appear in all tables (line 2). These values cannot be joined and are filtered out. This step is equivalent to enforcing maxRPWC [LPS13]. The next step tests whether it is possible for a to be subsumed. Values that appear as singletons can never be onto-substitutable (line 6). This step has been employed in the basic algorithm. The algorithm may take this step first but it has to do it again after the maxRPWC filtering because values that are not singleton initially may become so later.

The algorithm then calculates the number of tuples involving value a (*goal*). This can be done by simple multiplication without actually joining the tuples across tables. The upper bound (*ub*) is the maximum number of tuples that are shared by other values. Again, this can be easily computed and if the number is lower then the value a can never be subsumed and the algorithm can terminate and give a negative answer. Otherwise, the algorithm will try to enumerate all the tuples that share a's structure and see whether a can be completely covered (line 13) via the set *store*. Alternatively, this stage can be implemented more compactly using tuple sequence [RÍ11] marked by lower and upper bounds, instead of storing the actual tuples themselves.

Example 7 *Consider determining whether* $0 \in D(A)$ *from Table 5.5 (a) and (b) is onto-substitutable. The algorithm will check these four sets {0,2,3}, {0,1,4,5}, {0,2,4,5}, and {0,2,3,5} (the tuple involving {1} has nothing to do with the onto-substitutability of 0). First, value 1 is removed from {0,1,4,5}* *because there would be no tuple involving 1 after joining the two tables. There is no singleton containing 0 so the algorithm passes through the cutoff test (line 6). Next, goal is calculated to be 4, while ub is the size of the CPR involving value 2 (2) + value 3's (1) + value 4's (1) + value 5's (2) = 6. The algorithm also passes through the cutoff test (should the original two tables contain no tuple involving 2 and 5 the values of ub would be 2, for instance, and false would be returned in this case). The algorithm then adds the following tuples sequentially to store: (0,0,2,2), (0,0,3,3), (1,1,2,2), (1,1,3,3). At this point store's size is equal to goal so a is proved to be onto-substitutable.*

Algorithm 14: is-onto-substitutable(X, a)

```
1  pick some constraint $C'$ such that $X \in scp(C')$
2  foreach tuple $t \in rel(C')$ such that $a \in t[X]$ do
3      foreach $b \in t[X]$ such that $b \neq a$ do
4          if $b \notin t'[X]$ in each $C'' \neq C'$ for some $t'$ then
5              remove $b$ from $t[X]$
6  foreach $C$ such that $X \in scp(C)$ do
7      if $|t[X]| = 1$ for some $t \in rel(C)$ then
8          return false
9  $goal \leftarrow \sum_t |t|$, $a \in t[x]$ and $t$ is joined across all constraints
10 $ub \leftarrow \sum_t |t|$, $a \notin t[x]$ and $t$ is joined across all constraints
11 if $ub < goal$ then return false
12 $store \leftarrow \emptyset$
13 foreach tuple $t \in rel(C)$ such that $a \in t[X]$ do
14     foreach $b \in t[X]$ such that $b \neq a$ do
15         $u \leftarrow$ CPR joined across all constraints s.t. $b \in u[X]$
16         foreach $t'$ enumerated from $u$ do
17             store $\leftarrow$ store $\cup$ $\pi_{scp(C)\setminus\{X\}}(t)$
18         if $|store| = goal$ then
19             return true
20 return false
```

5.8 Conclusions

Onto-substitutability has been shown as a sufficient condition for a value to be treated as redundant. We extend its definition from unary to binary relation and introduce JI as a replacement for interchangeability. As a symmetric binary relation, JI allows us to remove either one of the interchangeable set of values, giving us more flexibility as a result. We then present an original strategy for dealing with redundant values: detection, creation, followed by elimination.

Since JI is a binary relation, this strategy makes sense: we can identify a group of values, create a JI-equivalent using fewer number of values, and then delete the original values.

The definition of values is expanded to include the concept of labels, which allows us to tease out semantics from the structure of the network. Different CSPs can be compared solely on their semantics (via their rendered solution). We then introduce VI, a new local reasoning that leads to JI. While it remains to be seen whether future work in this area will give us a new local property that also leads to JI, we have empirically shown the promise of VI as a compression tool. Table constraints can be compressed using other techniques such as decision trees [KW07], but they require specialized consistency algorithms unlike VI. VI may prove useful in different situations as well, for instance, VI could be used in addition to NS as a simplification operation before reasoning with CSP patterns [CE12].

In [BCM08], the authors raise “an interesting open issue: do there exist new (i.e., other than substitutability and inconsistency) properties for which local reasoning is sound and which imply removability?” We believe VI is one such property, provided that creation of new values is permitted. Allowing networks to be augmented in this manner could also lead to a more powerful general framework.

Chapter 6

Improving the Lower Bound of Simple Tabular Reduction

6.1 Abstract

Simple Tabular Reduction (STR) is a state-of-the-art filtering technique for enforcing Generalized Arc Consistency (GAC) on positive table constraints. Despite its good performance in practice, the STR2 algorithm suffers from a mandatory lower bound equal to the number of domain values in the current state of the problem, because the presence of each value must be justified every time STR2 is called. We overcome this fixed cost by redesigning STR2 and incorporating watched tuples. Experiments show that the revised algorithm is better at coping with problems that involve a large number of small changes during search and it can be more than twice as fast on certain structured problems.

6.2 Multistage Simple Tabular Reduction

For conciseness, the reader is assumed to have familiarity with STR and STR2 (details can be found in [Ull07, Lec11]). Pseudo-code of STR2 is reproduced in Algorithm 15 (minor details are omitted). For brevity, we use $\tau(c, i)$ to denote the tuple *table*$[c][$*position*$[c][i]]$ (*position*$[c]$ provides indirect access to the tuples of *table*$[c]$). STR/STR2's key concepts are summarized as follows. In essence, operations in STR center around individual tuples in turn. Each tuple is first

Algorithm 15: STR2(c: constraint)

```
$S^{val} \leftarrow \emptyset$
for $x \in scp(c)$ such that $|dom(x)| \neq lastSize[c][x]$ do   // Has $dom(x)$ changed?
    $S^{val} \leftarrow S^{val} \cup \{x\}$
    $lastSize[c][x] \leftarrow |dom(x)|$
$S^{sup} \leftarrow \emptyset$
foreach $x \in scp(c)$ such that $|dom(x)| > 1$ do
    $S^{sup} \leftarrow S^{sup} \cup \{x\}$
    $gacValues[x] \leftarrow \emptyset$            // values of $x$ having supports, initially $\emptyset$
$i \leftarrow 1$                                                      // Main loop
while $i \leq limit[c]$ do    // table size is dynamically adjusted up to $limit[c]$
    if isValidTuple $(c, S^{val}, \tau(c,i))$ then
        foreach $x \in S^{sup}$ do
            if $\tau(c,i)[x] \notin gacValues[x]$ then
                $gacValues[x] \leftarrow gacValues[x] \cup \{\tau(c,i)[x]\}$
                if $|gacValues[x]| = |dom(x)|$ then  $S^{sup} \leftarrow S^{sup} \setminus \{x\}$
        $i \leftarrow i + 1$
    else
        swapTuple $(c, i, limit[c])$    // shrink table, removing invalid tuple
        $limit[c] \leftarrow limit[c] - 1$
if $limit[c] = 0$ then return Failure
if $S^{sup} = \emptyset$ then return DomUnchanged
foreach $x \in S^{sup}$ do
    $dom(x) \leftarrow gacValue[x]$
    $lastSize[c][x] \leftarrow |dom(x)|$
return DomChanged
```

scanned to see if it passes the validity check (i.e., if the values of the tuple are compatible with the current domains). A tuple is removed if it is invalid. Otherwise, it is scanned through again, this time to collect values and mark them as being supported. The process repeats until every non-removed tuple is dealt with. STR2 improves upon STR in two ways. First, during the validity check, if the domain size of a variable x has not been changed since the last time the algorithm was called, then the validity of $\tau[x]$ (the projection of τ on x) remains unchanged for any tuple τ, and consequently, the entire column x of the table can be skipped. This is achieved through the use of a set called S^{val}. Similarly, when collecting, if the whole domain of x has already been proved to have supports then the entire column x of the table can be skipped over using set S^{sup} (a variation of this is used in [CY10]).

STR2 is forgetful and must rebuild a list of supported values from scratch every time. For this reason, STR2 suffers when the impact of propagation on the table is too small to warrant the buildup cost. We introduce STR2w (Alg. 16),

Algorithm 16: STR2w(c: constraint)

```
$S^{val} \leftarrow \emptyset$
foreach $x \in scp(c)$ such that $|dom(x)| \neq lastSize[c][x]$ do
    $S^{val} \leftarrow S^{val} \cup \{x\}$
    $lastSize[c][x] \leftarrow |dom(x)|$
$prevLimit \leftarrow limit[c]$
tuple-elimination($c$)                                    // Phase 1
if $limit[c] = 0$ then return Failure
support-deduction($c, prevLimit$)                         // Phase 2
if $S^{sup} = \emptyset$ then return DomUnchanged
value-accumulation($c$)                                   // Phase 3
if $S^{sup} = \emptyset$ then return DomUnchanged
foreach $x \in S^{sup}$ do
    foreach $a \in unsupported[x]$ do
        $watch(prevWatch(x,a)) \leftarrow watch(prevWatch(x,a)) \cup \{(x,a)\}$
    $dom(x) \leftarrow dom(x) \setminus unsupported[x]$
    $lastSize[c][x] \leftarrow |dom(x)|$
return DomChanged
```

Algorithm 17: tuple-elimination(c: constraint)

```
$i \leftarrow 1$
while $i \leq limit[c]$ do
    if $\neg$isValidTuple $(c, S^{val}, \tau(c,i))$ then
        while $i < limit[c]$ and $\neg$isValidTuple $(c, S^{val}, \tau(c, limit[c]))$ do
            $limit[c] \leftarrow limit[c] - 1$
        if $i < limit[c]$ then
            swapTuple $(c, i, limit[c])$
            $limit[c] \leftarrow limit[c] - 1$
    $i \leftarrow i + 1$
```

a redesign of STR2 that avoids the complexity bound on this step by rearranging STR2 into three inter-dependent phases.

The first phase eliminates invalid tuples from the table (`tuple-elimination` in Algorithm 17). The second phase deals with the actions from removing these tuples (`support-deduction` in Algorithm 18). Finally, the remaining tuples are traversed in the third phase (`value-accumulation` in Algorithm 19) and the encountered values are collected indirectly by discarding them from sets called *unsupported*. The values that remain in these sets are recognized to have no supports and can be removed from their respective domains.

Algorithm 18: support-deduction(c: constraint, $prevLimit$: integer)

```
$S^{sup} \leftarrow \emptyset$
foreach $x \in scp(c)$ such that $|dom(x)| > 1$ do
    $unsupported[x] \leftarrow \emptyset$
$i \leftarrow limit[c] + 1$
while $i \leq prevLimit$ do
    if $watch(\tau(c,i)) \neq \emptyset$ then
        foreach $(x,a) \in watch(\tau(c,i))$ such that $|dom(x)| > 1$ and $a \in dom(x)$ do
            $S^{sup} \leftarrow S^{sup} \cup \{x\}$
            $unsupported[x] \leftarrow unsupported[x] \cup \{a\}$
            $watch(\tau(c,i)) \leftarrow watch(\tau(c,i)) \setminus \{(x,a)\}$
            $prevWatch(x,a) \leftarrow \tau(c,i)$
    $i \leftarrow i + 1$
```

Algorithm 19: value-accumulation(c: constraint)

```
$i \leftarrow 1$
while $i \leq limit[c]$ and $S^{sup} \neq \emptyset$ do
    foreach $x \in S^{sup}$ do
        if $\tau(c,i)[x] \in unsupported[x]$ then
            $unsupported[x] \leftarrow unsupported[x] \setminus \{\tau(c,i)[x]\}$
            $watch(\tau(c,i)) \leftarrow watch(\tau(c,i)) \cup \{(x, \tau(c,i)[x])\}$
            if $unsupported[x] = \emptyset$ then
                $S^{sup} \leftarrow S^{sup} \setminus \{x\}$
    $i \leftarrow i + 1$
```

STR2w recasts STR2 as a repair-based algorithm. Each tuple is associated with a list of values that depend on this tuple as a proof of support. In other words, each tuple can be "watched" by several values (its "watchers"). Should a tuple be removed, its watchers must be reattached, if possible, to another valid tuple. During the second phase, the list of detached watchers is thereby collected and delivered to the third phase, whereupon new watched tuples must be found.

Pseudo-code of STR2w is given in Algorithms 16–20 where Algorithm 16 is the main function, to be called whenever GAC must be reinforced. The monolithic structure of STR2 (the while loop in line 10 of Algorithm 15) is broken down into three rounds of table traversals: tuple-elimination (line 6), support-deduction (line 8) and value-accumulation (line 10). These three sub-routines are explained in more detail below.

The elimination of tuples in Algorithm 17 is like STR2's with an improvement. The algorithm tries to enlarge the valid partition from top-to-bottom as well as to enlarge the invalid partition from bottom-to-top. To achieve that, we handle two pointers i and $limit[c]$. When both pointers can no longer be moved and i is still smaller than $limit[c]$, we know that swapping them enlarges both partitions at the same time (lines 7–8). By contrast, the swap in STR2 is half as efficient since only the invalid partition on the bottom is guaranteed to expand while nothing can be said about the tuple from the bottom just swapped in. In the worst case, where enforcing GAC causes wipeout, STR2 has to do the swapping for every (invalid) tuple whereas STR2w only moves the pointer $limit[c]$.

The support-deduction routine in Algorithm 18 determines which current supports do not remain valid. Two data structures are involved. The first one associates with each tuple τ of the table a list $watch(\tau)$ that contains all values $(x, \tau[x])$ such that $\tau[x]$ depends on tuple τ as a support. Therefore $0 \leq |watch(\tau)| \leq r$ for any τ where r is the arity of the constraint. The second associates with each variable x a list $unsupported[x]$ that contains all values in the domain of x that are currently not supported. Domain values are presumed to be supported at the beginning (line 3), and for any τ just removed, its watchers are marked as currently unsupported (line 9). Note that $prevWatch(x, a)$ records the last watched tuple of (x, a) so it can be reinstated later when a is found to have no other supports.

Value accumulation in Algorithm 19 differs from STR2 in that it deals with a list of unsupported rather than supported values, and reattaching values to new watched tuples (line 6). In STR2, the list of supported values is built up from scratch, so the complexity cost is fixed at $O(rd)$, where d is the greatest domain size. Conversely, the number of unsupported values at the beginning of this phase of STR2w ranges from zero to rd. In the best case, where the previous phase reports no unsupported value (i.e., S^{sup} is empty), STR2w can terminate early (line 9 of Algorithm 16). In the worst case, where the list of unsupported values is full, STR2w reduces to STR2. After this phase is finished, any value remained in *unsupported* can be removed (line 15 of Algorithm 16) and its previous *watch* restored (line 14 of Algorithm 16).

Algorithm 20 initializes STR2w's data structures and is executed only once, right after GAC preprocessing. Because STR2w requires tables to be free of invalid tuples before the search starts, *it cannot be used as a standalone GAC algorithm and must be used exclusively during search [LZS$^+$07, LLY12].* During the initialization, $unsupported[x]$ is set to be the whole domain of x (line 2), while *watch* is set to empty (line 3). Algorithm 19 is reused at the end.

Correctness of STR2w comes from the following two invariants. We take an

Algorithm 20: STR2w-initialization(c: constraint)

```
1 $S^{sup} \leftarrow scp(c)$
2 foreach $x \in scp(c)$ do  $unsupported[x] \leftarrow dom(x)$
3 foreach $tuple\ \tau$ in table constraint $c$ do  $watch(\tau) \leftarrow \emptyset$
4 value-accumulation ($c$)
```

invariant to be a property that remains true throughout the search at the points after STR2w is enforced, not during STR2w's execution where the filtering may not yet converge.

Invariant 1 *The collection $\mathcal{W} := \{watch(\tau) : \tau \in rel(c)\}$ represents a partition of $\mathcal{D} := \bigcup_{x \in scp(c)} dom(x)$ such that every element (x, a) of $\mathcal{D}$ is in one and only one set of $\mathcal{W}$.*

Invariant 2 *If $(x, a) \in watch(\tau)$, then τ is valid iff a is present in the domain of x (τ is invalid iff a is absent from the domain of x).*

It is important to remember that the network must be made GAC before the search begins. The two invariants trivially hold after *watch* is properly initialized in STR2w-initialization. During the execution of STR2w, one of the following three scenarios must occur to any individual watcher. First, a watcher may be unaffected throughout the search. The reason is that its watched tuple remains valid at all times. Second, a watcher is detached because its watched tuple becomes invalid. Another valid tuple is later found and the watcher is shifted to this tuple. Third, a watcher is detached from its watched tuple but no valid tuple is found, in which case the watcher is re-attached to the original tuple (line 14 of Algorithm 16). STR2w maintains the two invariants for any of these three cases and thus both invariants continue to hold when STR2w terminates.

Theorem 10 *The worst-case complexity of STR2w is the same as that of STR2.*

Central to the analysis of STR2w are the operations on *unsupported*, S^{sup}, and *watch*. Both *unsupported* and S^{sup} involve insertion, deletion, and membership tests. They cost $O(1)$ time using the sparse sets data structure [BT93, CY10]. The structure *watch* requires no membership test; its elements are accessed and deleted sequentially as well as being backtrack-stable. As a result, the complexity cost of STR2w is no more than that of STR2.

On the other hand, a table can be constructed so that STR2 has to go through

Table 6.1: Results from selected instances.

instance			STR2			STR2w			STR3
	bt	$w1$	$w2$	$w1/w2$	*time*	$w2$	*cutoff*	*time*	*time*
crossword-m1c-uk-vg7-7	25492	1533	604	2.54	13.44	253	7.04	**12.88**	41.45
crossword-m1c-uk-vg6-8	194006	574	227	2.53	43.50	107	7.84	**39.29**	246.48
crossword-m1c-uk-vg14-14	2657	13922	6222	2.24	14.57	1934	11.87	13.93	**6.99**
crossword-m1c-ogd-vg11-14	21513	76554	24969	3.07	1949.92	6206	18.08	1996.00	**502.02**
crossword-m1c-ogd-vg14-14	1004	54887	22154	2.48	29.14	5510	17.11	25.39	**6.86**
dubois-22	12.6m	6.03	2.77	2.17	46.97	1.58	4.55	**39.76**	172.03
dubois-23	25.2m	6.03	2.78	2.17	95.14	1.57	4.41	**83.10**	361.18
rand-3-20-20-60-632-1	28484	215	39	5.42	6.96	13	34.03	**6.40**	8.80
rand-3-20-20-60-632-2	60803	486	65	7.38	26.04	22	31.15	27.32	**25.56**
bdd-21-2713-15-79-1	986	4645	130	35.66	**31.57**	85	11.36	39.72	38.21
bdd-21-2713-15-79-2	970	4682	120	38.84	**29.59**	79	10.69	39.70	40.05
rand-10-60-20-30-p51200-0	6402	147234	3225	45.65	187.48	538	15.51	180.73	**21.41**
langford-3-11-ext.xml	14511	24	8.14	3.02	**15.36**	2.25	57.22	16.42	15.82
rand-8-20-5-18-800-1	298462	886	60	14.66	**158.15**	28	8.07	177.58	1022.16
rand-8-20-5-18-800-3	230790	583.54	41.1	14.19	**25.70**	20.6	8.76	25.93	768.9
inst-n21-k2713-a15-p79-i1	986	4645	130	35.66	**23.07**	86	11.36	29.37	37.36
inst-n21-k2713-a15-p79-i2	970	4682	121	38.84	**24.64**	80	10.69	32.47	39.49
ramsey-16-3	738081	35.46	8.67	4.09	40.77	1.78	46.80	**27.16**	131.50
renault-mod-4	860159	202.65	45.80	4.42	16.20	7.34	29.04	**10.81**	72.37

$O(rd)$ cells of the table before the algorithm concludes that every domain value has a support whereas in STR2w no watcher is detached during the support-deduction phase. Thus it follows that,

Property 1 *STR2w can save $O(rd)$ operations with respect to STR2 on some tables, where r is the arity of the table and d is the greatest domain size.*

6.3 Experimental Results

We tested STR [Ull07], STR2, STR2w, and STR3 [LLY12] with a solver written in C++. Experiments were conducted on a 2.6GHz quad-core Intel Core i7 on OS X 10.8. All algorithms use the *dom/ddeg* variable ordering heuristic [Lec11]. Selected results[1] are reported in Table 6.1, where *bt* is the number of backtracks, $w1$ is the average number per call of cells traversed during the tuple-elimination phase (this is the same for both algorithms), $w2$ is the average number of cells traversed during the value-accumulation phase, *time* is the running time in seconds, and *cutoff* is the percentage of calls to STR2w that results in S^{sup} being empty so that STR2w exits in line 9. STR2 is applied as a preprocessing step for STR2w and its running time is included in the results for STR2w. STR is slower than STR2 and STR2w on every instance tested (STR is at least 30% slower than STR2) so for succinctness results for STR are not shown in the table.

[1]Benchmarks in Table 6.1 are available at http://www.cril.univ-artois.fr/CSC09.

STR2w is competitive with STR2 according to Table 6.1. While STR2w has a much better lower bound (See Property 1), this theoretical advantage does not give substantial gains in Table 6.1. A closer inspection at the statistics reveals some interesting details. First, from the $w1/w2$ column, STR2 spends a high proportion of its effort on tuples-elimination, at least twice as much and often an order-of-magnitude more. It means that, the higher the ratio $w1/w2$ is, the lesser influence on the running time STR2w has, since STR2w basically attempts to reduce $w2$. Second, the absolute $w1$ and $w2$ counts are generally low. It turns out that the tabular reduction wipes out large number of invalid tuples early on, so that the search is left to deal with only small remnants of the original tables. This partly happens due to the fact that each variable instantiation eliminates all but a single value from the domain of a variable, which in turn may cause as large a reduction in the connected tables. Even though the theoretical lower bound of STR2w is better, STR2w is a slightly more complex algorithm that has a higher overhead associated with its specific operations. When $w1$ and $w2$ are low, which signifies that the average table size during search is small, this overhead may come to dominate the running time. The reason why optimizations in STR2w do not consistently pay off is due to many factors, such as small domain size, low STR2's $w2$ count, and high STR2's $w1/w2$ ratio. They are all present in the *bdd* series for example, where STR2w is noticeably slower; many instances in Table 6.1 share at least one of these factors. STR2w is designed to cope with frequent but small table reductions (i.e., "hard" problems for STR/STR2) so the fact that STR2w is competitive despite its overhead is good news. In this respect, STR2w is closer to STR3, which is path-optimal and has similar mechanism for dealing with values that lost supports. From the table, STR3 appears to be complementary to STR2w. The next experiment focuses on differentiating the three algorithms.

A pigeonhole problem ph-k consists of k variables, each with $\{0, \ldots, k-1\}$ domain, and any two variables are connected by a binary inequality constraint. Unlike most benchmarks, it allows a table to be reduced gradually, until a variable directly involved is instantiated. An augmented pigeonhole ph-k-j adds j new variables for each of the k variables, and chains them together with a j-ary table. The extra variables have larger domains to prevent the ordering heuristic from picking them prematurely. Results are shown in Table 6.2 where *sp* is the speedup factor (the running time of STR2 divided by that of STR2w). For pigeonhole (ph-k), STR2w is about 30% faster than STR2 while STR3 is twice as slow. The values of $w1$, $w2$, and $w1/w2$ are low, but uniform. On the

Table 6.2: Results from pigeonhole instances.

instance			STR2			STR2*w*			*sp*	STR3
	bt	*w*1	*w*2	*w*1/*w*2	*time*	*w*2	*cutoff*	*time*		
ph-9	40319	6.49	2.11	3.08	0.38	0.70	69.43	**0.27**	1.41	0.78
ph-10	362879	6.35	2.04	3.11	3.89	0.61	72.49	**2.75**	1.41	8.35
ph-11	3628799	6.23	1.99	3.14	42.92	0.55	74.93	**31.64**	1.36	99.66
ph-12	39916799	6.15	1.94	3.16	608.62	0.50	76.93	**389.05**	1.56	1349.79
ph-7-7	719	70999	66227	1.07	12.69	3541	52.63	5.56	2.28	**2.05**
ph-7-8	719	482728	454111	1.06	62.64	22124	52.63	30.37	2.06	**13.80**
ph-8-7	5039	152770	142487	1.07	164.30	5789	58.41	79.90	2.06	**37.28**
ph-8-8	5039	1211836	1139880	1.06	1259.44	40938	58.41	586.52	2.15	**322.04**
ph-9-5	40319	3404	3089	1.10	36.69	164	62.31	19.77	1.86	**9.97**
ph-9-6	40319	32133	29635	1.08	326.28	1315	62.31	161.88	2.02	**82.27**
ph-10-4	362879	444	391	1.14	55.18	26	65.79	30.37	1.82	**26.75**
ph-10-5	362879	4827	4381	1.10	529.77	231	65.79	276.18	1.92	**148.35**
ph-11-2	3628799	9.04	4.69	1.93	62.86	1.04	69.05	**43.33**	1.45	137.02
ph-11-3	3628799	48	38	1.26	121.57	3.91	69.05	**80.36**	1.51	164.40
ph-12-2	39916799	8.97	4.61	1.94	779.64	0.98	71.39	**505.36**	1.54	1764.72
ph-12-3	39916799	52	42	1.25	1670.17	4.15	71.39	**970.75**	1.72	2191.79

other hand, *cutoff* is high – in more than 70% of the calls, STR2w completely sidesteps the value-accumulation phase altogether. For augmented pigeonhole ph-k-j, the speedup factor is often over 2. The $w1/w2$ count is low, close to 1 in most cases, and STR2w reduces STR2's $w2$ count by more than one order-of-magnitude for almost every instance. Increasing j increases $w1$ without much change to $w1/w2$ thus increasing the speedup. STR3 is faster when the augmented tables are large.

6.4 Conclusions

GAC algorithms such as STR and mddc [CY10] are considered to be in the same class, in the sense that the whole of the constraint store must be traversed — be it table or mdd — whenever a perturbation occurs. The mddc algorithm was made incremental in [GSS11] to deal with this issue. Here, we show that STR2's value-accumulation phase can be made incremental as well, although this does not make STR2w optimal in the worst case (see [LLY12, MHD12] for optimal algorithms). Our experiments show that STR2w outperforms STR2 when solving problems in which a large number of small changes occur during search, whereas STR2 is better at problems where changes are big, while it is also complementary to STR3 at the same time. STR2 has now gained widespread acceptance as a de facto filtering algorithm for positive table constraints and it has been incorporated into other consistency algorithms [JN13, LPS13, XY13]. The techniques such as watched tuples and multistage STR may prove more effective in that context and should be considered as serious alternatives especially for problems where small changes are the norm.

Chapter 7

STR3: A Path-Optimal Filtering Algorithm for Table Constraints

7.1 Abstract

Constraint propagation is a key to the success of Constraint Programming (CP). The principle is that filtering algorithms associated with constraints are executed in sequence until quiescence is reached. Many such algorithms have been proposed over the years to enforce the property called Generalized Arc Consistency (GAC) on many types of constraints, including table constraints that are defined extensionally. Recent advances in GAC algorithms for extensional constraints rely on directly manipulating tables during search. This is the case with a simple approach called Simple Tabular Reduction (STR), which systematically maintains tables of constraints to their relevant lists of tuples. In particular, STR2, a refined STR variant is among the most efficient GAC algorithms for positive table constraints. In this chapter, we revisit this approach by proposing a new GAC algorithm called STR3 that is specifically designed to enforce GAC during backtrack search. By indexing tables and reasoning from deleted values, we show that STR3 can avoid systematically iterating over the full set of current tuples, contrary to STR2. An important property of STR3 is that it can completely avoid unnecessary traversal of tables, making it optimal along any path of the search tree. We also study a variant of STR3, based on an optimal circular way for traversing tables, and discuss the relationship of STR3 with two other optimal GAC algorithms introduced in the literature, namely, GAC4 and AC5TC-Tr. Finally, we

demonstrate experimentally how STR3 is competitive with the state-of-the-art. In particular, our extensive experiments show that STR3 is generally faster than STR2 when the average size of tables is not reduced too drastically during search, making STR3 complementary to STR2.

7.2 Introduction

Algorithms that establish Generalized Arc Consistency (GAC) on constraint problems (networks) filter out inconsistent values from variable domains in order to reduce the combinatorial search spaces of such problems. They have been a staple of Constraint Programming (CP) since its origin in the field of Artificial Intelligence (AI) in the seventies, with for example the introduction of algorithms (G)AC3 [Mac77] and (G)AC4 [MH86, MM88]. Typically, GAC is enforced at each step of a complete backtrack search, leading to the so-called MAC, Maintaining (generalized) Arc Consistency, algorithm [SF94a]. This chapter introduces a new GAC algorithm, called STR3, that works with positive table constraints. Furthermore, unlike most GAC algorithms, STR3 is specifically conceived to be used within MAC rather than being standalone.

A table is just a relation, as in classical relational database, and a positive table constraint contains (in a table) all permitted combinations of values for a subset of variables (whereas a negative table constraint contains all forbidden combinations of values). Table constraints have been well studied in the artificial intelligence literature and arise naturally in many application areas. For example, in configuration and databases, they are introduced to model the problem whatever the domain is. Besides, table constraints can be viewed as the universal mechanism for representing constraints, provided that space requirements can be controlled. The importance of table constraints makes them commonly implemented in all major constraint solvers that we are aware of (e.g., Choco, GeCode, JaCoP, OR-Tools).

For table constraints, many classical filtering algorithms that reduce search through inference (such as [BR97, LR05, LS06, BRYZ05]) work with constraints that stay unaltered while running. However, recent developments suggested that reducing the amount of traversal by discarding irrelevant tuples from tables can lead to faster algorithms. Simple Tabular Reduction (STR) and its improvements [Ull07, Lec11] fall into this category and have been shown to be among the best GAC algorithms for positive table constraints.

The main idea behind simple tabular reduction is to remove invalid tuples from tables as soon as possible in a systematic fashion. STR3 is based on the same principle as STR1 [Ull07] and STR2 [Lec11] but employs a different representation of table constraints. Similarly to a few other algorithms (e.g., GAC-allowed [BR97] and GAC-va [LS06]), STR3 provides an index for each constraint table, enabling a tuple sought with respect to a domain value to be found without visiting irrelevant tuples, thus reducing time complexity. Figure 7.1 shows an example for a ternary constraint. Importantly, for each constraint relation, STR3 maintains some specific data structures designed so that no constraint tuple is processed more than once along any path, through the search tree, going from the root to a leaf.

	X	Y	Z
1	a	f	l
2	b	f	m
3	e	g	m
4	a	f	m
5	b	g	o
6	a	h	o
7	d	h	o
8	b	i	n
9	c	j	k

(a) Standard table

X		Y		Z	
a	{1,4,6}	f	{1,2,4}	k	{9}
b	{2,5,8}	g	{3,5}	l	{1}
c	{9}	h	{6,7}	m	{2,3,4}
d	{7}	i	{8}	n	{8}
e	{3}	j	{9}	o	{5,6,7}

(b) Dual table

Figure 7.1: Standard and dual representations of the relation of a ternary constraint C with scope $\{X, Y, Z\}$.

Most of the GAC algorithms for table constraints previously introduced in the literature suffer from repeatedly traversing the same tables or related data structures during search [Lec11, CY10]. In contrast, STR3 avoids such repetition and is path-optimal: each element of a table is examined at most once along any path of the search tree. An important feature of STR3 is that it is designed specifically to be interleaved with backtracking search, where the main goal is to maintain the consistency while minimizing the cost of backtracking. As such, unlike most other GAC algorithms, STR3 is only applicable within the context of search: STR3 maintains GAC, but before commencement of search, GAC must be enforced by some other algorithm, such as STR2 for example.

We also investigated a promising circular manner for traversing tables in STR3. Although this seemed attractive at first because the circular approach described in [Gen13] has an optimal run time per branch when amortized across a search tree, our experiments found that it was not really effective for STR3 in practice.

To conclude our theoretical analysis, we discuss the relationships between STR3 and two other optimal GAC algorithms for table constraints, namely, GAC4 [MM88] and AC5TC-Tr [MHD12].

We present an extensive experimental study that demonstrates that STR3 is competitive with state-of-the-art algorithms. In particular, our experiments show that STR3 is rather complementary to STR2. STR2 is faster than STR3 where simple tabular reduction can eliminate so many tuples from the tables that they become largely empty. STR3, by contrast, outperforms STR2 when constraint relations do not shrink very much during search (this is when STR2 is the more costly). Hence, STR3 is complementary to STR2.

This chapter is organized as follows. Technical background is provided in Section 7.3. In Section 7.4, the concept of STR3 is explained together with its algorithm. A detailed example of STR3's step-by-step execution is given in Section 7.5. Section 7.6 analyzes the relationships among STR's data structures in greater detail. Theoretical analysis of STR3 is carried out in Sections 7.7 and 7.8. A variant of STR3 is studied in Section 7.9. Previous works related to STR3 are discussed in Section 7.10. Experimental results are reported in Section 7.11. The chapter concludes in Section 7.12.

7.3 Preliminaries

In this section, we introduce some technical background concerning the constraint satisfaction problem, and we recall the operation of a data structure called sparse set, which is key to STR3's optimality.

7.3.1 Constraint Satisfaction Problem

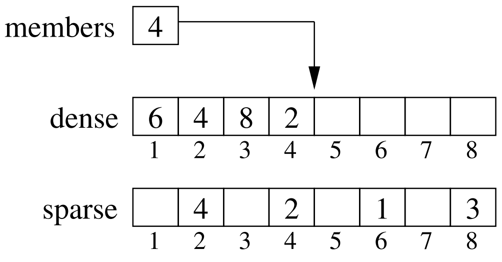
A finite *constraint network* $\mathcal{P}$ is a pair $(\mathcal{X}, \mathcal{C})$ where $\mathcal{X}$ is a finite set of n variables and $\mathcal{C}$ is a finite set of e constraints. Each *variable* $X \in \mathcal{X}$ has an *initial domain*, denoted by $D(X)$, which is the set of values that can be assigned to X. Each *constraint* $C \in \mathcal{C}$ involves an ordered subset of variables of $\mathcal{X}$, denoted by $scp(C)$, that is called the *scope* of C. The *arity* of a constraint C is the number of variables involved in C, i.e., $|scp(C)|$. A *binary* constraint involves two variables whereas a *non-binary* constraint involves strictly more than two variables. The semantics of a constraint C is given by a *relation*, denoted by $\mathit{rel}(C)$; if $\mathit{scp}(C)) = \{X_1, \ldots, X_r\}$, then $\mathit{rel}(C) \subseteq \prod_{i=1}^{r} D(X_i)$ represents the set of satisfying combinations of values, called *allowed* tuples, for the variables in $\mathit{scp}(C)$.

A *solution* to a constraint network is an assignment of a value to every variable such that every constraint is satisfied. A constraint network is *satisfiable* iff at least a solution exists. The *Constraint Satisfaction Problem* (CSP) is the NP-complete task of determining whether a given constraint network is satisfiable or not. Thus, a CSP instance is defined by a constraint network, which is solved either by finding a solution or by proving that no solution exists. Solving a CSP instance usually involves a complete backtrack search that is interleaved with some inference processes to reduce the search space. In this chapter, we shall focus on MAC (Maintaining Arc Consistency) [SF94b], which is considered to be among the most efficient generic search algorithms for CSP. MAC explores the search space depth-first, backtracks when dead-ends occur, and enforces a property called (generalized) arc consistency after each decision (variable assignment or value refutation) taken during search.

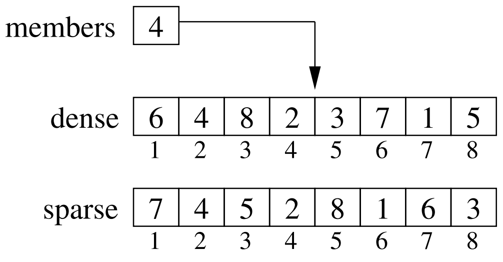
Below, we introduce some notations and definitions that will be useful in the rest of the chapter.

During search, $D^c(X)$ denotes the *current domain* of a variable $X \in \mathcal{X}$; we always have $D^c(X) \subseteq D(X)$. If a value $a \in D^c(X)$, we say that a is (currently) *present*; otherwise we say that a is *absent*. We use (X, a) to denote the value $a \in D(X)$ (or simply a when the context is clear), and we use $\tau[X_i]$ to denote the value a_i in any r-tuple $\tau = (a_1, \ldots, a_r)$ associated with a r-ary constraint C such that $\mathit{scp}(C) = \{X_1, \ldots, X_r\}$. A tuple $\tau \in \mathit{rel}(C)$ is *valid* iff $\tau[X] \in D^c(X)$ for each $X \in \mathit{scp}(C)$; otherwise τ is *invalid*. A tuple $\tau \in \mathit{rel}(C)$ is a *support* of (X, a) on C iff τ is valid and $\tau[X] = a$. We can now define *Generalized Arc Consistency* (GAC) as follows. A value (X, a) is generalized arc-consistent (GAC) on a constraint C involving X iff there exists a support of (X, a) on C. A constraint C is GAC iff for each $X \in \mathit{scp}(C)$ and for each $a \in D^c(X)$, (X, a) is GAC on C. A constraint network is GAC iff each of its constraints is GAC.

When $\mathit{rel}(C)$ is specified by enumerating its elements in a list, C is called a *positive table constraint*. Alternatively, $\mathit{rel}(C)$ may denote the set of forbidden combinations of values for the variables in $\mathit{scp}(C)$, in which case its extensional form is called a *negative table constraint*. A positive table constraint can be converted into a negative table constraint and vice versa. In this chapter we deal only with positive table constraints. For any such constraint C, we assume



(a) Focusing on present elements.



(b) Handling both present and absent elements.

Figure 7.2: A sparse set S with 4 elements: $S = \{2, 4, 6, 8\}$.

a total ordering on $rel(C)$ and define $\texttt{pos}(C, \tau)$ to be the position of the tuple τ in that ordering (also called "tuple identifier" in the database literature, or *tid* for short [SHWK76, ABC+76]) whereas $\texttt{tup}(C, k)$ denotes the k-th tuple of $rel(C)$. Thus, $\tau = \texttt{tup}(C, \texttt{pos}(C, \tau))$ for any tuple $\tau \in rel(C)$, and $k = \texttt{pos}(C, \texttt{tup}(C, k))$ for any $k \in \{1, \ldots, t\}$ where $t = |rel(C)|$.

7.3.2 Sparse Sets

The sparse set data structure was first proposed in [BT93], with the motivation to provide fast operations on sets of objects. These operations are clear-set, insertion, deletion, membership test, and iteration. Sparse sets have played a crucial role in many recent CP algorithms [CY10, cdsmSSL13, GSL10]. In the context of backtracking search, we are concerned with the speed of only three basic operations, which are insertion, membership test, and deletion.

A sparse set S is an abstract structure composed of two arrays, traditionally called `dense` and `sparse`, together with an integer variable called `members`. Both arrays are of equal size n, with an index ranging from 1 to n, because possible elements are drawn from the universe $\{1, \ldots, n\}$. The array `dense` acts like a normal array container, with all elements packed at the left, while the array `sparse` carries each element present in the set to its location in `dense`. The value of `members` indicates the number of elements in S. Arrays `dense` and `sparse` adhere to the following property:

$$v \in S \quad \Leftrightarrow \quad S.\texttt{sparse}[v] \leq S.\texttt{members} \ \wedge \ S.\texttt{dense}[S.\texttt{sparse}[v]] = v. \qquad (7.1)$$

An illustration is given by Figure 7.2a, where the sparse set representation of a set currently containing 4 values (out of 8 possible ones) is shown. Note that the arrays `dense` and `sparse` require no initialization[1]. Pseudo-code for insertion and membership test is given in Algorithm 21. Deletion in LIFO (Last In, First Out) order is achieved by decreasing the value of `members`.

In practice, membership tests may occur more frequently than insertions. In this case, a trade-off can be made so that the cost of testing the membership of a value v in a sparse set S is reduced to simply checking whether $S.\texttt{sparse}[v] \leq S.\texttt{members}$, at the expense of a more expensive insertion [GSL10]. The idea is to have the array `dense` containing a permutation of the n values. Operations on the set maintain such a permutation by swapping elements in and out of S. Because both arrays `dense` and `sparse` are initially filled with values from 1 to n, there must be only a single copy of each value in `dense` at any time. The condition $S.\texttt{dense}[S.\texttt{sparse}[v]] = v$ is thus unnecessary. The pseudo-code of the modified `addMember` and `isMember` is given in Algorithm 22. In addition, note that the enumeration of the elements that are not present in S is as simple as iterating from $\texttt{members} + 1$ to n in `dense`. An illustration is given by Figure 7.2b.

Algorithm 21: Original sparse set operations

```
1 addMember($S, v$)
2     $S.\texttt{members}++$
3     $S.\texttt{dense}[S.\texttt{members}] \leftarrow v$
4     $S.\texttt{sparse}[v] \leftarrow S.\texttt{members}$

5 isMember($S, v$)
6     $i \leftarrow S.\texttt{sparse}[v]$
7     return $i \leq S.\texttt{members}$ and $S.\texttt{dense}[i] = v$
```

7.4 STR3

This section introduces STR3, an algorithm based on simple tabular reduction for enforcing GAC on positive table constraints. GAC algorithms normally follow the same pattern: a domain value is proved to be consistent by producing a valid tuple containing that value (in the case of positive table constraints) or

[1]Actually, this is true provided that `sparse` only contains strictly positive values. In [BT93], it is assumed that indexing starts at 0 and `sparse` is an array of unsigned integers.

Algorithm 22: Sparse set operations optimized for membership testing.

```
addMember($S, v$)
    $S$.members ++
    $w \leftarrow S$.dense[$S$.members]
    $i \leftarrow S$.sparse[$v$]
    $S$.dense[$S$.members] $\leftarrow v$
    $S$.sparse[$v$] $\leftarrow S$.members
    $S$.dense[$i$] $\leftarrow w$
    $S$.sparse[$w$] $\leftarrow i$

isMember($S, v$)
    return $S$.sparse[$v$] $\leq S$.members
```

by producing some evidence from auxiliary structures, e.g., a path in case of BDDs (Boolean Decision Diagrams), MDDs (Multi-valued Decision Diagrams), or tries [CY10, GJMN07, CY06]. This is usually done by traversing these structures and running tests on each sub-structure. Reducing the amount of traversal has long been the focus of many works. For table constraints, optimization techniques include skipping over irrelevant rows or columns [LR05, LS06], or by compressing the tables themselves [CY10, GJMN07, CY06].

Simple Tabular Reduction (STR) [Ull07], called STR1 in this chapter, is a GAC algorithm that dynamically revises tables during search. While other GAC algorithms treat tables like fixed structures or resort to tackling comparable structures created from those tables, STR1 shows that handling tables directly during backtracking search is not as expensive as once thought. STR1 works by scanning each tuple one by one. If a tuple is invalid, it is removed and the table is contracted by one row as a result. Otherwise, the tuple is valid and its components are therefore domain values that have been proved to have a support. STR1 then considers the tuple again to collect these values into designated sets, (called `gacValues`(X) and defined for every variable X in the scope of the constraint). When STR1 has finished going through the whole table, any value not present in those sets has no support and will be removed from its domain.

STR2 [Lec11] provides two improvements to STR1. When a tuple τ is being inspected for validity, there is no need to check whether $\tau[X] \in D^c(X)$ if there has been no change to the domain of X since the last time STR2 was called on this constraint. In addition, in case τ is valid, when the algorithm goes through each component of τ, $\tau[X]$ is skipped over if it is known that `gacValues`$(X) = D^c(X)$ (i.e., every single value of $D^c(X)$ is already supported). The first improvement however involves additional data structures which must be maintained in order to deliver full optimization benefit. STR2 with these data structures maintained is called STR2+ [Lec11]. Throughout this chapter and especially in the experiments section, when we refer to STR2 we mean STR2+.

Like STR1 and STR2, STR3 no longer examines a tuple once it has been recognized as invalid. Unlike STR1 and STR2, STR3 does not immediately discard invalid tuples from tables. Indeed, STR3 does not process tables directly, but instead employs indexes allowing rapid identification of all tuples containing a given value of a given variable. STR3 keeps a separate data structure that enables validity checks to be done in constant time, rather than revising indexes dynamically during search.

7.4.1 Structures and Features

The main data structures used in STR3 are the following:

- For any $C \in \mathcal{C}, X \in scp(C)$, and $a \in D(X)$, we introduce `table`(C, X, a), called *sub-table* of C for (X, a), as the set of *tids* for allowed tuples of C involving (X, a), i.e., $\{\texttt{pos}(C, \tau) \mid \tau \in rel(C) \;\wedge\; \tau[X] = a\}$. We implement `table`$(C, X, a)$ as a simple array, with indexing starting at 1; the *i-th* element of `table`(C, X, a) is then denoted by `table`$(C, X, a)[i]$. We use an integer `table`(C, X, a)`.sep`, whose value lies between 1 and $|\texttt{table}(C, X, a)|$, which we call the *separator* of `table`(C, X, a). The separator is such that `table`$(C, X, a)[$`table`(C, X, a)`.sep`$]$ is the position of the last known support of (X, a) on C. For the sake of brevity, we sometimes use `table`$(C, X, a)[\uparrow]$ instead of `table`$(C, X, a)[$`table`(C, X, a)`.sep`$]$. The value of `sep` is maintained during search, that is to say, subject to save and restore operations.
- For any $C \in \mathcal{C}$, we introduce `inv`(C) as the set of *tids* for allowed invalid tuples of C, i.e., $\{\texttt{pos}(C, \tau) \mid \tau \in rel(C) \;\wedge\; \tau \text{ is invalid}\}$. We implement `inv`$(C)$ as a sparse set. The value of `inv`(C)`.members` is subject to save/restore operations during search.
- For any $C \in \mathcal{C}$ and $k \in \{1, \ldots, |rel(C)|\}$, we introduce `dep`$(C, k)$ as the *dependency list* associated with the k-th tuple of $rel(C)$. If $(X, a) \in$

`dep`(C, k), this means that the tuple $\tau =$ `tup`(C, k) provides the justification for a to be present in $D^c(X)$, i.e., (X, a) *depends* on τ. We implement `dep`(C, k) as a simple array (with indexing starting at 1) since only sequential iterations and basic transfers are required. The dependency lists may be altered during search but they are not maintained (i.e., subject to save/restore operations.)

- We introduce two stacks, denoted by `stackS` (S stands for Separator) and `stackI` (I stands for Invalid), that allow us to store values of the form `table`(C, X, a)`.sep` and `inv`(C)`.members` at different levels of search. Whenever a node in the search tree is visited for the first time, it is assumed that an empty container is placed on top of `stackS` and `stackI`.

STR3 is a fine-grained filtering algorithm, meaning that the filtering process, called propagation, is guided by events corresponding to deleted values that are put in a so-called propagation queue. Here are some important attributes of STR3:

- When a value (X, a) is deleted, it is put in the propagation queue. This value is then picked from the queue later by the main constraint propagation procedure (not detailed in this chapter), and STR3 is invoked for every constraint C such that $X \in scp(C)$. This invocation merges `table`(C, X, a) into `inv`(C), because `table`(C, X, a) contains the *tids* of all tuples that involve the deleted value (X, a).

- Subsequently, STR3 recognizes that a value (Y, b) is GAC on C if `table`$(C, Y, b) \setminus$ `inv`(C) is not empty.

- The separator `table`(C, X, a)`.sep` is useful to distinguish between two regions: the explored region which contains *tids* of tuples (of $rel(C)$ involving (X, a)) known to be invalid, and the unexplored region which contain *tids* of tuples not yet examined. Each separator moves sequentially from one end of `table`(C, X, a) to the other in a fixed direction. As search progresses, the explored region expands until it encompasses the whole set, at which point (X, a) has been proved not to be GAC on C.

- To check whether k is the *tid* of a tuple that has been found to be invalid, STR3 tests whether $k \in$ `inv`(C), through a call of the form `isMember`(`inv`$(C), k)$.

- Whenever a tuple of *tid* k becomes invalid, STR3 must look for a new support for every value in the dependency list $\texttt{dep}(C, k)$.

7.4.2 Algorithm

Algorithm 23: Algorithm STR3

```
GACinit($C$: Constraint)
  remove invalid tuples from $rel(C)$ and build every $\texttt{table}(C, X, a)$
  $\texttt{inv}(C).\texttt{members} \leftarrow 0$
  foreach $k \in \{1, \ldots, |rel(C)|\}$ do
      $\texttt{dep}(C, k) \leftarrow \emptyset$
  foreach $X \in scp(C)$ and $a \in D^c(X)$ do
      $\texttt{table}(C, X, a).\texttt{sep} \leftarrow |\texttt{table}(C, X, a)|$
      $k \leftarrow \texttt{table}(C, X, a)[\uparrow]$
      $\texttt{dep}(C, k) \leftarrow \texttt{dep}(C, k) \cup \{(X, a)\}$

STR3($C$: Constraint, $X$: Variable, $a$: Value)
  $\texttt{membersBefore} \leftarrow \texttt{inv}(C).\texttt{members}$
  for $p \leftarrow 1$ to $\texttt{table}(C, X, a).\texttt{sep}$ do
      $k \leftarrow \texttt{table}(C, X, a)[p]$
      if $\neg isMember(\texttt{inv}(C), k)$ then
          $\texttt{addMember}(\texttt{inv}(C), k)$
  if $\texttt{membersBefore} = \texttt{inv}(C).\texttt{members}$ then
      return true
  $\texttt{save}(C, \texttt{membersBefore}, \texttt{stackI})$
  foreach $i \in \{\texttt{membersBefore} + 1, \ldots, \texttt{inv}(C).\texttt{members}\}$ do
      $k \leftarrow \texttt{inv}(C).\texttt{dense}[i]$
      foreach $(Y, b) \in \texttt{dep}(C, k)$ such that $b \in D^c(Y)$ do
          $p \leftarrow \texttt{table}(C, Y, b).\texttt{sep}$
          while $p > 0$ and $isMember(\texttt{inv}(C), \texttt{table}(C, Y, b)[p])$ do
              $p \leftarrow p - 1$
          if $p = 0$ then
              $\texttt{removeValue}(C, Y, b)$
              if $D^c(Y) = \emptyset$ then return false
          else
              if $p \neq \texttt{table}(C, Y, b).\texttt{sep}$ then
                  $\texttt{save}((C, Y, b), \texttt{table}(C, Y, b).\texttt{sep}, \texttt{stackS})$
                  $\texttt{table}(C, Y, b).\texttt{sep} \leftarrow p$
              move $(Y, b)$ from $\texttt{dep}(C, k)$ to $\texttt{dep}(C, \texttt{table}(C, Y, b)[p])$
  return true
```

We emphasize that STR3's sole purpose is to maintain GAC during search [LZS+07, LZBF04]. It is not designed to establish GAC stand-alone. For that role, a separate GAC algorithm that is able to enforce GAC from scratch is

Algorithm 24: Auxiliary functions of Algorithm STR3

```
1  save(key, newData, store)
2    if $(key, oldData) \in top(store)$ for some oldData then
3        replace $(key, oldData)$ with $(key, newData)$
4    else
5        insert $(key, newData)$ to $top$(store)

6  restoreS()
7    $list \leftarrow pop($stackS$)$
8    foreach $((C, X, a), p) \in list$ do table$(C, X, a)$.sep $\leftarrow p$

9  restoreI()
10   $list \leftarrow pop($stackI$)$
11   foreach $(C, m) \in list$ do inv$(C)$.members $\leftarrow m$

12 removeValue($C$: Constraint, $X$: Variable, $a$: Value)
13   remove $a$ from $D^c(X)$
14   add $(C', X, a)$ to the propagation queue where $C' \neq C$ and $X \in scp(C')$
```

required, and it must be invoked before the search commences, usually in the preprocessing stage.

Pseudo-code of STR3 is given in Algorithms 23 and 24. `GACInit` (lines 1–9) is called first to remove all invalid tuples (by calling another GAC algorithm at line 2) and to initialize all data structures. In the beginning, `inv`(C)`.members` is set to zero as the remaining tuples are all valid (line 3), while the separator of each `table`(C, X, a) is set to its last possible index (recall that we start indexing at 1). We also put each value (X, a) into an arbitrary dependency list.

During search, `STR3` (lines 10–33) is called for a constraint C every time a value a is removed from the domain of a variable X involved in C. Note that the instantiation of a variable X effectively invokes `STR3`(C, X, a) for every value a that is present in the domain of X at the time of the assignment but which is not the value assigned to X. For each removed value (X, a), every tuple whose *tid* is in `table`(C, X, a) becomes invalid. `STR3` then merges these *tids* into `inv`(C) if they are not already present (lines 12–15). Values that need new supports are later processed (lines 19–32); we shall discuss this part of the algorithm in more details in Section 7.6. Upon backtracking, functions `restoreS` and `restoreI` are called so as to restore elements `table`(C, X, a)`.sep` and `inv`(C)`.members`, through the use of the stacks `stackS` and `stackI`. Values are stored in these stacks at lines 18 and 30 by calling function `save`.

	X					Y					Z				
	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
`table`(C)	1	2	$9_{\lhd_1}$	$7_{\lhd_1}$	$3_{\lhd_1}$	1	3	6	$8_{\lhd_1}$	$9_{\lhd_1}$	$9_{\lhd_1}$	$1_{\lhd_1}$	2	$8_{\lhd_1}$	5
	4	5				2	$5_{\lhd_1}$	$7_{\lhd_1}$					3		6
	$6_{\lhd_1}$	$8_{\lhd_1}$				$4_{\lhd_1}$							$4_{\lhd_1}$		$7_{\lhd_1}$

k	0	1	2	3	4	5	6	7	8	9
`inv`(C)`.sparse`										
`inv`(C)`.dense`										
`inv`(C)`.members`	$\uparrow_1$									
`dep`(C,k)		l		e	f	g	a	d	b	c
					m			h	i	j
								o	n	k

Figure 7.3: Status right after GAC preprocessing at node η_1. Sub-tables and dependency lists are displayed vertically (at the top and the bottom of the figure, respectively).

	X					Y					Z				
	a	b	c	$\boxed{d}$	e	f	g	$\boxed{h}$	$\boxed{i}$	j	k	l	m	$\boxed{n}$	$\boxed{o}$
`table`(C)	1	$2_{\lhd_2}$	$9_{\lhd_1}$	$7_{\lhd_1}$	$3_{\lhd_1}$	1	$3_{\lhd_2}$	6	$8_{\lhd_1}$	$9_{\lhd_1}$	$9_{\lhd_1}$	$1_{\lhd_1}$	2	$8_{\lhd_1}$	5
	$4_{\lhd_2}$	5				2	$5_{\lhd_1}$	$7_{\lhd_1}$					3		6
	$6_{\lhd_1}$	$8_{\lhd_1}$				$4_{\lhd_1}$							$4_{\lhd_1}$		$7_{\lhd_1}$

k	0	1	2	3	4	5	6	7	8	9
`inv`(C)`.sparse`						4	1	2	3	
`inv`(C)`.dense`		6	7	8	5					
`inv`(C)`.members`	$\uparrow_1$				$\uparrow_2$					
`dep`(C,k)		l	b	e	f			d	i	c
				g	m			h	n	j
					a			o		k

Figure 7.4: Status at η_2

7.5 Illustration

In this section, we trace the execution of STR3 on the ternary positive table constraint C, with scope $\{X, Y, Z\}$, depicted in Figure 7.1a. For each value in the table, its index or sub-table is given in Figure 7.1b. After GAC preprocessing, separators and dependency lists are initialized as shown in Figure 7.3. The symbol $\lhd_p$, where p is a number, points to the value whose position is assigned to `sep` at node η_p. We also denote the fact that `inv`(C)`.members` is

	X					Y					Z				
	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
`table`(C)	1	2	$9_{\lhd 1}$	$7_{\lhd 1}$	$3_{\lhd 1}$	1	3	6	$8_{\lhd 1}$	$9_{\lhd 1}$	$9_{\lhd 1}$	$1_{\lhd 1}$	2	$8_{\lhd 1}$	5
	4	5				2	$5_{\lhd 1}$	$7_{\lhd 1}$					3		6
	$6_{\lhd 1}$	$8_{\lhd 1}$				$4_{\lhd 1}$							$4_{\lhd 1}$		$7_{\lhd 1}$

k	0	1	2	3	4	5	6	7	8	9
`inv`(C)`.sparse`						4	1	2	3	
`inv`(C)`.dense`		6	7	8	5					
`inv`(C)`.members`	$\uparrow_1$									
`dep`(C, k)		l	b	e	f			d	i	c
				g	m			h	n	j
					a			o		k

Figure 7.5: After backtracking to η_1

	X					Y					Z				
	a	b	c	d	$\boxed{e}$	f	g	h	$\boxed{i}$	j	k	l	m	$\boxed{n}$	o
`table`(C)	1	2	$9_{\lhd 1}$	$7_{\lhd 1}$	$3_{\lhd 1}$	1	3	6	$8_{\lhd 1}$	$9_{\lhd 1}$	$9_{\lhd 1}$	$1_{\lhd 1}$	2	$8_{\lhd 1}$	5
	4	5				2	$5_{\lhd 1}$	$7_{\lhd 1}$					3		6
	$6_{\lhd 1}$	$8_{\lhd 1}$				$4_{\lhd 1}$							$4_{\lhd 1}$		$7_{\lhd 1}$

k	0	1	2	3	4	5	6	7	8	9
`inv`(C)`.sparse`				1		4	1	2	2	
`inv`(C)`.dense`		3	8	8	5					
`inv`(C)`.members`	$\uparrow_1$		$\uparrow_3$							
`dep`(C, k)		l	b	e	f	g		d	i	c
					m			h	n	j
					a			o		k

Figure 7.6: Status at η_3

assigned the value k at node η_p by placing $\uparrow_p$ at column k on the row titled `inv`(C)`.members`. For instance at node η_1, we have `inv`(C)`.members` $= 0$ and `table`(C, X, a)`.sep` $= 3$ (because $|$`table`$(C, X, a)| = 3$ the value of the separator ranges from 1 to 3, from top to bottom). We can also observe for example that `table`$(C, X, a) = \{1, 4, 6\}$ and `table`$(C, X, a)[3] = 6$ (this will be always true since sub-tables are never directly modified; only the separators can change).

Assume values h, i, and o are eliminated. The result after STR3's propagation converges is shown in Figure 7.4. In all figures hereafter, we mark a domain value that has been deleted by putting it inside a box. STR3 eventually merges the *tids* of tuples that involve these values (`table`(C, Y, h), `table`(C, Y, i),

$\texttt{table}(C, Z, o))$ into $\texttt{inv}(C)$, making $\texttt{inv}(C).\texttt{dense} = \{6, 7, 8, 5\}$. Values that depend on the tuples with those *tids* are in need of new supports. They are $\bigcup_{5 \leq k \leq 8} \texttt{dep}(C, k) \setminus \{h, i, o\} = \{g, a, d, b, n\}$ (values h, i, and o are already absent so they stay at their positions according to the condition in line 21 of STR3). Let us look first at what happens to g. STR3 locates a new valid support for g which is $\texttt{tup}(C, 3)$. The value of $\texttt{table}(C, Y, g).\texttt{sep}$ is then changed from 1 to 0 and g is transferred from $\texttt{dep}(C, 5)$ to $\texttt{dep}(C, 3)$. Similar operations are performed with respect to a and b. Now, concerning values d and n, it is clear that no other support exists. Consequently, these values can be deleted from their respective domains while continuing to exist in $\texttt{dep}(C, 7)$ and $\texttt{dep}(C, 8)$.

Now suppose that the search backtracks to η_1. The result is shown in Figure 7.5. The dependency lists $\texttt{dep}(C, k)$ are unaffected: they remain as in η_2, but note that they are different from those initially in η_1 (see Figure 7.3 for comparison). This is an example of unsynchronized supports, which will be discussed in the next section. On the other hand, separators and $\texttt{inv}(C).members$ have been rolled back to their previous values, as initially in η_1.

From this point, we suppose that e and n are eliminated. Figure 7.6 shows the results after STR3 is called. The *tids* from both values' indexes (3 and 8) are first added to $\texttt{inv}(C)$. Values for which STR3 needs to produce new supports are $\bigcup_{k \in \{3,8\}} \texttt{dep}(C, k) \setminus \{e, n\} = \{g, i\}$. Value i is removed because it has no further support. Value g stays present because $\texttt{tup}(C, \texttt{table}(C, Y, g)[\uparrow])^2 = \texttt{tup}(C, 5)$ is a support of g. STR3 then moves g from $\texttt{dep}(C, 3)$ to $\texttt{dep}(C, 5)$. Notice that the value of $\texttt{table}(C, Y, g).\texttt{sep}$ remains unchanged from Figure 7.5. Finally, while it is true that the invalidation of $\texttt{table}(C, X, b)[\uparrow] = 8$ deprives b of a support, since b is not part of $\texttt{dep}(C, 8)$ STR3 will not try to find a new support for b. As a matter of fact, $b \in \texttt{dep}(C, 2)$. As a result, STR3 will start seeking a new support for b only after $\texttt{tup}(C, 2)$ becomes invalid.

7.6 Synchronized vs. Unsynchronized Supports

Central to STR3 is the relationship between the separators and the dependency lists. A present value (X, a) is GAC on C because (1) $(X, a) \in \texttt{dep}(C, k)$ for some $k \notin \texttt{inv}(C)$, or (2) $\texttt{table}(C, X, a)[\uparrow] = k'$ with $k' \notin \texttt{inv}(C)$. Only one of the two conditions is sufficient for (X, a) to be proved to be GAC, and when

[2]Again, $\texttt{table}(C, X, a)[\uparrow]$ referes to $\texttt{table}(C, X, a)[\texttt{table}(C, X, a).\texttt{sep}]$.

both conditions are true, STR3 does not necessarily force the *tid* k to be equal to the *tid* k' as might be expected. This flexibility allows STR3 to keep a maximum of two different supports for each value with virtually no effort, thus almost doubling the chance that STR3 can avoid seeking a new support later. This section studies when and how this happens.

When $k = k'$, we say that the dependency lists and the separators are *synchronized* at (X, a) (or that the supports of (X, a) are synchronized). `GACInit` initializes separators and dependency lists so that they are synchronized from the beginning (see lines 6–9). In this case, the role of the dependency lists is straightforward: it just mirrors what happens to the separators. As soon as a tuple $\texttt{tup}(C, k)$ becomes invalid, STR3 looks for a new support for each value in the dependency list indexed at k, i.e., $\texttt{dep}(C, k)$ (line 21). Potential supports for a value (Y, b) are in the sub-table $\texttt{table}(C, Y, b)$, so, they are tested one by one against $\texttt{inv}(C)$, starting from the separator of the sub-table (lines 22–24). If no support is found the value is removed (line 26), and `STR3` immediately fails when that value is the last one left in the domain (line 27). If a new support is found, the value of the current separator is recorded for backtrack purposes (line 30) before being replaced by the position of the new support (line 31). Dependency lists are always updated accordingly (line 32).

The separators and the dependency lists remain synchronized until backtracking occurs. Given $(X, a) \in \texttt{dep}(C, k)$ for some *tid* k, when the search backtracks $\texttt{dep}(C, k)$ remains unperturbed whereas $\texttt{table}(C, X, a).\texttt{sep}$ must revert back to its previous state if possible. For this reason, the separators and the dependency lists may no longer be synchronized at (X, a). In such cases, the tuples $\texttt{tup}(C, \texttt{table}(C, X, a)[\uparrow])$ and $\texttt{tup}(C, k)$ diverge and become two distinct supports of (X, a) on C.

We now consider in details different circumstances during `STR3`'s execution when the validity of these two tuples later change (not including the cases where `STR3` reports inconsistency). These relationships are portrayed in Figure 7.7. We assume $|\texttt{table}(C, X, a)| > 1$ for all (X, a) so that the claim of two distinct supports is not trivially unfeasible. The diagram is explained in details as follows. We consider $(X, a) \in \texttt{dep}(C, k)$ for some *tid* $k \notin \texttt{inv}(C)$ and $\texttt{table}(C, X, a)[\uparrow] = k'$ for some *tid* $k' \notin \texttt{inv}(C)$. In Figure 7.7, `dep` and `sep` failing means that k and k' are in $\texttt{inv}(C)$, respectively.

Supports are synchronized at the beginning, which means $k = k'$. There are two possible transitions:

(1) `tup`(C, k) *becomes invalid.* STR3 must seek a new support in this case. Consequently, both supports remain synchronized (albeit with a *tid* different from k).

(2) `table`(C, X, a)`.sep` *is restored to some previous value.* This is caused by backtracking. The two supports of (X, a) become unsynchronized.

Suppose there are two distinct supports for (X, a) at this point. There are three possible transitions, namely (3), (6), and (7).

(3) `tup`(C, k') *becomes invalid while* `tup`(C, k) *remains valid.* Because STR3 seeks a new valid support for (X, a) only when `tup`(C, k) is invalid (line 21), nothing needs to be done. This is what happens to value b in Figure 7.6. There are two possible choices after this state:

  (4) `tup`(C, k) *becomes invalid as well.* The search for a new valid support proceeds as usual. The dependency lists and the separators are synchronized at (X, a) on a newly acquired support if it exists.

  (5) `table`(C, X, a)`.sep` *is restored to some previous value.* Again, this is caused by backtracking and (X, a) would end up with two distinct supports as in case (2).

(6) `tup`(C, k') *remains valid while* `tup`(C, k) *becomes invalid.* (X, a) is simply shifted to another dependency list (i.e., from `dep`(C, k) to `dep`(C, k') by line 32). Because `tup`(C, k') is valid, the second condition in line 23 will fail and the separator pointing at k' will never move. The dependency lists and the separators are synchronized at (X, a) as a result. In effect, the remaining support is copied over when the other fails. This is what happens to value g in Figure 7.6.

(7) `table`(C, X, a)`.sep` *is restored to some previous value.* Same as (2) and (5).

Even though the trigger for `STR3` to seek a new support is set up only on dependency lists, from the diagram it is clear that both `tup`(C, k) and `tup`(C, k') in fact provide two different sources of supports for (X, a); when one fails STR3 will draw on the other for support equally, and only after both fail does `STR3` start seeking a new support. In an interesting scenario, a support in a dependency list may fail and get replaced by another support successively without STR3 having to explicitly seek a new support. The reason is that STR3

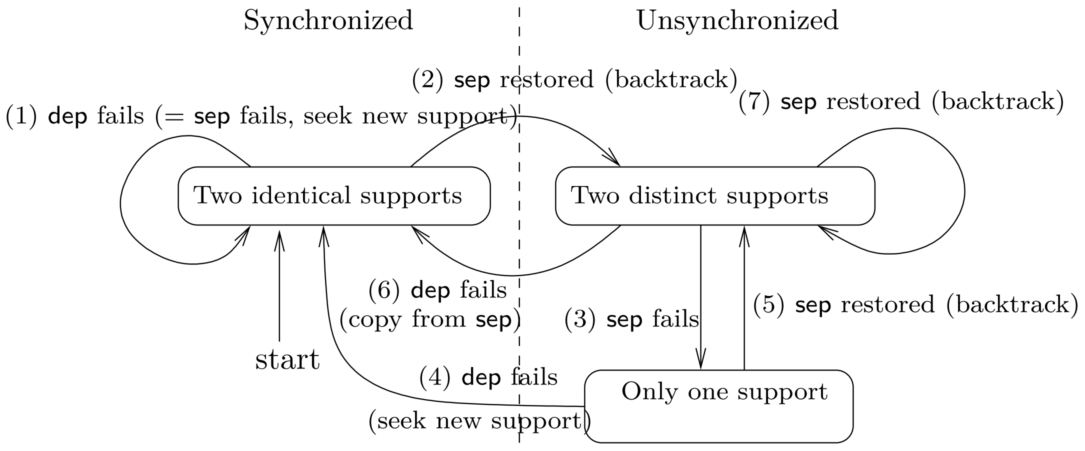


Figure 7.7: Transition diagram depicting the relationship between the two supports of a value (X, a). STR3 is triggered only when `dep` fails. Both supports of (X, a) are synchronized as a result.

may cycle from synchronized to unsynchronized state and back repeatedly through cases (2) and (6), where the limit is the depth of the search tree.

While STR3 starts off with synchronized supports, unsynchronized supports are possible too. Indeed, because STR3 requires a form of preprocessing in order to remove invalid tuples, the ones that remain are all supports. Consider Figure 7.1 for instance. After GAC preprocessing the tuples $\texttt{tup}(C, 2)$, $\texttt{tup}(C, 3)$, and $\texttt{tup}(C, 4)$ are all recognized as supports for (Z, m). In line 7, $\texttt{table}(C, Z, m).\texttt{sep}$ is assigned the value 3, therefore the ordinary STR3 that starts with synchronized supports would add (Z, m) to $\texttt{dep}(C, \texttt{table}(C, Z, m)[3]) = \texttt{dep}(C, 4)$. However, because the tuples $\texttt{tup}(C, 2)$, $\texttt{tup}(C, 3)$, and $\texttt{tup}(C, 4)$ are all supports of (Z, m), we could choose adding (Z, m) to $\texttt{dep}(C, \texttt{table}(C, Z, m)[1]) = \texttt{dep}(C, 2)$, or to $\texttt{dep}(C, \texttt{table}(C, Z, m)[2]) = \texttt{dep}(C, 3)$ as well. Since the separator moves down from the last cell to 1, choosing 1, the furthest cell away at the opposite end, would be the most natural choice. Line 8 would then be changed to:

$$k \leftarrow \texttt{table}(C, X, a)[1]$$

In our experiments, we use a variant of STR3 that starts with unsynchronized supports (thus benefiting initially from two supports for each value).

Observe that the separators and the dependency lists are somewhat comparable to watched literals [MMZ+01] introduced for SAT. However, there are significant differences as follows. To begin with, for a given value, the relevant dependency list is the only activation point, working as a primary support while the separator serves as a possible backup; the separator points to a tuple that may or may not be valid. In contrast, there are always two watched literals for SAT, both equivalent in every way. For STR3, the separators are rigid and must maintain their values at all times while the dependency lists are not maintained, just like the two watched literals for SAT. Separators and dependency lists can be synchronized or unsynchronized depending on circumstances, in effect providing either a single support or two distinct supports whereas for SAT the two watched literals are always distinct where possible.

7.7 Correctness

This section proves properties that are crucial to STR3's correctness. An invariant is taken to be a property that remains true throughout the search, at the points before and after STR3 is enforced, but not necessarily while STR3 is running. For simplicity, arrays are sometimes perceived as mathematical sets in what follows.

The first invariant states that the dependency lists associated with a constraint C represent a disjoint collection of all domain values for the variables in the scope of C.

Invariant 3 *For any constraint* $C \in \mathcal{C}$*, the collection of sets* $\mathcal{S} := \{\texttt{dep}(C, i) : i \in 1 \dots |\mathit{rel}(C)|\}$ *represents a partition of* $\mathcal{D} := \bigcup_{X \in \mathit{scp}(C)} D(X)$ *such that every element of* $\mathcal{D}$ *is in one and only one set of* $\mathcal{S}$*.*

Proof: The invariant holds initially after `GACInit` is called. Line 32 contains the only statement that affects dependency lists and it moves one element from one list to another, thus preserving the invariant. □

The second invariant states that $\texttt{inv}(C)$ contains all *tids* of invalid tuples from *rel*(C) and nothing else.

Invariant 4 *For any constraint* $C \in \mathcal{C}$*,*
$\texttt{inv}(C) = \bigcup_{X \in \mathit{scp}(C),\ a \notin D^c(X)} \texttt{table}(C, X, a)$*.*

Proof: This is obvious from the fact that STR3 merges $\texttt{table}(C, X, a)$ into $\texttt{inv}(C)$ as soon as (X, a) becomes invalid, but it is worth emphasizing that there are exactly $|\mathit{scp}(C)|$ copies for each *tid* and they are distributed among different subtables (see Figure 7.1b for example). Line 14 ensures that $\texttt{inv}(C)$

does not include duplicates and therefore conforms to the prerequisite of sparse sets. The invariant holds after backtracking because the right-hand side of the equation, $\bigcup_{X \in scp(C),\ a \notin D^c(X)}$ $\texttt{table}(C, X, a)$, is conditioned upon domain values while the left-hand side, $\texttt{inv}(C)$, is controlled through $\texttt{inv}(C).\texttt{members}$, both of which are maintained by STR3. □

The third invariant guarantees that no support resides in the explored regions, i.e., after separators. We use $\texttt{table}(C, X, a).\texttt{explored}$ to denote $\texttt{table}(C, X, a)[\texttt{sep} + 1 \ldots \texttt{size}]$ where $\texttt{sep} = \texttt{table}(C, X, a).\texttt{sep}$ and $\texttt{size} = |\texttt{table}(C, X, a)|$.

Invariant 5 *For any $C \in \mathcal{C}$, $X \in scp(C)$, and $a \in D^c(X)$, no* tid *of any support of (X, a) on C exists in* $\texttt{table}(C, X, a).\texttt{explored}$.

Proof: This invariant holds when the search starts since `GACInit` initially eliminates all invalid tuples and assigns the separators to their maximum values. Afterwards, when the tuple $\texttt{tup}(C, \texttt{table}(C, X, a)[\uparrow])$ becomes invalid, `STR3` scans down $\texttt{table}(C, X, a)$ until a new valid support is found, in which case the separator is set to the new value; the invariant holds. If no valid support is found, a is removed and the separator remains unchanged. The invariant still holds because it is conditioned on a being present in the domain. When a backtrack occurs, a becomes present again and the invariant still holds because the separator is maintained. □

Finally, the fourth invariant states that values stored in dependency lists do correspond to supports (for present values).

Invariant 6 *For any $C \in \mathcal{C}$, $X \in scp(C)$, and $a \in D^c(X)$, if $(X, a) \in \texttt{dep}(C, k)$, then $\texttt{tup}(C, k)$ is a support of (X, a) on C.*

Proof: The invariant holds right after `GACInit`. From the code, we see that whenever $\texttt{tup}(C, k)$ becomes invalid, any (X, a) in $\texttt{dep}(C, k)$ will be moved to another $\texttt{dep}(C, k')$ when a different valid *tid* k' is found (line 32). The invariants associated with $\texttt{dep}(C, k)$ and $\texttt{dep}(C, k')$ are preserved. If no valid *tid* is found, (X, a) becomes invalid. The invariant remains true because it deals only with present values.

We now look at the relationship between $\texttt{table}(C, X, a).\texttt{sep}$ and dependency lists when a backtrack is involved. If (X, a) switches from being absent to present after a backtrack, the invariant remains true, because either (1) (X, a) was removed as a consequence of the instantiation of X to some other value

$b \neq a$, in which case the invariant is unaffected, or (2) chronological backtracking ensures that the tuple that (X, a) depended on most recently is restored as well (through rolling back of separators and `inv`(C)`.members`). An interesting situation happens when (X, a) is present before and after backtracking. In this case, `table`(C, X, a)`.sep` may be reverted. Assume the value of `table`$(C, X, a)[\uparrow]$ is k before the backtrack, and k' after the backtrack (with $k < k'$). This means of course that (X, a) is in `dep`(C, k) before the backtrack. Now, consider `dep`(C, k) and `dep`(C, k') after backtrack. Because backtracking never invalidates tuples, the tuple `tup`(C, k) must still be valid after backtrack, and because dependency lists are not maintained, (X, a) remains in `dep`(C, k). For this reason, the invariant for `dep`(C, k) is still true, although `table`$(C, X, a)[\uparrow]$ is no longer k. The invariants involving values in `dep`(C, k') are unaffected.

Next, consider what happens if the search moves forward when there are two distinct supports. That is, $(X, a) \in$ `dep`(C, k) while `table`$(C, X, a)[\uparrow] = k' \neq k$. If `tup`$(C, k)$ becomes invalid, we need to find a new support for (X, a). If there exists $1 \leq k'' \leq k'$ such that `tup`(C, k'') is valid, `STR3` merely moves (X, a) from `dep`(C, k) to `dep`(C, k''). The invariants for `dep`(C, k) and `dep`(C, k'') hold afterward. If no valid support is found, (X, a) remains in `dep`(C, k) and a becomes absent, making the invariant trivially true. □

We can now prove that STR3 does maintain GAC during backtrack search.

Theorem 11 *STR3 maintains GAC.*

Sketch of Proof: We assume the standard value-based propagation framework and that the network is already GAC before STR3 is called for the first time. Two key observations for the completion of a proof are as follows. First, a value is deleted and put in the propagation queue as soon as STR3 exhausts all possibilities for supports (Invariant 5 where `table`(C, X, a)`.explored` $=$ `table`(C, X, a) and line 26). Second, every present value has at least one valid support. This is due to Invariants 3 and 6 and the fact that a present value depends on exactly one tuple of a table constraint. □

7.8 Complexity

Many algorithms repeatedly compute a new value from an old one after a small modification to the computation context. An algorithm is incremental if it does not compute the new value from scratch but exploits both the old value and the modifications made to the environment. STR3 is designed to be incremental by avoiding repeated domain checks along the same path, going from the root to a leaf, in the search tree. In the following analysis, we consider the worst-case accumulated cost along a single path of length m in the search tree. It is assumed that (1) each variable domain is of size d, (2) each positive table constraint is of arity r and contains t tuples, (3) the tables do not contain invalid tuples before STR3 starts. Also, we consider that $d \leq t$ (a value $a \in D(X)$ must initially appear in all tables involving the variable X.)

Theorem 12 *The worst-case space complexity of STR3 is $O(rt + mrd)$ per constraint.*

Proof: There are three main data structures in STR3: `table`, `inv` and `dep`. According to Invariant 3, the space complexity for `dep` is $O(rd)$. For `inv`, it is $O(t)$ whereas it is $O(rt)$ for `table`. For managing the restoration of data structures, we have `stackI`, which is $O(m)$, and `stackS`, which is $O(mrd)$, assuming that we may need to record information up to m levels. The total cost is $O(rd + t + t + rt + m + mrd)$, which is $O(rt + mrd)$. □

Theorem 13 *The worst-case time complexity of STR3 along a single path of length m in the search tree is $O(rt + m)$ per constraint.*

Proof: STR3's operations can be seen from the point of view of the three main data structures: `table`, `inv`, and `dep`. We consider them in this order:

- For a value a that stays present along a path, the cost of STR3 on $\texttt{table}(C, X, a)$ is $O(|\texttt{table}(C, X, a).\texttt{explored}|)$. If a is absent, there is an extra cost for merging the rest of $\texttt{table}(C, X, a)$ into $\texttt{inv}(C)$ (line 12), which is $O(|\texttt{table}(C, X, a)[1 \ldots \texttt{sep}]|)$. In both cases, the cost is $O(|\texttt{table}(C, X, a)|)$. The total cost is $O(\sum_{X \in scp(C),\ a \in D(X)} |\texttt{table}(C, X, a)|) = O(rt)$.
- The maximum size of $\texttt{inv}(C)$ is t. Because $\texttt{inv}(C)$ is implemented as a sparse set, adding a member takes $O(1)$ time. The size of $\texttt{inv}(C)$ can only grow along the path so the cost of STR3 in dealing with $\texttt{inv}(C)$ is $O(t)$.
- Lastly, we consider the cost associated with dependency lists. When the tuple $\texttt{tup}(C, k)$ becomes invalid, each element in $\texttt{dep}(C, k)$ is processed and shifted if necessary. For each value (X, a), it can be shifted around at most $|\texttt{table}(C, X, a)|$ times. Because each dependency list $\texttt{dep}(C, k)$ is

processed sequentially once along the path, the total cost is $O(\sum_{X \in scp(C),\ a \in D(X)} |\texttt{table}(C, X, a)|) = O(rt)$.

The worst-case time complexity of STR3 along a single path is thus $O(rt + t + rt + m) = O(rt + m)$, as all other statements have fixed costs ($O(1)$) at each node. □

As in the case of disjoint set union operations [CLRS09], a cascading series of transfers in dependency lists, where the number of elements to be moved keeps growing, is conceivable. This has no bearing on the complexity cost associated with dependency lists since we already show it to be $O(rt)$ through another argument, but it should be noted that the cascading cost is small due to the limited size of each list $\texttt{dep}(C, k)$, which holds no more than r elements. It follows that the worst-case cost in a monotonically increasing series of transfers is $O(\sum_{k=1}^{r-1} k) = O(r^2)$. Incidentally, the practical upper bound on operations involving dependency lists should be much closer to $O(rd)$ (the lower bound) than $O(rt)$ since absent values in dependency lists are never touched (line 21).

Property 1 *The worst-case time complexity along a single path of length m in the search tree can be as much as $O(rtm)$ for STR2 per constraint.*

Reasoning: Recall that STR2 improves over standard STR1 in two major ways. First, any (X, a) can be disregarded if $D^c(X)$ is fully supported. Second, no validity check is necessary for (X, a) if it is known that there is no change to the domain of X since the last time STR2 was called. Because STR2 is sensitive to ordering, we can build a table constraint and a search path such that (1) each call to STR2 involves a domain reduction of exactly one value on every domain, so that the second improvement is useless, (2) each call to STR2 eliminates exactly one tuple, which is found at the end of the table. As a result, the cost is $O(\sum_{i=1}^{m} r(t - i))$, which is $O(rtm)$ when $m << t$. □

It can be shown in a similar fashion that MDD^c [CY10] or tries [GJMN07] are not path-optimal. On the other hand, each backtrack costs $O(rd)$ in the worst-case for STR3, whereas it is $O(r)$ for STR2.

7.9 STR3 with Circular Seek

Gent [Gen13] reported that seeking an element in an array in a circular fashion during backtracking search can be proved optimal when the cost is amortized.

In other words, there is no theoretical difference with the traditional optimal procedure where a cursor's position is saved and restored upon backtrack. The circular approach is a factor two slower in the worst case, but experiments with SAT solvers show that it can be faster in practice [Gen13].

Because STR3 is based on incremental scans of the sub-tables, its cursors (separators) can be made to move circularly as well. The question is whether STR3 with circular seek remains optimal. This section addresses this variation of STR3 with respect to correctness, optimality, and performance.

Pseudo-code of STR3 with circular seek (STR3^{circ}) is given in Algorithm 25. Differences from ordinary STR3 are:

- d1 The search of supports performed from line 22 to 24 in Figure 23 is modified to accommodate circular seek (line 13 to 21 in Algorithm 25). The routine `restoreS` as well as the trailing of the separators at line 30 of Algorithm STR3 in Figure 23 are no longer needed.
- d2 The condition of the for-loop in line 12 of Algorithm STR3 in Figure 23 is changed. The result is shown in line 3 in Algorithm 25.

Theorem 14 *STR3^{circ} maintains GAC.*

Sketch of Proof: STR3^{circ}'s correctness can be derived from STR3's when their differences are accounted for as follows:

Case d1: The circular move guarantees that, unless a variable X is assigned to a value $b \neq a$, the only condition for the value a to be removed is the absence of support in the sub-table involving (X, a). Because separators are not maintained, it is always true that $(X, a) \in \texttt{dep}(C, k) \Leftrightarrow \texttt{table}(C, X, a).\texttt{sep} = k$. That is, separators and dependency lists are always synchronized. Invariant 6 holds as a result.

Case d2: In STR3, the subarray $\texttt{table}(C, X, a).\texttt{explored}$ always contains only *tids* of invalid tuples because the separators are not maintained. In STR3^{circ}, this is no longer true. Every *tid* in `table` has to be checked against $\texttt{inv}(C)$, but the effect on $\texttt{inv}(C)$ is the same as STR3's. □

Theorem 15 *The worst case time complexity of STR3^{circ} along a single path of length m in the search tree is $O(rt + m)$ per constraint.*

Proof: We need to look only at the operations involving `table` as the ones involving `dep` and `inv` are unchanged from STR3. For any value (X, a) that

Algorithm 25: STR3 with circular seek. Only the main procedure STR3^{circ} is shown.

```
1  STR3$^{circ}$($C$: Constraint, $X$: Variable, $a$: Value)
2    membersBefore $\leftarrow$ inv($C$).members
3    for $p \leftarrow 1$ to $|$table$(C, X, a)|$ do
4        $k \leftarrow$ table$(C, X, a)[p]$
5        if $\neg$isMember(inv($C$), $k$) then
6            addMember(inv($C$), $k$)
7    if membersBefore $=$ inv($C$).members then
8        return true
9    save($C$, membersBefore, stackI)
10   foreach $i \in \{$membersBefore $+ 1, \ldots,$ inv($C$).members$\}$ do
11       $k \leftarrow$ inv($C$).dense[$i$]
12       foreach $(Y, b) \in$ dep$(C, k)$ such that $b \in D^c(Y)$ do
13           $p \leftarrow$ table$(C, Y, b)$.sep
14           while $p > 0$ and isMember(inv($C$), table$(C, Y, b)[p]$) do
15               $p \leftarrow p - 1$
16           if $p = 0$ then
17               $p \leftarrow |$table$(C, Y, b)|$
18               while $p >$ table$(C, Y, b)$.sep and
                 isMember(inv($C$), table$(C, Y, b)[p]$) do
19                   $p \leftarrow p - 1$
20               if $p =$ table$(C, Y, b)$.sep then
21                   $p \leftarrow 0$
22           if $p = 0$ then
23               removeValue$(C, Y, b)$
24               if $D^c(Y) = \emptyset$ then return false
25           else
26               if $p \neq$ table$(C, Y, b)$.sep then
27                   table$(C, Y, b)$.sep $\leftarrow p$
28               move $(Y, b)$ from dep$(C, k)$ to dep$(C,$ table$(C, Y, b)[p])$
29   return true
```

stays present along a single path, the cost of circular scan on $\texttt{table}(C, X, a)$ is $O(|\texttt{table}(C, X, a)|)$ according to Theorem 13 in [Gen13]. If a is absent, there is an extra cost for merging $\texttt{table}(C, X, a)$ into $\texttt{inv}(C)$ (line 3 of Algorithm 25). The cost is $O(|\texttt{table}(C, X, a)| + |\texttt{table}(C, X, a)|) = O(|\texttt{table}(C, X, a)|)$ for either case. Summing on X and a makes the final complexity $O(rt + m)$, where the factor m includes other $O(1)$ costs at each node. □

Proposition 14 in [Gen13] indicates that overheads of the circular move can be a constant factor of two larger than the trailing of separators. For STR3^{circ}, there is a further cost of merging $\texttt{table}(C, X, a)$ for an absent value as mentioned in the proof above, which is another $O(rt)$ in total. Thus, the time complexity of STR3^{circ} hides a constant factor of three larger than STR3's. Our experiments show that STR3^{circ} is slower than both STR2 and STR3 on every problem instance tested.

7.10 Related Works

A number of fine-grained (G)AC algorithms have been proposed in the literature, for example, AC5 [HDmT92], AC6 [BC93], AC7 [BFR99], GAC4 [MM88], and AC5TC-Tr [MHD12]. In this section, we discuss about the connections existing between GAC4, AC5TC-Tr, and STR3 since they are all optimal and closely related algorithms.

7.10.1 GAC4

GAC4 [MM88] is a generalized version of AC4 [MH86] for non-binary constraints. The pseudocode for the propagation portion of GAC4 is given by Algorithm 26.

Algorithm 26: Algorithm GAC4

```
queue-processing($Q$: Queue)
  while $Q \neq \emptyset$ do
      pick and delete $(X, a)$ from $Q$
      foreach constraint $C$ such that $X \in scp(C)$ do
          foreach tuple $\tau \in \sup(C, X, a)$ do
              foreach $Y \in scp(C) \mid Y \neq X$ do
                  $b \leftarrow \tau[Y]$
                  remove $\tau$ from $\sup(C, Y, b)$
                  if $\sup(C, Y, b) = \emptyset$ and $b \in D^c(Y)$ then
                      $Q \leftarrow Q \cup \{(Y, b)\}$
                      $D^c(Y) \leftarrow D^c(Y) \setminus \{b\}$
```

GAC4's approach to propagation is based on an incremental reduction of the support lists; the list of supports of a value (X, a) on a constraint C is denoted by $\sup(C, X, a)$. When a value (X, a) becomes absent, every tuple τ involving that value becomes invalid. Because it may be a support for other values, $\sup(C, Y, \tau[Y])$ for every $Y \neq X$ must be updated as well. A value (X, a) has no longer a support on a constraint C when $\sup(C, X, a) = \emptyset$. It is then removed from $D^c(X)$ and put on the queue Q for further processing.

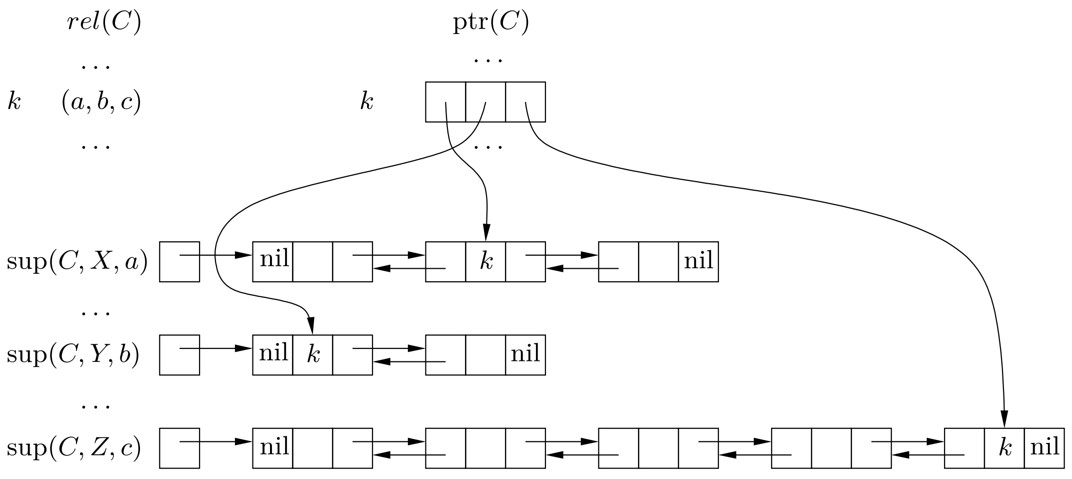


Figure 7.8: Data structures of GAC4 for a constraint C such that $scp(C) = \{X, Y, Z\}$.

In order to keep the time complexity optimal, it is proposed in [MM88] that the list of supports be implemented as a doubly linked list. Removing an element of a support list therefore takes $O(1)$ time. The use of an additional data structure containing pointers to elements of support lists is suggested so that these elements can also be accessed in $O(1)$ time — specifically, a two-dimensional array $\texttt{ptr}(C)$ of size $|rel(C)| \times |scp(C)|$. For every tuple $\texttt{tup}(C, k)$, $1 \leq k \leq |rel(C)|$, $\texttt{ptr}(C)[k]$ is a one-dimensional array of size r, where $scp(C) = \{X_1, \ldots, X_r\}$, such that for all $1 \leq i \leq r$, if $\texttt{tup}(C, k)[X_i] = a$ then $\texttt{ptr}(C)[k][i]$ points to a node in $\text{sup}(C, X_i, a)$ whose value is k. Figure 7.8 illustrates this with a ternary constraint C such that $scp(C) = \{X, Y, Z\}$.

Although attractive because admitting an optimal time complexity, GAC4 remains basically a standalone GAC algorithm. Indeed, no provision has been made for it to be used as a filtering step during backtracking search, although one can always make a simple adaptation by trailing all the relevant data structures, which are sup and `ptr` in this case. This incurs overheads, and it is unclear how costly the maintenance would be in practice since MGAC4 has never been reported in any experiment as far as we are aware. However, note that GAC4 remains useful when pre-caching of all supports is considered worthwhile. For instance, an extension of GAC4 is employed to find a form of interchangeable values in interactive configuration problems [BF13].

7.10.2 AC5TC-Tr

Several optimal GAC algorithms for table constraints are described in [MHD12, MHD14]. As a representative example, we focus our attention here on AC5TC-Tr [MHD12] (see [MHD14] for a comprehensive treatment of GAC algorithms for table constraints based on the AC5 structure [HDmT92].) We provide pseudocode of a key procedure of AC5TC-Tr in Algorithm 27 and explain its function as follows.

In AC5TC-Tr, a support list on C for a value (X, a) is dynamically maintained as a doubly linked list headed by $\mathit{FS}(C, X, a)$ ("first support") while its elements are connected via pointers `nextTr` and `predTr`. Both pointers satisfy the following invariants for every variable $X \in \mathit{scp}(C)$ and every tuple $\texttt{tup}(C, i)$ where $1 \leq i \leq t = |\mathit{rel}(C)|$:

$$\begin{aligned}
\texttt{nextTr}(C, X, i) &= \min\{j \mid i < j \;\; \wedge \;\; \texttt{tup}(C, j)[X] = \texttt{tup}(C, i)[X] \;\; \wedge \\
&\quad\;\; (\texttt{tup}(C, i) \in D(X, Q, C)) \Rightarrow \texttt{tup}(C, j) \in D(X, Q, C)\} \\
i &= \texttt{predTr}(C, X, \texttt{nextTr}(C, X, i))
\end{aligned}$$

$D(X, Q, C)$ denotes the "local view" of the domain $D(X)$ with respect to the propagation queue Q, defined as $D(X, Q, C) = D(X) \cup \{a \mid (C, X, a) \in Q\}$. For any $\mathcal{V} \subseteq \mathcal{X}$, $D(\mathcal{V}, Q, C) = \prod_{V \in \mathcal{X}} D(V, Q, C)$.

Furthermore, $\mathit{FS}(C, X, a)$, must satisfy the following invariant (a top value $\top$ is the largest value while a bottom $\perp$ is the smallest.)

$$\begin{aligned}
\mathit{FS}(C, X, a) = i \iff\;& i \neq \top \;\; \wedge \\
& \texttt{tup}(C, i)[X] = a \;\; \wedge \\
& \texttt{tup}(C, i) \in D(\mathit{scp}(C), Q, C) \;\; \wedge \\
& \forall j < i, \texttt{tup}(C, j)[X] = a \Rightarrow \texttt{tup}(C, j) \notin D(\mathit{scp}(C), Q, C)
\end{aligned}$$

`valRemoveTC-Tr` is a primary function of AC5TC-Tr that deals with the consequences of the removal of a value (X, a) with respect to a constraint C. When (X, a) becomes absent, the function goes through each tuple τ in the support list of (X, a) and updates the support list of each value (Y, b) that contains τ. The idea is fundamentally the same as MGAC4, but with the operations on doubly linked lists spelled out. One important difference is that the data structure `ptr` is not used. Indeed, although the use of `ptr` was put forward in [MM88] it is actually not necessary because tuples are already grouped along domain values of a given variable. That is, assuming a total order on *rel*(C), both `nextTr`(C, X, i) and `predTr`(C, X, i) return unique tuples (row positions). Thus next and previous tuples can be referred to directly through two pieces of information, namely, i (row) and X (column), as done in `valRemoveTC-Tr`.

Algorithm 27: Algorithm AC5TC-Tr

```
ValRemoveTc-Tr($C$: Constraint, $X$: Variable, $a$: Value)
  $\triangle \leftarrow \emptyset$
  $i \leftarrow FS(C, X, a)$
  while $i \neq \top$ do
     foreach $Y \in scp(C)$ such that $Y \neq X$ do
        $b \leftarrow$ tup$(C, i)[Y]$
        if $FS(C, Y, b) = i$ then
           $FS(C, Y, b) \leftarrow$ nextTr$(C, Y, i)$
           if $FS(C, Y, b) = \top$ and $b \in D(Y)$ then $\triangle \leftarrow \triangle \cup (Y, b)$
        else
           if predTr$(C, Y, i) \neq \bot$ then
              nextTr$(C, Y,$ predTr$(C, Y, i)) \leftarrow$ nextTr$(C, Y, i)$
           if nextTr$(C, Y, i) \neq \top$ then
              predTr$(C, Y,$ predTr$(C, Y, i)) \leftarrow$ predTr$(C, Y, i)$
     $i \leftarrow$ nextTr$(C, Y, i)$
```

STR3 and AC5TC-Tr achieve path-optimality through different routes. Both algorithms are based on the same concept of support lists, although STR3 works exclusively on row indexes. Traversals on the lists are analogous, whether through the doubly linked lists in AC5TC-Tr or through `table` in STR3. We discuss their differences in the rest of this section.

- *Indicators for the first valid supports.* The data structure *FS* plays the same role in AC5TC-Tr as `sep` does in STR3. Both mark the earliest valid support found according to the ordering in *rel*(C). Due to its representation, STR3 needs `dep` as a reverse pointer in addition to `sep`. Because `dep` does not need trailing, STR3 is able to sidestep some of AC5TC-Tr's efforts. However this notion can be replicated in AC5TC-Tr by adding a similar data structure, which in this case serves purely as residues [LLS$^+$08].

- *Overheads of doubly linked lists.* Besides the obvious extra traversal cost of doubly linked lists, the more important concern is its maintenance overheads during backtracks. AC5TC-Tr must keep track of the positions of all removed elements as well as the depth of the search when such removals happen. Should backtracks occur, AC5TC-Tr would use this information in order to restore these elements to their original positions. These removals may not be consecutive, and hence their restorations can be more expensive than the simple STR3's trailing mechanism.
- *Centralization of invalid tuples.* STR3 takes a lazy approach by partitioning a support list into two contiguous parts: an explored region known not to have any valid support, and an unexplored region. The two regions are separated by a cursor. As a result, when a tuple is proved invalid it must be remembered as such so that this fact can be recalled later when an unknown region is examined. To this end, STR3 channels all handling of invalid tuples through a centralized facility (`inv`). By contrast, AC5TC-Tr actively maintains support lists: when a tuple becomes invalid it is immediately removed from all involving support lists so it will not be encountered again in the future.
- *Local views and granularity of invariants.* AC5TC-Tr is derived from the AC5 framework, whose correctness stems in turn from invariants on its data structures. These invariants differ from STR3's in two respects. First, in AC5 they deal with local views instead of the current domains. Invariants must therefore hold even for values that are absent but still remain in the local views; for instance the invariant on FS must be maintained even for some absent values. By contrast, STR3 suspends all operations on a value once it becomes absent. Second, AC5's invariants hold at the point before and after each value (X, a) is dequeued and processed. STR3's invariants, on the other hand, are coarser. During search they hold at the point before and after STR3 is completely executed, not during its execution where the filtering may not have yet converged to a fixed point.

Finally, STR3 can be regarded as an improved version of AC5TC-Sparse (previously called AC5TC-Cutoff in [MHD12]) where an additional data structure is introduced for recording supports, which avoids restoring and checking operations to some extent.

7.11 Experimental Results

In order to show the practical interest of STR3, we have conducted an experimentation (with our solver AbsCon) using a cluster of bi-quad cores Xeon processors at 2.66 GHz with 16GiB of RAM under Linux. We have compared STR3, first with other classical STR variants, and then with other related GAC algorithms developed for table constraints. For all our experiments, we have used the backtrack search algorithm called MAC, which maintains (G)AC during search, equipped with the variable ordering heuristic *dom/ddeg* [SF94a] and the value ordering heuristic *lexico*. For each tested problem instance, we have searched to find a solution or prove that no solution exists, within $1,200$ seconds. It is important to note that the two chosen heuristics guarantee that we explore the very same search tree regardless of the filtering algorithm used (contrary to, for example, *dom/wdeg* [BHLS04]).

7.11.1 STR3 versus STR2: Comparison on Problem Series

Because it has been shown that STR2 is state-of-the-art on many series of instances [Lec11], we have compared the respective behavior of STR3 and STR2. Also included as a baseline are the results obtained with the original STR algorithm, as proposed by Ullmann [Ull07] and referred to as STR1 here.

First, we have considered some classical series of instances[3] involving positive table constraints with arity greater than 2. We give a brief description of these series:

- The Crossword puzzle involves filling a blank grid using words from a specified dictionary. We have used four series of instances, called *crosswords-lex*, *crosswords-uk*, *crosswords-words* and *crosswords-ogd*, which have been generated from a set of grids without any black square, called *Vg*, and four dictionaries, called *lex*, *uk*, *words* and *ogd*. Dictionaries *lex* and *words* are small whereas *uk* and *ogd* are large. The arity of the constraints is given by the size of the grids: for example, *crosswords-ogd-5-6* involves table constraints of arity 5 and 6 (the grid being 5 by 6).

[3]Available at `http://www.cril.univ-artois.fr/CSC09` or `http://www.cril.fr/~lecoutre/benchmarks.html`

- A Renault Megane configuration problem, converted from symbolic domains to numeric ones, has been introduced in [AFM02]. The series *renault-mod* contains instances generated from the original one, after introducing a form of perturbation. Such instances involve domains containing up to 42 values and some constraints of large arity (8 to 10); the largest table contains about 50,000 6-tuples.
- A Nonogram is built on a rectangular grid and requires filling in some of the squares in the unique feasible way according to some clues given on each row and column. Such clues can be modeled with table constraints. Constraint have typically large arities as for a grid of size $r \times c$, we get r constraints of arity c and c constraints of arity r. The size of the tables vary accordingly the size of the grids and the specified clues: some tables only contain a few tuples whereas the largest ones may usually contain tens or hundreds of thousands of tuples. The series *nonogram* here corresponds to the instances introduced in [PQZ12].
- Table constraints can be naturally derived from BDDs and MDDs. Series *bdd-15-21* and *bdd-18-21* were introduced in [CY06]: the former contains instances with table constraints of arity 15 and size $2,713$ whereas the latter contain instances with table constraints of arity 18 and size 133. Series *mdd-7-25-05* and *mdd-7-25-09* contain instances with constraints of arity 7 derived from MDDs built in a post-order manner with a specified probability p that controls how likely a previously created sub-MDD will be reused [CY10]. For the first series (also called *mdd-half*), the probability p is 0.5 whereas it is 0.9 for the second one.
- Series denoted by *rand-r-n* stand for random instances where each instance involves n variables and some constraints of arity r. The series *rand-3-20*, *rand-5-12*, *rand-8-20* and *rand-10-60* will permit us to experiment on random instances with various arity (from arity 3 to 10).

The results that we shall present include the following metrics:

- CPU time (in seconds); note that the CPU time for STR3 includes the preprocessing step (in which STR2 is employed).
- memory (mem) usage in MiB.
- avgP (average proportion), which is the ratio “size of the current table” to “size of the initial table” averaged over all table constraints and over all

nodes of the search tree; this is a relative value.

- avgS (average size), which is the size of the current table averaged over all table constraints and over all nodes of the search tree; this is an absolute value.

In order to avoid some “noise” generated by very easy instances, we have decided to discard from our results all instances that were systematically solved within 3 seconds (when embedding any filtering algorithm).

Table 7.1 shows mean results per series. Following the name of each series is the number of tested instances, which corresponds to the number of instances that are not too easy (as mentioned above) and not too difficult (solved by MAC within $1,200$ seconds when using any of the three algorithms). A first observation is that STR3 requires on average up to two times more memory than STR2; more memory was expected, but this is much better than what worst-case complexity suggests. A second observation is that the results seem to vary widely. STR2 and STR3 are respectively the best approaches on different series: *nonogram*, *bdd-15-21*, *bdd-18-21*, *mdd-7-25-05* and *rand-8-20* for STR2; *crosswords-ogd*, *rand-3-20*, *rand-5-12*, and *rand-10-60* for STR3. On other series, the gap between STR2 and STR3 is less significant. Table 7.2 gives details on some representative instances.

What is interesting to note, when looking at Tables 7.1 and 7.2, is that there appears to be a correlation between the values of avgP and avgS and the ranking of STR2 and STR3: the higher the values of avgP and avgS are, the more competitive STR3 becomes. Note that instances in Table 7.2 are ranked according to the values of avgP (from 0.5% to $51,2\%$ for structured instances, and from 0.2% to $25,6\%$ for random instances) in order to make the transition more apparent. Intuitively, higher values of avgP and avgS also imply that there are fewer chances that the solver can reach deeper levels of the search tree, which in turn suggests a connection to unsatisfiability. To confirm this hypothesis, Table 7.3 divides crossword instances (all series taken together) according to satisfiability, an avgP threshold (pragmatically set to 10%) and avgS threshold (pragmatically set to $1,000$). Clearly, it appears that STR3 is the best approach when tables are not reduced too much in proportion and/or size (on average), contrary to STR2. For example, on the 46 Crossword instances for which avgS $< 1,000$, STR2 is about 20% speedier than STR3 whereas on the 39 Crossword instances for which avgS $\geq 1,000$, STR3 is about 40% speedier than STR2.

Table 7.1: Mean CPU time (in seconds) to solve instances from different series (a time-out of 1,200 seconds was set per instance) with MAC.

Series	#		STR1	STR2	STR3
crosswords-lex	18	CPU	19.9	12.8	**12.6**
(avgP=10.2% / avgS=328)		mem	137M	137M	144M
crosswords-ogd	18	CPU	97.4	40.0	**25.0**
(avgP=23.2% / avgS=6,070)		mem	123M	145M	236M
crosswords-uk	25	CPU	98.5	45.3	**43.1**
(avgP=18.9% / avgS=1,500)		mem	126M	141M	183M
crosswords-words	23	CPU	62.8	37.4	**36.5**
(avgP=11.9% / avgS=637)		mem	138M	138M	150M
renault-mod	27	CPU	31.1	23.4	**20.6**
(avgP=27.8% / avgS=139)		mem	153M	153M	179M
nonogram	44	CPU	18.7	**9.4**	18.2
(avgP=9.7% / avgS=767)		mem	361M	386M	734M
bdd-15-21	35	CPU	64.0	**19.8**	60.7
(avgP=6.7% / avgS=466)		mem	219M	224M	2,049M
bdd-18-21	35	CPU	23.9	**7.2**	142
(avgP=6.1% / avgS=3,547)		mem	181M	182M	1,043M
mdd-7-25-05	5	CPU	222	**130**	621
(avgP=0.8% / avgS=348)		mem	264M	265M	500M
mdd-7-25-09	9	CPU	154.0	**100**	105
(avgP=5.9% / avgS=2,351)		mem	266M	267M	508M
rand-3-20	50	CPU	122	93	**79**
(avgP=7.6% / avgS=221)		mem	143M	143M	159M
rand-5-12	50	CPU	59.8	38.9	**15.1**
(avgP=24.4% / avgS=3,048)		mem	259M	259M	485M
rand-8-20	18	CPU	25.2	**14.7**	24.8
(avgP=0.2% / avgS=191)		mem	221M	221M	379M
rand-10-60	19	CPU	352	191	**91.7**
(avgP=23.0% / avgS=11,750)		mem	248M	248M	457M

Next, we tried to push the limit of the Crossword benchmarks by using a new dictionary *crosswords-ogd08*, which is twice larger than *ogd* (itself the largest one among the four dictionaries mentioned previously). There are 807,624 words in *ogd08* versus 435,705 in *ogd*. Most instances are of extreme cases: 45 instances are timed-out in all three algorithms tested and 12 are trivially solved (finished by both STR2 and STR3 within 3 seconds). The remaining instances are shown in Table 7.4 in order of the grid size (i.e. arity). Here we can see clearly the transition from satisfiable instances with low avgP and avgS values to unsatisfiable instances with high avgP and avgS values. STR2 is faster on the

Table 7.2: Detailed results on selected instances, sorted by avgP. The symbol "-" indicates a timeout.

		STR1	STR2	STR3
Structured instances				
crosswords-ogd-6-9	CPU	13.2	**7.9**	25.7
(sat - avgP=0.5% - avgS=227)	mem	148M	148M	202M
mdd-7-25-05-2	CPU	228	**123**	532
(sat - avgP=1.2% - avgS=467)	mem	264M	264M	500M
bdd-15-21-7	CPU	82.1	**24.5**	114
(unsat - avgP=3.6% - avgS=254)	mem	220M	225M	2,063M
nonogram-143	CPU	11.7	**7.6**	18.5
(sat - avgP=5.4% - avgS=320)	mem	413M	422M	875M
crosswords-ogd-11-13	CPU	-	1,086	**762**
(unsat - avgP=10.7% - avgS=5,144)	mem	-	157M	294M
nonogram-65	CPU	64.6	17.3	**10.9**
(sat - avgP=19.4% - avgS=453)	mem	159M	163M	205M
crosswords-ogd-14-14	CPU	53.7	21.0	**12**
(unsat - avgP=31.8% - avgS=7,491)	mem	144M	145M	236M
renault-27	CPU	21.4	15.6	**11.3**
(unsat - avgP=51.2% - avgS=223)	mem	151M	151M	178M
Random instances				
rand-8-20-8	CPU	86.2	**46.8**	565
(sat - avgP=0.2% - avgS=175)	mem	222M	222M	380M
rand-3-20-1	CPU	22.7	17.3	**17.1**
(sat - avgP=4.6% - avgS=136)	mem	144M	144M	161M
rand-3-20-26	CPU	243	178	**135**
(sat - avgP=7.1% - avgS=205)	mem	144M	144M	159M
rand-3-20-18	CPU	63.5	51.1	**38.2**
(unsat - avgP=12.5% - avgS=349)	mem	143M	143M	158M
rand-5-12-26	CPU	55.2	31.7	**13.3**
(unsat - avgP=25.4% - avgS=3,167)	mem	259M	259M	486M
rand-10-60-5	CPU	319	114	**57.6**
(unsat - avgP=25.6% - avgS=13,141)	mem	248M	248M	457M

Table 7.3: Mean CPU time (in seconds) to solve Crossword instances (a time-out of 1, 200 seconds was set per instance) with MAC.

	#	STR1	STR2	STR3
sat	14	18.9	**11.5**	31.0
unsat	71	152	68.9	**52.8**
avgP $< 10\%$	34	93.9	**50.8**	56.7
avgP $\geq 10\%$	51	154	65.2	**44.3**
avgS $< 1,000$	46	44.3	**26.7**	32.1
avgP $\geq 1,000$	39	231	98.0	**69.5**

Table 7.4: Results on the crossword puzzles for the *ogd08* dictionary. The symbol "-" indicates a timing out.

		STR1	STR2	STR3
crosswords-ogd08-6-8	CPU	2.14	**1.78**	5.59
(sat - avgP=0.2% - avgS=143)	mem	349M	349M	409M
crosswords-ogd08-6-9	CPU	12.8	**8.51**	51.8
(sat - avgP=0.2% - avgS=157)	mem	341M	341M	477M
crosswords-ogd08-7-7	CPU	2.41	**1.87**	4.94
(sat - avgP=0.4% - avgS=238)	mem	273M	273M	341M
crosswords-ogd08-7-8	CPU	-	**888**	-
(sat - avgP=0.3% - avgS=187)	mem	-	409M	-
crosswords-ogd08-15-17	CPU	-	1174	**735**
(unsat - avgP=20.0% - avgS=3,657)	mem	-	341M	477M
crosswords-ogd08-15-19	CPU	414	152	**103**
(unsat - avgP=24.9% - avgS=3,908)	mem	273M	273M	417M
crosswords-ogd08-16-19	CPU	778	346	**287**
(unsat - avgP=14.4% - avgS=1,361)	mem	273M	273M	409M
crosswords-ogd08-16-20	CPU	226	96.4	**90.2**
(unsat - avgP=19.2% - avgS=1,712)	mem	273M	273M	409M

4 first instances while STR3 is faster on the 4 last instances.

While it is not immediately clear what factors are involved concerning table reduction, one thing is certain: we know that every time a variable X is assigned a value a, all but tuples involving a are removed from the table of any constraint involving X, making avgP for this constraint low as a result. We have seen a surprising number of benchmarks with tables that are virtually wiped out by simple tabular reduction (no more than a few percent of the initial tuples remained on average) and variable instantiation may play an outsized role in this regard. We introduce now a benchmark where the instantiation is localized and has minimal effect on overall table reduction. A pigeonhole

Table 7.5: Results on the augmented pigeonhole problems.

		STR1	STR2	STR3
ph-6-9	CPU	20.3	10.6	**8.4**
(unsat - avgP=65.2% -avgS=1,273K)	mem	551M	547M	1751M
ph-7-7	CPU	11.7	6.2	**4.8**
(unsat - avgP=54.7% -avgS=153K)	mem	65M	62M	290M
ph-7-8	CPU	73.9	37.6	**26.8**
(unsat - avgP=54.7% - avgS=919K)	mem	422	422M	1146M
ph-8-6	CPU	25.1	12.6	**5.1**
(unsat - avgP=47.0% - avgS=55,293)	mem	33M	33M	126M
ph-8-7	CPU	195.6	99.3	**71.1**
(unsat - avgP=47.0% - avgS=387K)	mem	207M	207M	631M
ph-9-5	CPU	44.3	23.4	**7.9**
(unsat - avgP=41.1% - avgS=13,479)	mem	18M	18M	40M
ph-9-6	CPU	389	201	**149**
(unsat- avgP=41.1% - avgS=108K)	mem	32M	32M	227M
ph-10-4	CPU	62.8	33.9	**17.3**
(unsat - avgP=36.6% - avgS=2,399)	mem	15M	15M	21M
ph-11-3	CPU	86.4	63.8	**39.5**
(unsat - avgP=32.9% - avgS=329)	mem	10M	10M	21M

problem of size k is composed of k variables, each with $\{0, \ldots, k-1\}$ domain, and any two variables are connected by a binary inequality constraint, making the problem unsatisfiable. Unlike most benchmarks, the pigeonhole problem allows its tables to be reduced gradually during search until a variable directly involved is instantiated. An augmented pigeonhole ph-k-j adds an extra j-ary table and j new variables for each of the k variables and chain them together. The extra variables have larger domains to prevent the variable ordering heuristic from picking them prematurely (i.e., heuristics involving domain size) and the extra tables are very loose. Variable heuristics would be steered toward picking variables from the unsatisfiable core as a first priority. Results are shown in Table 7.5, where, once again, it is clear that STR3 is efficient with high values of avgP and avgS.

7.11.2 STR3 versus STR2: Overall Comparison

We present now a few scatter plots to provide an overall insight of the respective behaviors of STR2 and STR3. We use a large set of $2,005$ instances including the series introduced earlier, and also:

- the series of instances introduced in [MHD14] that can be found at `http://becool.info.ucl.ac.be/resources/`,
- the series of instances introduced in [XY13],
- the series of instances introduced in [PQZ12] for problem *kakuro*.

In each plot, a dot represents an instance whose coordinates are defined by, on the horizontal axis, the CPU time required to solve the instance with STR2, and on the vertical axis, the CPU time required to solve the instance with STR3. Thus, every dot below the line $x = y$ corresponds to an instance solved more efficiently by STR3, and every dot above the line $x = y$ corresponds to an instance solved more efficiently by STR2.

Figure 7.9 depicts with a first scatter plot the results obtained with STR2 and STR3 (within MAC) on the full set of $2,005$ instances. This scatter plot confirms that STR2 and STR3 are complementary: evidently, there are series/instances where STR2 is faster than STR3 and other ones where STR3 is faster than STR2. Figures 7.10, 7.11 and 7.12 compare the performance of STR2 vs. STR3 with respect to satisfiability, value of avgP, and value of avgS. Finally, Figure 7.13 plots the relative efficiency of STR2 against STR3 with respect to the value of avgS, when considering the $2,005$ instances of our experimental study. On some cases, where tables remain large ($> 1,000$), STR3 can be up to 3.6x faster than STR2.

7.11.3 STR3 versus STR2: Comparison on Classes of Random Problems

In the following set of experiments, we focus on classes of random problems, starting with those that can be found at the phase transition. We have generated different classes of instances from Model RD [XBHL07]. Each generated class $(r, 60, 2, 20, t)$ contains instances involving 60 Boolean variables and 20 r-ary constraints of tightness t. Provided that the arity r of the constraints is greater than or equal to 8, Theorem 2 [XBHL07] holds: an asymptotic phase transition is guaranteed at the threshold point $t_{cr} = 0.875$. It means that the hardest instances are generated when the tightness t is close to t_{cr}. Figure 7.14a shows the mean CPU time required by MAC to solve 20 instances of each class $(13, 60, 2, 20, t)$ where t ranges from 0.8 to 0.96. On these instances of intermediate difficulty, we observe that STR3 is worse off than even

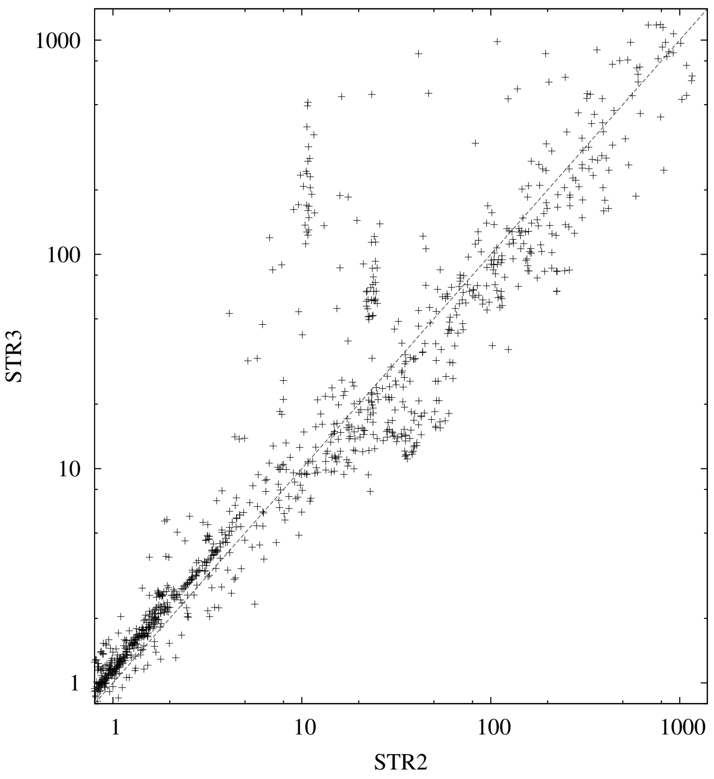
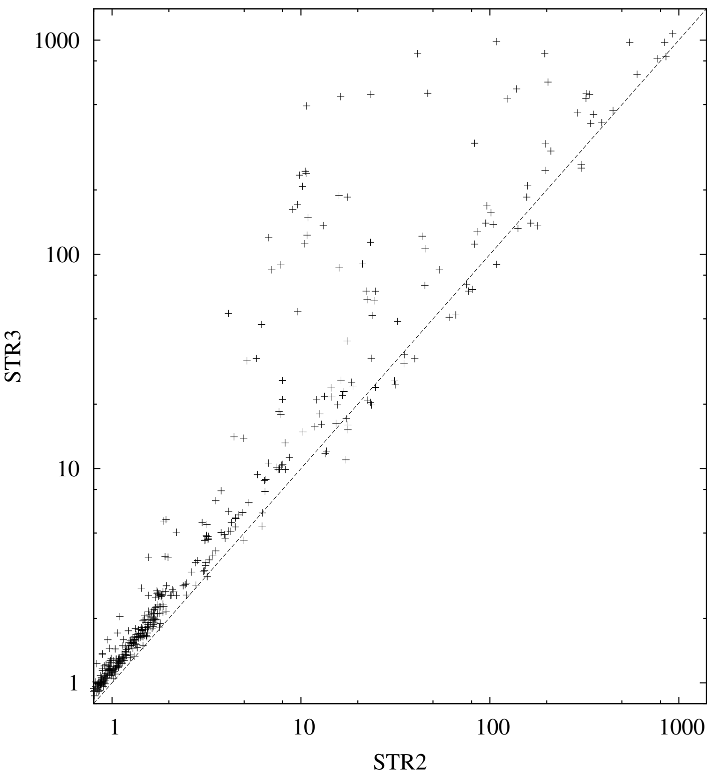


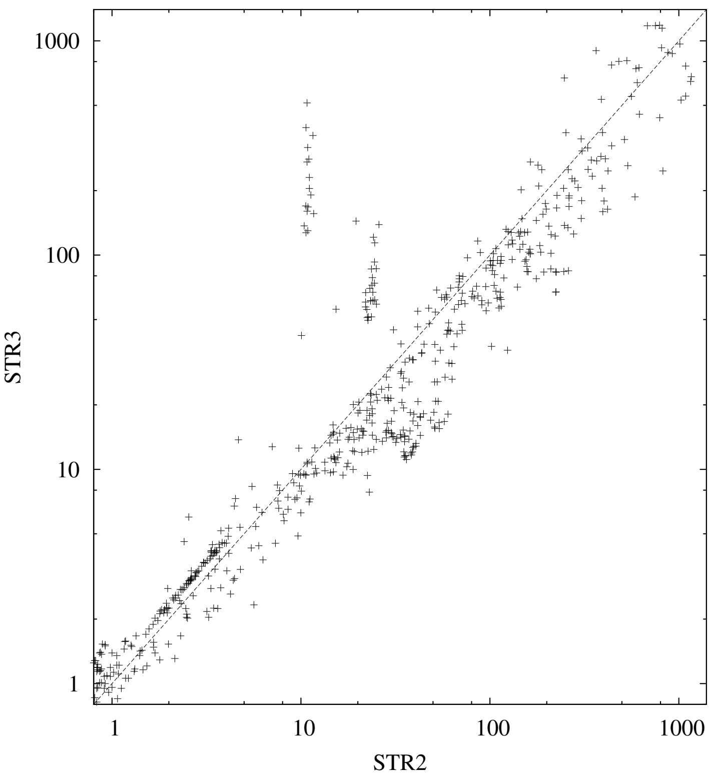
Figure 7.9: Pairwise comparison (CPU time) on 2,005 instances from many series involving table constraints. The time-out to solve an instance is 1,200 seconds.

STR1. Results on different classes of r follow the same pattern. Closer inspection reveals that avgP for this class is very low, especially at the phase transition where avgP is less than 4%.

For random problems, the metric avgP can be made higher when the instances that are generated do not lie in the phase transition area (therefore, there is no theoretical guarantee about their hardness). So, we have generated several classes of under-constrained instances. Each generated class $(5, 12, 12, 200, t)$ contains instances involving 12 variables with 12 possible values, and 200 constraints of arity 5 and tightness t. Figure 7.14b shows the mean CPU time required by MAC to solve 10 instances of each class $(5, 12, 12, 200, t)$ where t ranges from 0.51 to 0.99; the size of the tables ranges from $121,928$ (when $t = 0.51$) to $2,488$ (when $t = 0.99$). On these instances whose difficulty decreases with the tightness, we observe that STR3 is far better than STR2. Indeed, the values of both avgS and avgP are large: avgS ranges from $1,093$ to $8,170$ whereas avgP may reach up to 43%.

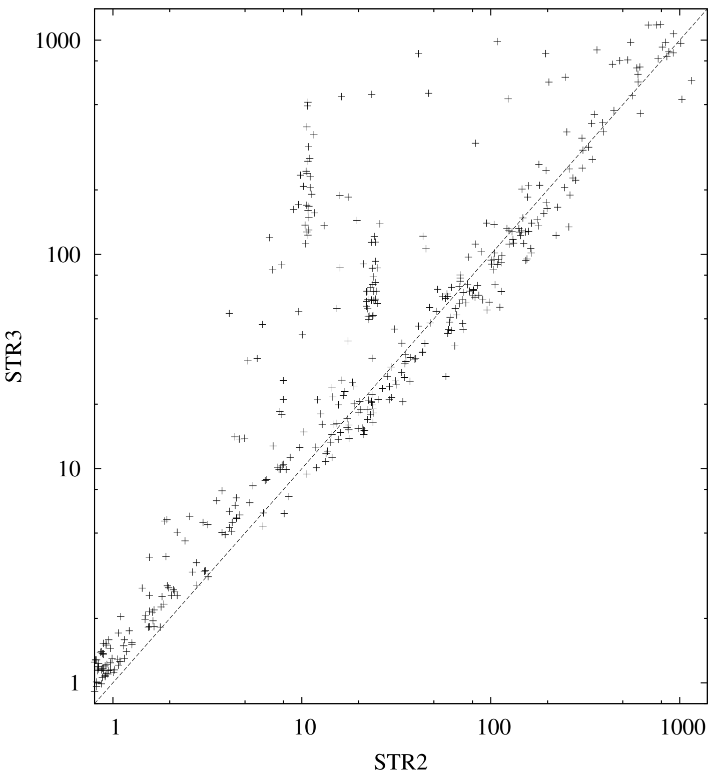


(a) Comparison on satisfiable instances

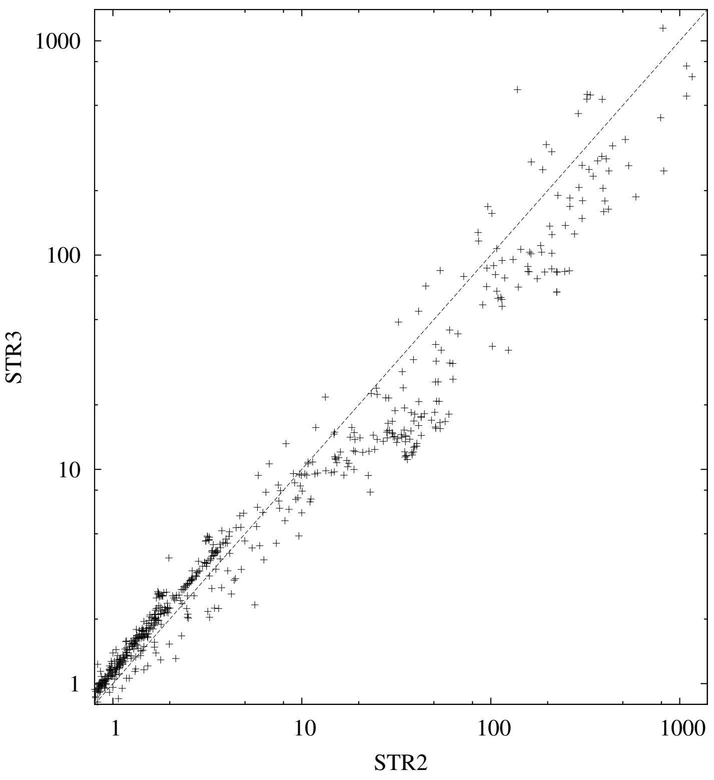


(b) Comparison on unsatisfiable instances

Figure 7.10: Comparison (CPU time) of STR2 and STR3 on instances that are respectively satisfiable and unsatisfiable.

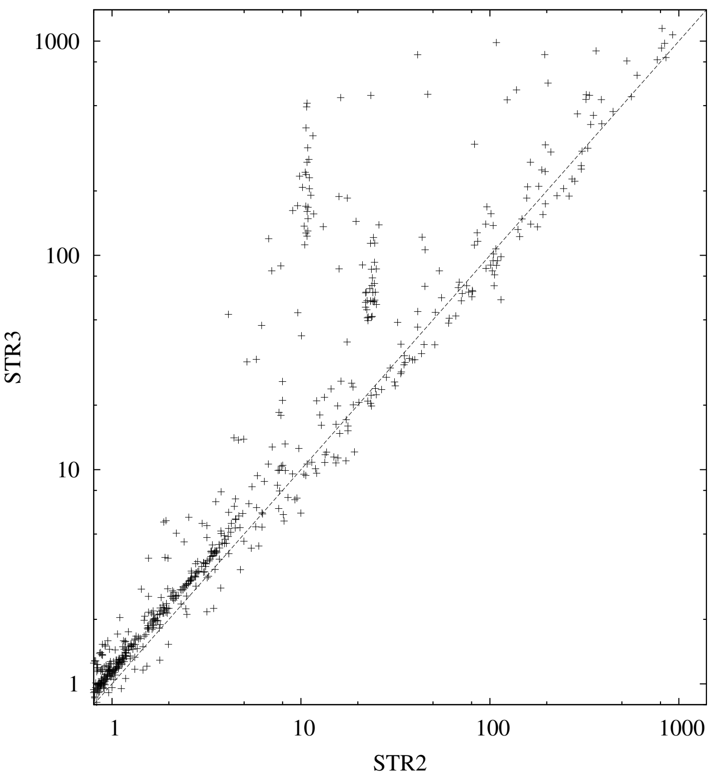


(a) Comparison on instances where avgP < 10%

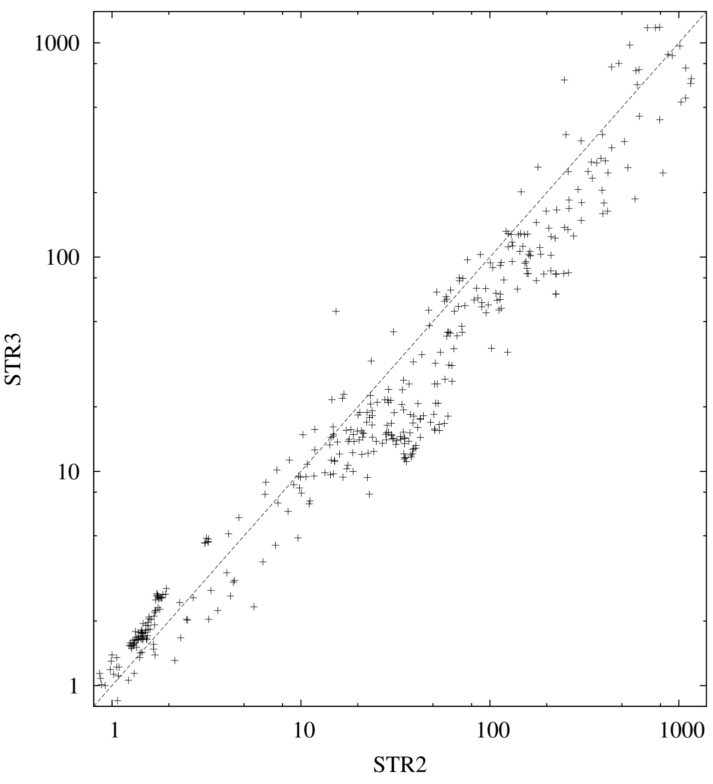


(b) Comparison on instances where avgP ≥ 10%

Figure 7.11: Comparison (CPU time) of STR2 and STR3 on instances where avgP is respectively less than and greater than 10%.



(a) Comparison on instances where avgS < 1000



(b) Comparison on instances where avgS ≥ 1000

Figure 7.12: Comparison (CPU time) of STR2 and STR3 on instances where avgS is respectively less than and greater than 1000.

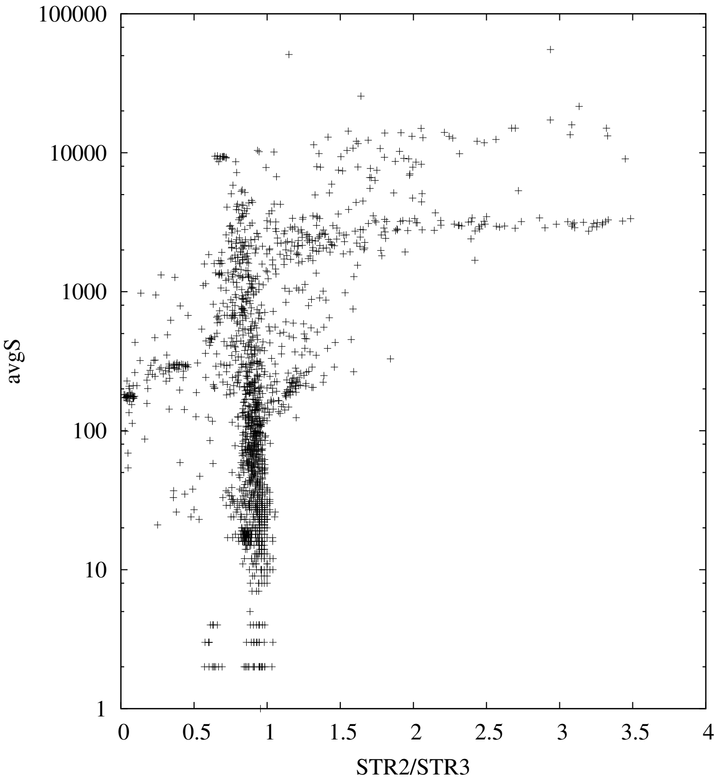
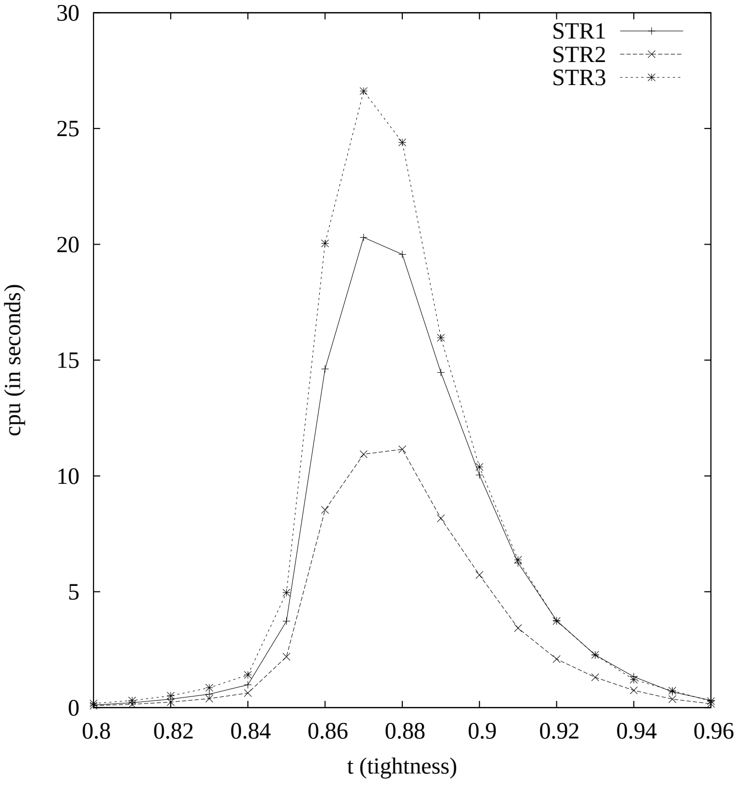


Figure 7.13: The ratio “cpu STR2” to “cpu STR3” is plotted against avgS (average size of tables during search). Dots correspond to instances solved by both algorithms within MAC (from the set of 2,005 instances used as our benchmark).

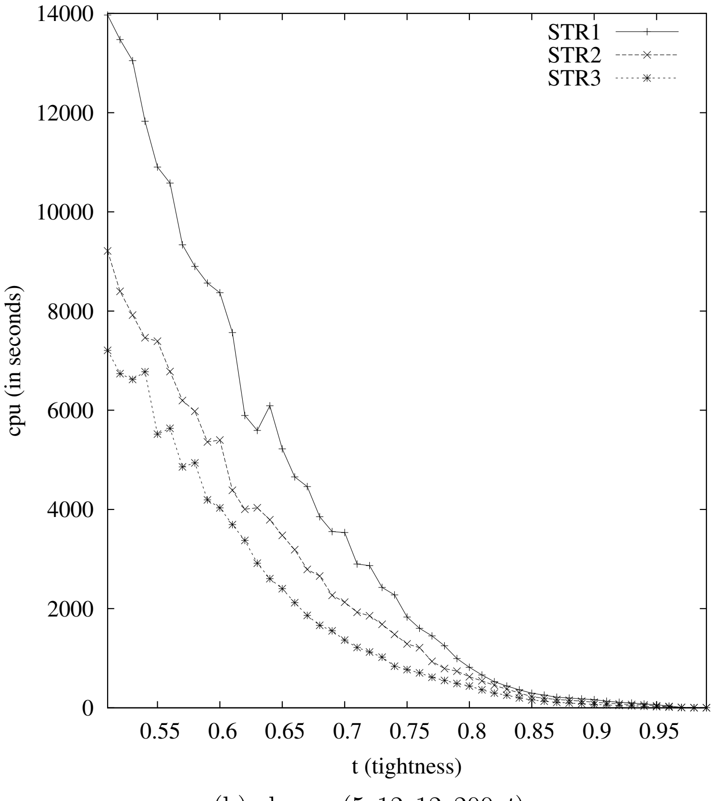
7.11.4 Comparison with Other GAC Algorithms

As mentioned in Section 7.10, STR3 and GAC4 are both optimal filtering algorithms. However, the overhead of maintaining the data structures of GAC4 during search can be significant. To confirm this, we have implemented GAC4 for it to be used within MAC by simply trailing its relevant data structures. Figure 7.15 depicts with a scatter plot the results obtained with STR3 and GAC4 (within MAC) on the full set of $2,005$ instances involving table constraints. This scatter plot clearly shows that GAC4 is largely outperformed by STR3. Note that many crosses appear on the right of the figure, at $x = 1,200$. They correspond to instances timed out by GAC4.

AC5TC, which has been proposed recently [MHD14] is also an optimal algorithm. However, the solver AbsCon that we use does not permit, in its current shape, to implement easily this type of algorithms. This is the reason why we present in Table 7.6 an excerpt (with the kind permission of the authors) of the results obtained by Mairy, Van Hentenryck and Deville with the



(a) classes $(13, 60, 2, 20, t)$



(b) classes $(5, 12, 12, 200, t)$

Figure 7.14: Mean search cost of solving instances in random classes with MAC.

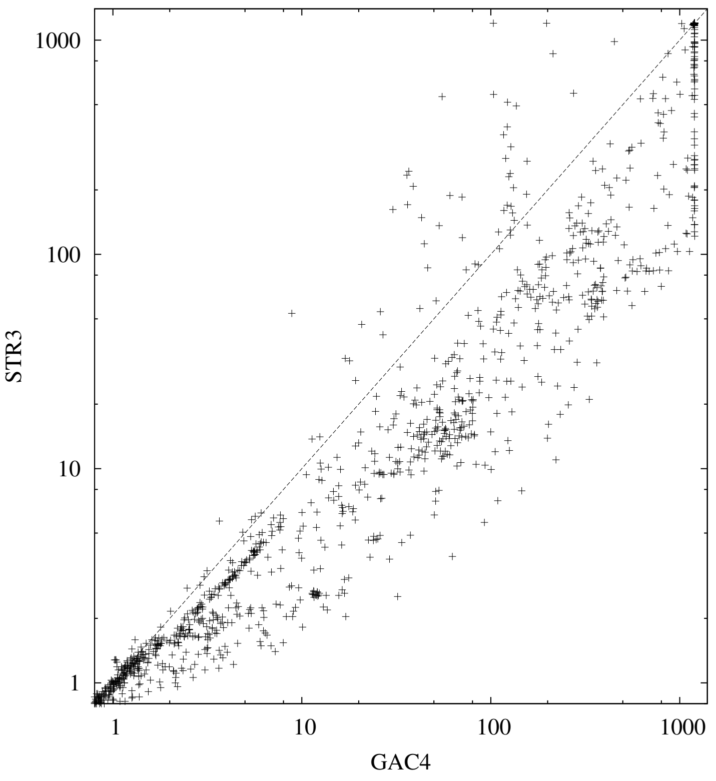


Figure 7.15: Pairwise comparison (CPU time) on 2, 005 instances from many series involving table constraints. The time-out to solve an instance is 1,200 seconds.

Table 7.6: Results obtained with AC5TC (version OptSparse) and MDD^c on classical series of instances involving table constraints of large arity. Values correspond to CPU times given as percentage to the best. This table is an excerpt from Table 10 in [MHD14].

	AC5TC	MDD^c	STR2	STR3
crosswords-lex	116	293	121	**100**
crosswords-ogd	249	704	162	**100**
crosswords-uk	247	713	135	**100**
crosswords-words	155	328	138	**100**
renault-mod	141	332	**100**	193

solver Comet. As mentioned in their chapter (and can be observed from Table 7.6), STR2, STR3 and MDD^c outperform AC5TC when table constraints have large arity (i.e., greater than 4). However, on constraints of arity 3 and 4, AC5TC is shown to be very fast (see details in [MHD14]).

The reader must be aware that compression-based filtering algorithms remain appropriate when compression is highly effective. This has been shown in

Table 7.7: Mean CPU time (in seconds) to solve selected binary problems.

	#	AC3^{bit+rm}	AC3^{rm}	MDD^{c}	STR2	STR3
blackhole-4-4	10	**0.91**	1.10	1.24	1.42	1.81
bqwh-18-141	100	**17.9**	20.6	33.5	45.8	71.1
composed	14	**47.0**	62.3	105	119	164
driver	7	**7.0**	8.5	22.7	56.5	211
ehi-85	90	**47.7**	63.7	72	127	340
frb-45-21r	10	**91**	168	290	300	317
geom-50-20	100	**3.63**	6.7	12.3	12.6	14.1
qcp-10	10	**13.1**	15.8	29.3	43	72.5
qwh-15	10	**2.06**	2.52	3.94	5.49	14.1
rand-2-40	698	**5.42**	7.4	21.9	18.9	18.8

[KW07, CY06, CY08, CY10, XY13]. Typically, when the compression ratio is high, an algorithm such as MDD^{c} outperforms STR algorithms.

There are also three well-known binary encodings of non-binary constraint networks, called dual encoding [DP89], hidden variable encoding [RPD90] and double encoding [SW99]. One could wonder whether or not a classical generic AC algorithm applied on such encodings could be competitive with simple tabular reduction. Actually, this has already been studied in [Lec11], with respect to STR2: it is shown in that chapter that the dual and the double encodings can rapidly run out of memory, and that STR2 is usually two or three times faster than AC3^{bit+rm} [LV08] and HAC [SW99] on the hidden variable encoding.

Finally, we would like to finish this presentation of experimental results with binary problems. Most of the recent works about filtering table constraints concentrate solely on non-binary constraints even though binary constraints in extensional form are tables too. Only after [MHD14] was published that it was made clear the compression-based and the STR methods are not competitive with generic binary AC algorithms such as AC3^{rm} [LH07] and AC3^{bit+rm} [LV08]. We confirm their findings with the results in Table 7.7.

7.12 Conclusions

We have introduced STR3, a new GAC algorithm for positive table constraints that is competitive and complementary to STR2, a state-of-the-art algorithm. STR3 is able to completely avoid unnecessary traversal of tables. Along with

AC5TC [MHD14], STR3 is one of the only two path-optimal GAC algorithms that have been reported so far. Unlike AC5TC's performance, which declines as arity increases, STR3's is consistent across a wide range of arity. Indeed, we have shown that it correlates to the average proportion (avgP) and number (avgS) of tuples remaining in tables during search. Compared to STR2, STR3 is faster on problems in which avgP and avgS are not low. Interestingly, the advantage of STR2 appears to depend largely on excessively high rates of table reduction (very low avgP). As soon as the reduction rate drops below 90%, STR2 becomes much less effective. Another dividing line is satisfiability: STR3 is stronger on unsatisfiable problems but weaker on satisfiable problems whereas STR2 is the opposite.

STR3 is an instance of fine-grained algorithms as its propagation is guided by deleted values. Both STR2 and STR3 are extended to handle compressed tuples (c-tuples) in [XY13]. While STR3 is more complex than STR2, once implemented the algorithm is easier to extend because it is based on just one notion: checking a tuple's validity; as a result, only the routine involving validity test needs modification. By contrast, STR2 may process a tuple twice in different manners — testing its validity and then collecting values from its components. Extending STR2 to cope with c-tuples is therefore twice more complicated conceptually. This is true for fine-grained vs. coarse-grained propagation in general. In recent years, STR2 has been incorporated into many other algorithms [LPS13, JN13, BFL13, GHLR14], and it remains to be seen whether STR3 can be adopted in a similar fashion. The work which extends STR2 and STR3 to c-tuples [XY13] is a start in this direction.

References

[ABC$^+$76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[AFM02] J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic CSPs - application to configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002.

[BC93] Christian Bessière and Marie-Odile Cordier. Arc-consistency and arc-consistency again. In *Proceedings of AAAI-93*, pages 108–113, Washington, DC, USA, 1993.

[BC94] N. Beldiceanu and E. Contejean. Introducing global constraints in chip. *Mathl. Comput. Modelling*, 12(20):97–123, 1994.

[BCFR04] J Christopher Beck, Tom Carchrae, Eugene C. Freuder, and Georg Ringwelski. Backtrack-free search for real-time constraint satisfaction. In *Proceedings of CP-04*, Toronto, Canada, 2004.

[BCH$^+$94] Amit Bellicha, Christian Capelle, Michel Habib, Tiborr Kokeny, and Marie-Catherine Vilarem. Csp techniques using partial orders on domains values. In *ECAI-04 Workshop on constraint satisfaction issues raised by practical applications*, pages 47–56, Amsterdam, The Netherlands, 1994.

[BCM08] L. Bordeaux, M. Cadoli, and T. Mancini. A unifying framework for structural properties of CSPs: Definitions, complexity, tractability. *JAIR*, 32:607–629, 2008.

[BCvBW02] F. Bacchus, X. Chen, P. van Beek, and T. Walsh. Binary vs. non-binary constraints. *Artificial Intelligence*, 140(1–2):1–37, 2002.

[BCZ01] Amy M. Beckwith, Berthe Y. Choueiry, and Hui Zou. How the level of interchangeability embedded in a finite constraint satisfaction problem affects the performance of search. In *Proceedings of the 14th Australian Joint Conference on AI*, pages 50–61, 2001.

[BD05] Christian Bessiere and Romuald Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI-05*, Edinburgh, U.K., 2005.

[BF92] Brent W. Benson and Eugene C. Freuder. Interchangeability preprocessing can improve forward checking search. In *Proceedings of ECAI-92*, pages 28–30, Vienna, Austria, 1992.

[BF13] Caroline Becker and Hélène Fargier. Maintaining alternative values in constraint-based configuration. In *Proceedings of IJCAI-13*, pages 454–460, Beijing, China, 2013.

[BFL13] C. Bessiere, H. Fargier, and C. Lecoutre. Global inverse consistency for interactive constraint satisfaction. In *Proceedings of CP-13*, pages 159–174, 2013.

[BFR99] Christian Bessière, Eugene C. Freuder, and Jean-Charles Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107(1):125–148, 1999.

[BHLS04] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, Valancia, Spain, 2004.

[BL03] James Bowen and Chavalit Likitvivatanavong. Splitting the atom: A new approach to neighbourhood interchangeability in constraint satisfaction problems. In *Proceedings of IJCAI-03*, pages 1366–1367, Acapulco, Mexico, 2003.

[BL04] James Bowen and Chavalit Likitvivatanavong. Domain transmutation in constraint satisfaction problems. In *Proc. of AAAI-04*, 2004.

[BMR$^+$99] Stefano Bistarelli, Ugo Montanari, Francesca Rossi, Thomas Schiex, Gerard Verfaillie, and Helene Fargier. Semiring-based csps and valued csps: Frameworks, properties, and comparison. *Constraints*, 4(3):199–240, 1999.

[BR97] Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI-97*, pages 398–404, Nagoya, Japan, 1997.

[BRYZ05] Christian Bessière, Jean-Charles Régin, Roland H. C. Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2), 2005.

[BT93] Preston Briggs and Linda Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1–4):59–69, 1993.

[CD02] Berthe Y. Choueiry and Amy M. Davis. Dynamic bundling: Less effort for more solutions. In *Proceeding of the 5th International Symposium on Abstraction, Reformulation and Approximation (SARA)*, pages 64–82, 2002.

[CDE15] Martin C. Cooper, Aymeric Duchein, and Guillaume Escamocher. Broken triangles revisited. In *Proceedings of CP-15*, 2015.

[cdsmSSL13] V. Le clement de saint marcq, P. Schauss, C. Solnon, and C. Lecoutre. Sparse-sets for domain implementation. In *Techniques foR Implementing Constraint programming Systems (TRICS'13)*, 2013.

[CE12] Martin C. Cooper and Guillaume Escamoche. A dichotomy for 2-constraint forbidden CSP patterns. In *Proceedings of AAAI-12*, 2012.

[CJS10] M. C. Cooper, P. G. Jeavons, and A. Z. Salamon. Generalizing constraint satisfaction on trees: Hybrid tractability and variable elimination. *AIJ*, 9–10(174):570–584, 2010.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 2009.

[CMTZ14] Martin C. Cooper, Achref El Mouelhi, Cyril Terrioux, and Bruno Zanuttini. On broken triangles. In *Proceedings of CP-14*, pages 9–24, 2014.

[CN98] Berthe Y. Choueiry and Guevara Noubir. On the computation of local interchangeability in discrete constraint satisfaction problems. In *Proceedings of AAAI-98*, pages 326–333, Madison, Wisconsin, 1998.

[Coo14] Martin C. Cooper. Beyond consistency and substitutability. In *Proceedings of CP-14*, pages 256–271, 2014.

[CS03] Assef Chmeiss and Lakhdar Saïs. About neighborhood substitutability in CSPs. In *CP-03 Workshop on Symmetry in Constraint Satisfaction Problems*, Kinsale, Ireland, 2003.

[CY06] Kenil C. K. Cheng and Roland H. C. Yap. Maintaining generalized arc consistency on ad hoc n-ary boolean constraints. In *Proceedings of ECAI-06*, Trento, Italy, 2006.

[CY08] Kenil C. K. Cheng and Roland H. C. Yap. Maintaining generalized arc consistency on ad hoc r-ary constraints. In *Proceedings of CP-08*, Sydney, Australia, 2008.

[CY10] Kenil C. K. Cheng and Roland H. C. Yap. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2):265–304, 2010.

[DB01] Romuald Debruyne and Christian Bessière. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, (14):205–230, 2001.

[DP87] Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34(1):1–38, 1987.

[DP89] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38(3):353–366, 1989.

[FD94] Daniel Frost and Rina Dechter. Dead-end driven learning. In *Proceedings of AAAI-94*, pages 294–300, Seattle, Washington, 1994.

[FE96] Eugene C. Freuder and Charles D. Elfe. Neighborhood inverse consistency preprocessing. In *Proceedings of AAAI-96*, pages 202–208, Portland, Oregon, 1996.

[FH95] Eugene C. Freuder and Paul D. Hubbe. Extracting constraint satisfaction subproblems. In *Proceedings of IJCAI-95*, pages 548–555, Montreal, Canada, 1995.

[Fre91] Eugene C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proc. of AAAI-91*, 1991.

[Fre11] Eugene C. Freuder. Dispensable instantiations. In *CP-11 Workshop on Constraint Modelling and Reformulation*, 2011.

[FW92] Eugene C. Freuder and Richard J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 1–3, 1992.

[Gec15] Gecode Team. Gecode: Generic constraint development environment, 2015. Available from `http://www.gecode.org`.

[Gen13] Ian P. Gent. Optimal implementation of watched literals and more general techniques. *JAIR*, 48:231–252, 2013.

[GHLR14] N. Gharbi, F. Hemery, C. Lecoutre, and O. Roussel. Sliced table constraints: Combining compression and tabular reduction. In *Proceedings of CPAIOR-14*, 2014.

[GJMN07] Ian P. Gent, Chris Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of AAAI-07*, pages 191–197, Vancouver, Canada, 2007.

[GMP$^+$01] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh. Random constraint satisfaction: Flaws and structure. *Constraints*, 6(4):345–372, 2001.

[GPR98] Matthew L. Ginsberg, Andrew J. Parkes, and Amitabha Roy. Supermodels and robustness. In *Proceedings of AAAI-98*, pages 334–339, Madison, Wisconsin, 1998.

[GSL10] Graeme Gange, Peter J. Stuckey, and Vitaly Lagoon. Fast set bounds propagation using a BDD-SAT hybrid. *JAIR*, 38:307–338, 2010.

[GSS11] Graeme Gange, Peter J. Stuckey, and Radoslaw Szymanek. MDD propagators with explanation. *Constraints*, 16(4):407–429, 2011.

[Has93] Alois Haselböck. Exploiting interchangeabilities in constraint satisfaction problems. In *Proc. of IJCAI-93*, pages 282–287, 1993.

[HC88] P. Van Hentenryck and J.-P. Carillon. Generality vs. specificity: an experience with AI and OR techniques. In *Proceedings of AAAI-88*, pages 40–45, 1988.

[HDmT92] Pascal Van Hentenryck, Yves Deville, and Choh man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2-3):291–321, 1992.

[HF92] Paul D. Hubbe and Eugene C. Freuder. An efficient cross-product representation of the constraint satisfaction problem search space. In *Proc. of AAAI-92*, pages 421–427, 1992.

[IBM14] IBM ILOG Team. Ilog optimization suites, 2014. Available from `http://www-01.ibm.com/software/info/ilog/`.

[JaC15] JaCoP Team. Jacop - java constraint programming solver, 2015. Available from `http://jacop.osolpro.com`.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.

[JN13] Christopher Jefferson and Peter Nightingale. Extending simple tabular reduction with short supports. In *Proceedings of IJCAI-13*, pages 573–579, Beijing, China, 2013.

[KW07] George Katsirelos and Toby Walsh. A compression algorithm for large arity extensional constraints. In *Proc. of CP-07*, 2007.

[KWC$^+$10] Shant Karakashian, Robert Woodward, Berthe Y. Choueiry, Steven Prestwich, and Eugene C. Freuder. A partial taxonomy of substitutability and interchangeability. In *CP-10 Workshop on Symmetry in Constraint Satisfaction Problems*, 2010.

[LC05] Christophe Lecoutre and Stephane Cardon. A greedy approach to establish singleton arc consistency. In *Proceedings of IJCAI-05*, pages 199–204, Edinburgh, U.K., 2005.

[LCF05] Anagh Lal, Berthe Y. Choueiry, and Eugene C. Freuder. Neighborhood interchangeability and dynamic bundling for non-binary finite CSPs. In *Proceedings of AAAI-05*, pages 397–404, 2005.

[Lec11] C. Lecoutre. STR2: Optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371, 2011.

[Les94] David Lesaint. Maximal sets of solutions for constraint satisfaction problems. In *Proceedings of ECAI-94*, pages 110–114, Amsterdam, The Netherlands, 1994.

[LH07] C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proceedings of IJCAI'07*, pages 125–130, Hyderabad, India, 2007.

[LLS$^+$08] 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.

[LLY12] Christophe Lecoutre, Chavalit Likitvivatanavong, and Roland H. C. Yap. A path-optimal GAC algorithm for table constraints. In *Proceedings of ECAI-12*, France, 2012.

[LPS13] C. Lecoutre, A. Paparrizou, and K. Stergiou. Extending STR to a higher-order consistency. In *Proc. of AAAI-13*, pages 576–582, 2013.

[LR05] Olivier Lhomme and Jean-Charles Régin. A fast arc consistency algorithm for n-ary constraints. In *Proceedings of AAAI-05*, Pittsburgh, Pennslyvania, 2005.

[LS06] Christophe Lecoutre and Radoslaw Szymanek. Generalized arc consistency for positive table constraints. In *Proceedings of CP-06*, pages 284–298, 2006.

[LV08] Christophe Lecoutre and Julien Vion. Enforcing arc consistency using bitwise operations. *Constraint Programming Letters*, 2:21–35, 2008.

[LY08] Chavalit Likitvivatanavong and Roland H. C. Yap. A refutation approach to neighborhood interchangeability in CSPs. In *Proc. of Australasian Joint Conference on AI*, 2008.

[LY13] Chavalit Likitvivatanavong and Roland H. C. Yap. Eliminating redundancy in CSPs through merging and subsumption of domain values. *ACM SIGAPP Applied Computing Review*, 13(2), 2013.

[LZBF04] Chavalit Likitvivatanavong, Yuanlin Zhang, James Bowen, and Eugene C. Freuder. Arc consistency in MAC: A new perspective. In *CP-04 Workshop on Constraint Propagation And Implementation*, Toronto, Canada, 2004.

[LZS$^+$07] Chavalit Likitvivatanavong, Yuanlin Zhang, Scott Shannon, James Bowen, and Eugene C. Freuder. Arc consistency during search. In *Proceedings of IJCAI-07*, pages 137–142, Hyderabad, India, 2007.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[MH86] Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225–233, 1986.

[MHD12] Jean-Baptiste Mairy, Pascal Van Hentenryck, and Yves Deville. An optimal filtering algorithm for table constraints. In *Proceedings of CP-12*, pages 496–511, Canada, 2012.

[MHD14] Jean-Baptiste Mairy, Pascal Van Hentenryck, and Yves Deville. Optimal and efficient filtering algorithms for table constraints. *Constraints*, 19(1):77–120, 2014.

[MLB01] S. Merchez, C. Lecoutre, and F. Boussemart. AbsCon: a prototype to solve CSPs with abstraction. In *Proc. of CP-01*, pages 730–744, 2001.

[MM88] Roger Mohr and Gérald Masini. Good old discrete relaxation. In *Proceedings of ECAI-88*, pages 651–656, Germany, 1988.

[MMZ$^+$01] M. Moskewisz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *The 39th Design Automation Conference*, 2001.

[Mon74] Ugo Montanari. Networks of constraints: fundamental properties and applications to picture processing. *Information Sciences*, (7):95–132, 1974.

[NSB+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *Proceedings of the CP-07*, pages 529–543, 2007.

[O'S02] Barry O'Sullivan. Interactive constraint-aided conceptual design. *Journal of Artificial Intelligence for Engineering Design Analysis and Manufacturing (AIEDAM)*, 16(4), 2002.

[Pes04] G. Pesant. A regular language membership constraint for finite sequences of variables. In *Proceedings of CP-04*, pages 482–495, 2004.

[PFL14] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014.

[PQZ12] G. Pesant, C.-G. Quimper, and A. Zanarini. Counting-based search: Branching heuristics for constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 43:173–210, 2012.

[R9́6] Jean-Charles Régin. Genaralized arc consistency for global cardinality constraint. In *Proceedings of AAAI-96*, pages 209–215, 1996.

[R1́1] J-C. Régin. Improving the expressiveness of table constraints. In *CP-11 Workshop on Constraint Modelling and Reformulation*, 2011.

[Rég94] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI-94*, pages 362–367, 1994.

[RPD90] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proceedings of ECAI'90*, pages 550–556, Stockholm, Sweden, 1990.

[SF94a] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP-94*, pages 10–20, Seattle WA, 1994.

[SF94b] Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of ECAI-94*, pages 125–129, Amsterdam, The Netherlands, 1994.

[SHWK76] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. The design and implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.

[SS05] N. Samaras and K. Stergiou. Binary encoding of non-binary constraint satisfaction problems: Algorithms and experimental results. *JAIR*, 24:641–684, 2005.

[SW99] K. Stergiou and T. Walsh. Encodings of non-binary constraint satisfaction problems. In *Proceedings of AAAI'99*, pages 163–168, Orlando, Florida, 1999.

[Ull07] J. R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Science*, 177(18):3639–3678, 2007.

[vBD95] Peter van Beek and Rina Dechter. On the minimality and global consistency of row-convex constraint networks. *Journal of the ACM*, 42(3):543–561, 1995.

[WF99] Rainer Weigel and Boi V. Faltings. Compiling constraint satisfaction problems. *Artificial Intelligence*, 115(2):257–287, 1999.

[WFC96] Rainer Weigel, Boi V. Faltings, and Berthe Y. Choueiry. Context in discrete constraint satisfaction problems. In *Proceedings of ECAI-96*, pages 205–213, Budapest, Hungary, 1996.

[XBHL07] Ke Xu, Frederic Boussemart, Fred Hemery, and Christophe Lecoutre. Random constraint satisfaction: easy generation of hard (satisfiable) instances. *AIJ*, 171(8–9):514–534, 2007.

[XY13] Wei Xia and Roland H. C. Yap. Optimizing STR algorithm with tuple compression. In *Proceedings of CP-13*, pages 724–732, Sweden, 2013.

[YDIK92] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proc. of ICDCS-92*, 1992.

[Yok94] Makoto Yokoo. Weak-commitment search for solving constraint satisfaction problems. In *Proceedings of AAAI-94*, pages 313–318, Seattle, Washington, 1994.