A COMPARATIVE STUDY OF GENERALIZED ARC-CONSISTENCY ALGORITHMS

by

Olufikayo S. Adetunji

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Berthe Y. Choueiry

Lincoln, Nebraska

December, 2014

A COMPARATIVE STUDY OF GENERALIZED ARC-CONSISTENCY ALGORITHMS

Olufikayo S. Adetunji, M. S.

University of Nebraska, 2014

Adviser: Berthe Y. Choueiry

In this thesis, we study several algorithms for enforcing Generalized Arc-Consistency (GAC), which is the most popular consistency property for solving Constraint Satisfaction Problems (CSPs) with backtrack search. The popularity of such algorithms stems from their relative low cost and effectiveness in improving the performance of search. Virtually all commercial and public-domain constraint solvers include some implementation of a generic GAC algorithm. In recent years, several algorithms for enforcing GAC have been proposed in the literature that rely on increasingly complex data structures and mechanisms to improve performance. In this thesis, we study, assess, and compare a basic algorithm for generic constraints (i.e, GAC2001), new algorithms for table constraints (i.e., STR1, STR2, STR3, eSTR1, eSTR2, and STR-Ni), and an algorithm for constraints expressed as multi-valued decision diagram (i.e., mddc). We explain the mechanisms of the above algorithms, and empirically evaluate and compare their performances. We propose a new hybrid algorithm that uses a selection criterion to combine the use of STR1 and STR-Ni.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Berthe Y. Choueiry, for her continued support and encouragement. I was really inspired by working with her. My first experience with the area of Constraint Processing was in her class. My good performance in that course and deep understanding of the material owe to her help and that of the graduate teaching assistant, Robert Woodward. I am grateful to the entire faculty of the Department Computer Science and Engineering at UNL, through whom I have learned a lot and have grown both intellectually and in confidence.

I am also thankful to the members of the committee, Dr. Ashok Samal and Dr. Stephen D. Scott for their invaluable feedback that allowed me to improve the content and presentation of my thesis. I am particularly thankful to the members of the Constraint Systems Laboratory, Robert Woodward, Anthony Schneider, and Daniel Geschwender for their constant support and immediate help when I encountered difficulties. I want to thank especially Robert Woodward for always being available to help me understand mechanisms that were not clear. I am also grateful to Shant Karakashian, Robert Woodward, and Anthony Schneider for creating an extensive, flexible, and reliable solver framework that I exploited to build, test, and evaluate the algorithms that I studied. The search and GAC2001 algorithms used in my thesis come from their solver.

Finally, I am grateful to my loving family, who encouraged me to pursue my passion for Computer Science. They have provided me with their unwavering support both financially and spiritually. I am especially grateful to God because without Him, I will not be the person I am today.

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Constraint Satisfaction Problems (CSPs) are used to model many decision problems of practical importance. Examples include scheduling and resource allocation (e.g., school time-table and airport gate scheduling), games (e.g., Sudoku, Minesweeper), databases, product configuration and design, and natural language processing.

1.1 Motivation

Enforcing a given level of consistency and constraint propagation are central to the area of Constraint Processing (CP).[1] Algorithms for enforcing consistency and propagating constraints allow us to effectively reduce the cost of solving CSPs with backtrack search, which is exponential in the size of the problem. Such algorithms are typically efficient (i.e., polynomial) in both time and space. The most basic mechanism, which enforces *Generalized Arc Consistency* (GAC), is indeed at the foundation of CP [Waltz, 1975; Mackworth, 1977].

For many the restriction to binary CSPs (i.e., constraints apply to at most two

[1] Consistency propagation is the process of iterative reduction of the domain and/or relations of a Constraint Satisfaction Problem, while not loosing solutions.

variables) has shifted the focus of the research to developing a wide variety of algorithms for *Arc Consistency* (AC), we have seen an increase of the number of new algorithms for GAC since 2007 with the advent of algorithms that filter the relations, called Simple Tabular Reduction (STR) [Ullmann, 2007].

In this thesis, we conduct an in-depth study of the latest algorithms for enforcing GAC and evaluate them empirically on both random and benchmark problems.

1.2 Contributions

In this thesis, we study various algorithms that are proposed in the literature to enforce Generalized Arc Consistency. We study a number of algorithms based on Simple Tabular Reduction (STR) [Ullmann, 2007] and one algorithm where constraints are represented as Multivalued Decision Diagrams [Cheng and Yap, 2010]. We implement these algorithms and compare their performances on various benchmark and randomly generated problems. Finally, we introduce a hybrid algorithm that combines the use of two of those algorithms. Our main contributions are as follows:

1. We identify the conditions and problem characteristics for which one algorithm outperforms all others. We believe that those conditions can help a human user or an automated selection procedure decide which algorithm to apply in which context.

2. We introduce a hybrid algorithm that combines the advantages of two simple STR-based algorithms, one that is most effective on positive table constraints and the other most effective on negative table constraints. As a result, we are able to handle individually the constraints in the problem as it is most fit.

3. We provide a simple, crisp, and didactic description of the mechanism and data structures of the implemented algorithms, which will make it easier for other researchers/students to quickly grasp how those algorithms operate and to understand their differences.

1.3 Outline of Thesis

This thesis is structured as follows. Chapter 2 reviews background information about CSPs. Chapter 3 discusses the various studied GAC algorithms, explaining and illustrating their mechanisms and the data structures they rely on. Chapter 4 describes an empirical evaluation of their performance on randomly generated CSPs as well as benchmark problems. Chapter 5 concludes this thesis with future direction for research. Finally, Appendix A provide the pseudocode of the various algorithms studied, Appendix B describes the data sets discussed in Chapter 4, and Appendix C summarizes the structure of the C code of our implementation.

Chapter 2

Background

In this chapter, we provide background information about Constraint Satisfaction problems (CSPs), and discuss various constraint representations. We review into domain and relational consistency properties, and focus on the concept of Generalized Arc Consistency (GAC). Finally, we summarize how a CSP instance is solved with backtrack search. And finally, we give related work.

2.1 Constraint Satisfaction Problem

A Constraint Satisfaction Problem (CSP) is defined a triplet (X, D, C), where:

- $X = \{x_1, x_2, \ldots, x_n\}$ is the set of variables;
- $D = \{dom(x_1), dom(x_2), \ldots, dom(x_n)\}$ is the set of variables domains, where $dom(x_i)$ is the nonempty set of domain values for the variable x_i; and
- $C = \{c_1, c_2, \ldots, c_m\}$ is the set of constraints that apply to the variables, restricting the allowed combinations of values for variables.

A solution to a CSP is an assignment of a value to each variable such that all the constraints are satisfied. A CSP instance is said to be *satisfiable* or *consistent* if a solution exists. Solving a CSP consists in determining the existence of a solution, and is NP-complete.

We consider that the variable domains are finite sets of values. A constraint c_i is specified by:

- a scope $scp(c_i)$, which is the set of variables to which the constraint applies, and
- a relation $rel(c_i)$, which is a subset of the Cartesian product of the domains of the variables in $scp(c_i)$. Each tuple $\tau_i \in rel(c_i)$ specifies a combination of values that is allowed (i.e., supports) or forbidden (i.e., conflicts or no-goods).

The *arity* of the constraint is the size of the scope. A constraint can be unary (arity 1), binary (arity 2), or non-binary (arity >2).

Example 1 The n-queens problem is to place n queens on an $n \times n$ chessboard such that no two queens attack each other. This problem can be modeled as a binary CSP as follows. The variables are the columns of the chessboard $\{x_1, \ldots, x_n\}$. The values of a variable are the positions in the corresponding row of the chessboard. Binary constraints exist between every pair of variables, and forbid the positions of the queens that are on the same row or the same diagonal. Figure 2.1 shows a solution for the 4-queens problem.

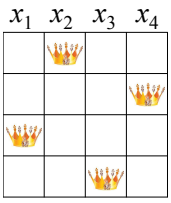


Figure 2.1: Board representation of the 4-queens problem and one of its two solutions

Example 2 Consider the cryptarithmetic puzzle TWO + TWO = FOUR, where each letter in the puzzle refers to a digit, and no two letters refer to the same digit. The question is to map each letter to a digit so that the arithmetic operation holds. This puzzle can be modeled as non-binary CSP. Figure 2.2 shows the puzzle, where we to introduce the letters x_1, x_2, and x_3 to account for the carry over. Here, the set

$$\begin{array}{cccc} x_3 & x_2 & x_1 & \\ & \mathbf{T} & \mathbf{W} & \mathbf{O} \\ + & \mathbf{T} & \mathbf{W} & \mathbf{O} \\ \hline \mathbf{F} & \mathbf{O} & \mathbf{U} & \mathbf{R} \end{array}$$

Figure 2.2: An example cryptarithmetic problem

of variables is $X = \{\text{T,W,O,F,U,R},x_1, x_2, x_3\}$; the domains of T, W, O, F, U and R are $\{0,\ldots,9\}$, and the domains of x_1, x_2, x_3 are $\{0,1\}$. The constraints are defined arithmetically as follows:

$$\begin{aligned} c_1&: \quad \text{R} + 10\ x_1 = \text{O} + \text{O} \\ c_2&: \quad \text{U} + 10\ x_2 = x_1 + \text{W} + \text{W} \\ c_3&: \quad \text{O} + 10\ x_3 = x_2 + \text{T} + \text{T} \\ c_4&: \quad \text{F} = x_3 \\ c_5&: \quad \text{T} \neq \text{W} \neq \text{O} \neq \text{F} \neq \text{U} \neq \text{R} \end{aligned}$$

A CSP is represented as a graph, called the constraint graph or constraint network. This representation makes it possible to apply graph algorithms to the representation of the CSP.

Figures 2.1 and Figure 2.4 show the graphical representations of the 4-queens and the cryptarithmetic problems respectively. The constraint graph contains a node for every variable and there are edges linking variables if they share constraints.

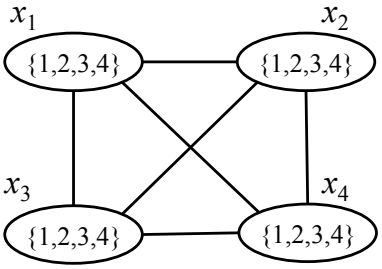


Figure 2.3: Graphical representation of the 4-queens problem

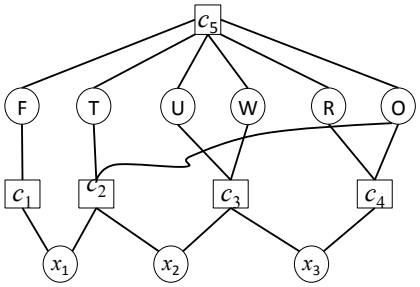


Figure 2.4: Graphical representation of the cryptarithmetic problem

2.2 Representing Constraint Relations

As stated above, a constraint is defined by its scope, which is the set of variables to which it applies, and its relation, which determines the allowed combination of values to the variables in the scope of the constraint. Given that the relation is a set, it can be expressed either in *extension* or in *intension*. In this thesis, we focus on constraints expressed in extension. The tuples in a relation represent either allowed combinations of values (a.k.a., goods or supports), or forbidden combinations (a.k.a., nogoods or conflicts).

We distinguish three different encoding of a relation expressed in extension: linked list, table, and multi-valued decision diagram. Below, we discuss the latter of those representations.

2.2.1 Table constraints

A table constraint is defined by explicitly listing the tuples that are either allowed or disallowed for the variables in its scope. If the tuples listed are allowed then the table is said to be a *positive* table; otherwise it is a *negative* table. Table constraints are sometimes referred to as *ad-hoc* constraints. Positive constraints typically arise in practice in configuration problems and databases.

The set of tuples of a constraint c is denoted $table(c)$. We encode this set as an array of tuples indexed from 0 to $|table(c)| - 1$. The worst-case space complexity for storing a table is $\mathcal{O}(tr)$, where t is the number of tuples in the table and r is the constraint arity. The status column in this array indicates whether the tuple is alive (0) or deleted (1).

Most algorithms studied in this thesis use the data structure $subtable(c, (x, a))$, which gives quick access to all the tuples of a constraint c with a given *variable-value pair* (x, a). Those structures are generated for each constraint and each variable-value pair of each variable in the scope of the constraint. Typically, a sub-table is implemented as an array with indices ranging from 0 to $|table(c, x, a)| - 1$. Formally, we have $subtable(c, (x, a)) = \sigma_{x \leftarrow a}(c) = \{t | (t \in c) \wedge (\pi_{\{x\}}(t) = (a))\}$, where σ is the relational selection operator. Figure 2.5 shows a table constraint and two of its three sub-tables. Those sub-tables are accessed via hash-maps, where the key is a constraint, a variable, and its value. The worst-case space complexity of the hash-maps remains $\mathcal{O}(tr)$.

2.2.2 Multi-Valued Decision Diagrams (MDD)

The tuples in a constraint can also be represented by a multi-valued decision diagram (MDD), which is an arc-labeled directed acyclic diagram. In the special case where

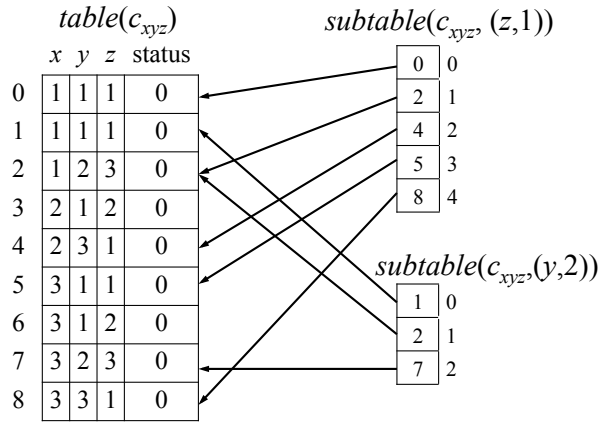


Figure 2.5: A table constraint and two of its three sub-tables

all the domains have only two values, the MDD becomes a Binary Decision Diagram (BDD). An MDD has at least one root node (source). It also has exactly two terminal nodes (sinks) that can be either tt (allowed tuples) or ff (forbidden tuples). For simplicity, when the MDD represents allowed tuples, ff node is usually omitted from the diagram. Figure 2.6 depicts the MDD representation of the table constraint in Figure 2.5.

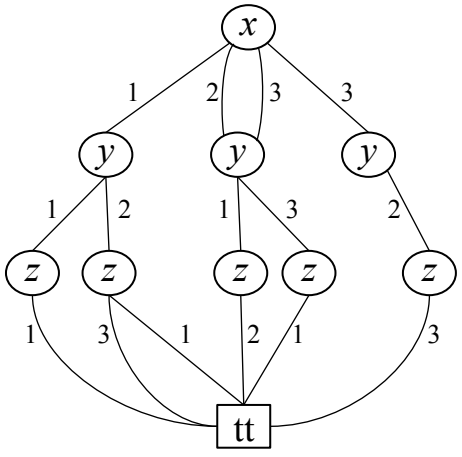


Figure 2.6: MDD representation of a constraint

The MDD of a constraint c on variables x, y, z is denoted $mdd(c_{xyz})$. The main advantage of MDDs is that they require less space than table constraints and provide a

quicker access to a tuple. In [2010], Cheng and Yap introduced the algorithm 'mddify,' which takes as input a table constraint (as supports or conflicts) and outputs a reduced size MDD. We use that algorithm to create our MDD constraints. Its description is beyond the scope of this thesis.

2.3 Consistency

CSPs are NP-complete and typically solved using backtrack search (BT) algorithms. In order to reduce the size of the search space, we enforce local consistency property. We distinguish two types of consistency properties: those formulated on the domains of the variables and those formulated on the relations. The algorithms for enforcing a given consistency property typically remove inconsistent values from the domains of the variables or inconsistent tuples from the relations of the constraints.

2.3.1 Domain consistency properties

The most common domain consistency property is arc consistency.

Definition 1 *A* ***constraint*** c *on the variables* x, y *with the domains* $dom(x)$ *and* $dom(y)$ $(rel(c) \subseteq dom(x) \times dom(y))$ *is* ***arc consistent*** *if*

- $\forall a \in dom(x) \exists b \in dom(y)$ such that $(a, b) \in rel(c)$
- $\forall b \in dom(y) \exists a \in dom(x)$ such that $(a, b) \in rel(c)$

A CSP is arc consistent *if all its binary constraints are arc consistent.*

Many algorithms to enforce arc consistency exist and include AC-1, AC-3, AC-4, AC-5, etc. The above definition of arc consistency refers only to binary constraints.

The corresponding property for non-binary constraints is known as Generalized Arc Consistency (GAC).

Definition 2 ***(Generalized arc consistency)*** *Given a CSP (X, D, C), a constraint $c \in C$, and a variable $x \in X$,*

- *A value $a \in dom(x)$ is GAC for $c \in C$ where $x \in scp(c)$ iff there exists a valid tuple τ satisfying c such that $\pi_x(\tau) = (a)$. τ is called a* support *for (x, a) in c.*
- *A variable $x \in scp(c)$ is GAC for c iff all the values in $dom(x)$ are GAC with c.*
- *A constraint c is GAC iff all the variables in its scope are GAC for c.*
- *The CSP is GAC iff all the constraints in C are GAC.*

Enforcing GAC on a CSP is accomplished by removing GAC-inconsistent values from the domains of the variables. If no domain is empty in the result CSP, the CSP is said to be GAC.

In this thesis, we study various algorithms for enforcing GAC and compare their performance on CSP benchmarks and randomly generated instances.

Other domain consistency properties include node consistency, k-consistency (where 3-consistency is called path consistency), and singleton arc consistency (SAC).

2.3.2 Relational consistency properties

Relational consistency properties are properties formulated on the relations of the constraints.

In [1997], Dechter and van Beek introduced relational consistency properties that may require adding new constraints to the CSP. Such properties have not yet been exploited in practice because they typically modify the constraint network, which is highly undesirable.

In [Gyssens, 1986], Gyssens introduced m-wise consistency. A CSP is m-wise consistent iff for every combination of m relations in the CSP, every tuple in every relation can be extended in a consistent manner to the $m-1$ relations in the CSP. Karakashian *et al.* [2010] give the first practical algorithm for enforcing m-wise consistency, denote R($*$,m)C.

In this thesis, we are interested in pairwise consistency, where $m = 2$.

Definition 3 *[Gyssens, 1986]* ***Pairwise Consistency (PWC)****. A tuple τ_i in the table of a constraint c_i is PWC iff $\forall c_j \in C$, $\exists \tau_j \in table(c_j)$ such that $\pi_{scp(c_i) \cap scp(c_j)}(\tau_i) = \pi_{scp(c_i) \cap scp(c_j)}(\tau_j)$. We say that τ_i and τ_j are PWC and a PW-support of one another. A CSP is PWC iff every tuple of every constraint has a PW-support.*

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

2.4 Algorithms for Generalized Arc Consistency

GAC is formulated as a consistency property of the domains of the variables. Until recently, all algorithms for enforcing GAC filtered only the domains of the variables. Such is the case of GAC3 [Mackworth, 1977] and GAC2001 [Bessière *et al.*, 2005].

More recently, Ullmann [2007] introduced an algorithm for enforcing MAC that not only removes inconsistent values from the domains of the variables but also updates the relations accordingly removing a tuple whenever any of their value in the tuple is removed from the domain of the corresponding variable. This algorithm is known as Simple Tabular Reduction (STR) because it operates on table constraints. STR blurs the boundary that separates domain consistency properties and relation

consistency properties because it ensures that the relations themselves have no tuples that are inconsistent.

Several recent algorithms were proposed in the literature to improve the performance of the STR algorithm. One such algorithm, extended STR (eSTR), goes beyond enforcing GAC and enforces pairwise consistency. Thus, eSTR-like algorithms enforce both domain and relation consistency properties.

In this thesis, we study these algorithms in detail.

2.5 Solving CSPs

CSPs can be solved either with backtrack search (BT) or local search. Backtrack search exhaustively and systematically explores combinations of values for variables, constructively building consistent solutions. The space requirement of BT is linear in the number of variables because BT explores the search space in a depth-first manner. It provides the only sound and complete procedure for finding a solution to a CSP.

There are many factors that can affect the performance of search. One of such factors is the way the variables/values are ordered for instantiation. Many heuristics for variable and for value ordering have been introduced. The common wisdom is to instantiate the most constrained variable first. To improve the cost of search, it is always beneficial to enforce a consistency property at the *pre-processing* stage (i.e., before search starts) and maintaining it after the instantiation of each variable, throughout search (i.e., *look-ahead*).

2.6 Phase Transition

Cheeseman *et al.* [1991] presented empirical evidence, for some random combinatorial problems, of the existence of a phase transition phenomenon at a critical value (cross-over point) of an order parameter. They showed a significant increase in the cost of solving these problems around the critical value. Figure 2.7 illustrates this situation.

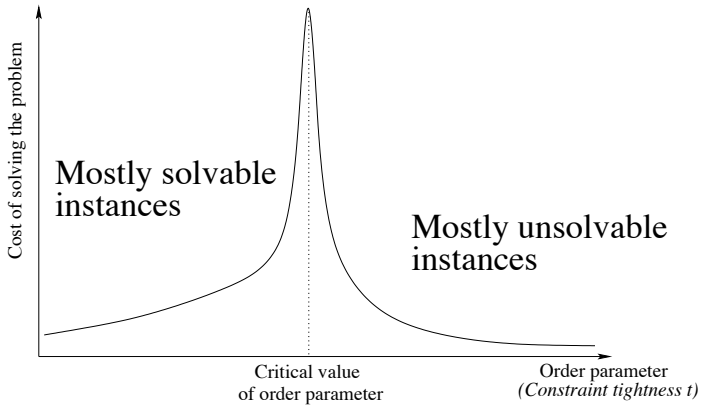


Figure 2.7: Cost of problem solving

They also showed that the location of the phase transition and its steepness change with the size of the problem. Because problems at the cross-over point are acknowledged to be probabilistically the most difficult to solve, empirical studies to compare the performance of algorithms are typically conducted in this area. In the case of CSPs, constraint tightness (with fixed values for the number of variables, domain size, and constraint density or ratio) and constraint ratio (with fixed values for number of variables, domain size, and constraint tightness) are often used as order parameters.

Summary

In this chapter, we reviewed some background information on CSPs that is relevant to this thesis. We described how the relations of the constraints can be represented. We reviewed common local consistency properties, and discussed Generalized Arc Consistency, which is the focus of this thesis.

Chapter 3

GAC Algorithms

The most commonly used algorithm for enforcing GAC is GAC2001 [Bessière *et al.*, 2005]. Recently, new algorithms have been proposed to enforce GAC, that exploit the representation of the relations. They all enforce GAC and filter the domains. These algorithms are mainly the following:

- *Positive table constraints*: The following algorithms filter the relations of the constraints: STR1 [Ullmann, 2007], STR2 [Lecoutre, 2011], STR3 [Lecoutre *et al.*, 2012]. The following algorithm enforces pairwise consistency as well: eSTR* (which includes eSTR1 and eSTR2) [Lecoutre *et al.*, 2013].

- *Negative table constraints*: The following algorithm operates on relations formatted as tables of nogoods (i.e., forbidden tuples): STR-Ni [Li *et al.*, 2013]. It removes from the table the nogoods that no longer need to be checked.

- *Multi-valued decision diagrams (MDD) constraints*: The following algorithm keeps track of nodes from the MDD that lead to inconsistent tuples: mddc [Cheng and Yap, 2010].

Table 3.1 summarizes the above-listed algorithms, their space and time complexity, as well as the dedicated data structures on which they rely.

Table 3.1: Overview of the GAC algorithms studied

Algorithm	**Complexity**		**Data Structures**
	Space	**Time**	
GAC2001 [Bessière *et al.*, 2005]	$\mathcal{O}(erd)$	$\mathcal{O}(er^2d^r)$	$LastGAC$
STR1 ≡ GACstr [Ullmann, 2007]	$\mathcal{O}(n+rt)$	$\mathcal{O}(r'd+rt')$	$gacValues$, $table(c)$
STR2 ≡ GACstr2 [Lecoutre, 2011]	$\mathcal{O}(n+rt)$	$\mathcal{O}(r'(d+t'))$	$gacValues$, $table(c)$, S^{val}, S^{sup}
STR3 [Lecoutre *et al.*, 2012]	$\mathcal{O}(rd+t)$	$\mathcal{O}(rt+m)$	$row(c,x)$, $invalid(c)$, $dep(c)$
eSTR [Lecoutre *et al.*, 2013]	$\mathcal{O}(n+max(r,g)t)$	$\mathcal{O}(rd+max(r,g)t)$	$ctr[c][c_i]$, $ctrIndices[c][c_i]$, $ctrLink[c][c_i]$
STR-Ni [Li *et al.*, 2013]	$\mathcal{O}(n+rt')$	$\mathcal{O}(r'd+rt')$	$table(c)$, $count(x,a,c)$
mddc [Cheng and Yap, 2010]	$\mathcal{O}((hd+k+1)m)$	$\mathcal{O}(e_{mdd(c)}+\lambda)$	$MDD(c)$, Σ^{NO}, Σ^{YES}, $gacValues(x)$

n the number of variables in the CSP
d the maximum domain size of the variables
e is the number of constraints
r the constraint arity
t the maximum size of a relation
m the length of a path in the search tree
g the number of intersecting constraints
$e_{mdd(c)}$ the number of MDD edges
λ number of GAC-inconsistent values
h number of MDDs
k number of MDD constraints

In this chapter, we study the mechanisms of the above algorithms and the data structures that they exploit. We state their complexity in terms of e is the number of constraints, r the constraint arity, t the maximum size of a relation, n the number of variables in the CSP, d the maximum domain size of the variables, and m the length of a path in the search tree. Further, we propose a hybrid algorithm, STR-h. Our algorithm combines STR1 and STR-Ni because they are compatible with respect to the data structures they use.

3.1 GAC2001

GAC2001 (a.k.a. GAC-3.1) adapts, to non-binary CSPs, the algorithm for binary CSPs AC2001/3.1 [Bessière *et al.*, 2005] is based on AC2001/3.1 (for binary constraints). Remember that enforcing the property GAC on a CSP requires that we check that each value a in the domain of a variable x has a supporting tuple (that is

alive or active) in each of the constraint c that apply to the variable (i.e., $x \in scp(c)$)). GAC2001 uses the data structure $LastGAC((x, a), c)$ that keeps a pointer to the tuple $\tau \in rel(c)$ such that τ was the latest found support of (x, a) in $rel(c)$. Because the table of the constraint c is traversed linearly, from top to bottom, we say that τ is the 'smallest' support of c.

In order to check the GAC consistency of a variable x given a constraint c, GAC2001 calls the function REVISE2001 This function checks for each value $a \in dom(x)$ whether $LastGAC((x, a), c)$ still belongs to $rel(c)$ (i.e., is still active). If it does not, REVISE2001 continues to traverse the table $rel(c)$ seeking another supporting tuple for (x, a). If it finds a supporting tuple $\tau \in rel(c)$, we set $LastGAC((x, a), c) \leftarrow \tau$. Otherwise, we remove a from $dom(x)$ (i.e., $dom(x) \leftarrow dom(x) \setminus \{a\}$). The function $succ(\tau, rel(c))$ returns the smallest tuple in $rel(c)$ greater than τ. The pseudo-code of the REVISE2001 is provided as Algorithm 1 in Appendix A.

We illustrate the operation of REVISE2001 on the simple example of Figure 3.1. Suppose that value 1 was removed for some reason from the domain of the variable

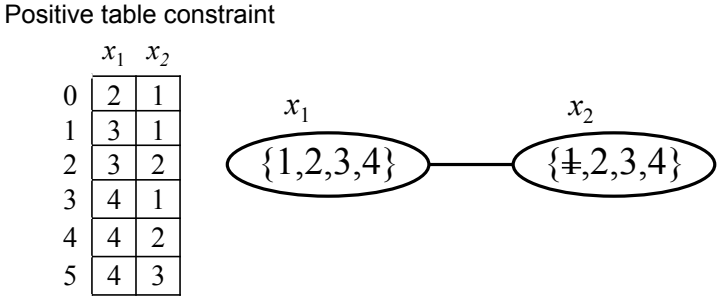


Figure 3.1: An example using REVISE2001

x_2. REVISE2001 on $(x_1, 4)$ finds that $LastGAC((x_1, 4), c_{12}) = 3$ (where c_{12} is the constraint between x_1 and x_2). The tuple at index 3 can no longer be a GAC support for $(x_1, 4)$ because the value $1 \notin dom(x_2)$. So, REVISE2001 moves to the next tuple in the table, the one at index 4, verifies that it is indeed a supporting tuple for $(x_1, 4)$,

and resets $LastGAC((x_1, 4), c_{12}) = 4$.

The worst-case time complexity of GAC2001 is $\mathcal{O}(er^2d^r)$ and its space complexity is $\mathcal{O}(erd)$. (Reminder, e is the number of constraints, r is the constraint arity, and d is the maximum domain size of the variables.)

3.2 Simple Tabular Reduction for Positive Table Constraints

Ullmann [2007] proposed the first GAC algorithm that updates the relations to remove inconsistent tuples. This algorithm is called "Simple Tabular Reduction" (STR). In this thesis, we refer to this algorithm as STR1 and GACstr in order to differentiate it from its later variations.

Whenever a value is removed from the domain of a variable, all tuples mentioning the deleted domain values are removed from all the constraint tables in which the variable appears. In turn, whenever a domain value for a given variable is missing from the active tuples of any constraint that applies to the variable is removed from the domain of the variable.

Below, we discuss STR1 and the subsequent algorithms that improve it: STR2, STR3, eSTR*, STR-Ni. An important feature/advantage of STR-like algorithms is the cheap restoration of the data structures used when backtracking occurs.

We denote by $future(c)$ all the variables in the scope of c that have not yet been instantiated.

3.2.1 STR1

The formulation of STR depends on the representation of each constraint as an object. The constraint object has to maintain the following accessors/fields:

- $table(c)$ contains the initial set of tuples allowed by the constraint c.
- $table(c)[i]$ is a tuple in the ith position of $table(c)$, where $0 \leq i \leq t-1$ and t is the number of tuples in the table.
- $first(c)$ is the position of the first non-deleted tuple in $table(c)$. When the table of c is empty, $first(c) = -1$.
- *removedHead* is an array of size n such that $removedHead(c)[d]$ is the position of the first deleted tuple of $table(c)$ that was removed from the search at depth d. It is -1 if nothing was removed at depth d.
- *removedTail* is an array of size n such that $removedTail(c)[d]$ is the position of the last deleted tuple of $table(c)$ that was removed from the search at depth d. It is initialized only if $removedHead(c) \neq -1$.
- *next* is a pointer from a given active tuple in $table(c)$ to the next active tuple in the table. *next* is implemented as an array of size t. For the tuple in position i in $table(c)$, $next(c)[i]$ gives the position in $table(c)$ of the next active tuple in the table. When no such active tuple exists, then $next(c)[i] \leftarrow -1$.

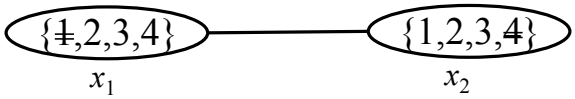
As stated in Section 2.2.1, our implementation of $table(c)$ has a column called *status* where $table(c)[i][status] = 0$ when the tuple at position i is still active and $table(c)[i][status] = 1$ otherwise. As a result, we do not need to implement *removedHead* and *removedTail*

When processing a given constraint c, we store for each the variable x in the scope of c the domain values that are GAC for x given c in the data structure $gacValues(x)$.

We implemented the function GACstr (Algorithm 2 in Appendix A), which is an improvement by Lecoutre [2011], of the original algorithm of Ullmann [2007] . The loops at lines 1, 7 and 15 only operate on future variables because we can only remove values from the domains of uninstantiated variables. The sets $gacValues(x)$, $x \in scp(c)$ are emptied at lines 1 and 2 because no value for x is initially guaranteed to be GAC. Then, in lines 4–14, we loop through all tuples of $table(c)$ to test them for validity (i.e., we check if the tuples are allowed or forbidden). When a tuple τ is found to be invalid, it is marked as deleted with respect to the current depth of search. Otherwise, the domain values appearing in τ are added to the $gacValues$ of their corresponding variables. After going through all tuples in $table(c)$, we remove unsupported values ($dom(x) \setminus gacValues(x)$) at lines 15–20. Whenever a domain becomes empty, inconsistency is returned at line 18.

When backtracking occurs during search, all values and tuples that were removed at that depth, or deeper, in search are restored by simply setting $table(c)[i][status]$ back to 0.

We illustrate the operation of STR1 on a simple example. Suppose we first run STR1 on the constraint c_{12} over x_1 and x_2 shown in Figure 3.2. While we looping



$table(c_{12})$

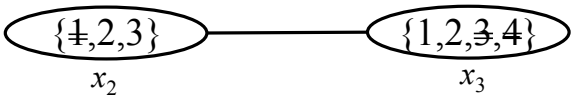
	x_1	x_2	status
0	2	1	0
1	3	1	0
2	3	2	0
3	4	1	0
4	4	2	0
5	4	3	0

Figure 3.2: STR1: Enforcing GAC on $table(c_{12}$

through all the tuples in $table(c_{12})$, we update $gacValues$ as follows:

- $gacValues(x_1) \leftarrow \{2,3,4\}$
- $gacValues(x_2) \leftarrow \{1,2,3\}$

Next, we update $dom(x_1), dom(x_2)$ to reflect the filtering by STR1. After that, we run STR1 on the constraint c_{23} defined over x_2 and x_3 as shown in Figure 3.3.



$table(c_{23})$

	x_2	x_3	status
0	2	1	0
1	3	1	0
2	3	2	0
3	4	1	1
4	4	2	1
5	4	3	1

Figure 3.3: STR1: Enforcing GAC on $table(c_{23}$

While looping through the tuples of $table(c_{23})$, we delete the tuples at indices 3, 4, and 5 because the value 4 no longer appears in $dom(x_2)$. Further, we update $gacValues$ as follows:

- $gacValues(x_2) \leftarrow \{2,3\}$
- $gacValues(x_3) \leftarrow \{1,2\}$

At this point, the domains of x_2, x_3 are updated accordingly.

The worst-case time complexity of STR1 (Algorithm 2 in Appendix A) is $\mathcal{O}(r'd + rt')$ where, for a given constraint c, $r' = |future(c)|$ denotes the number of uninstantiated variables in c and t' is the size of the current table of c. The worst-case space complexity of STR1 is $\mathcal{O}(n + rt)$ per constraint.

3.2.2 STR2

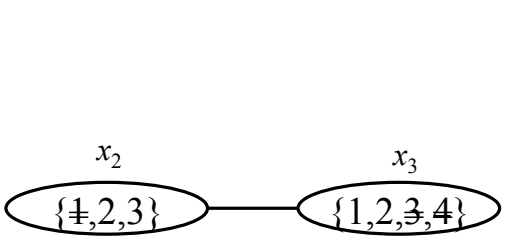
Lecoutre [2011] introduced an improved version of GACstr known as GACstr2 or simply STR2. STR2 offers two improvements.

According to the first improvement, while looping through the active tuples of a constraint and as soon all the values in the domain of a variable have been proven to be GAC, we do not need to continue to seek supports for the values of this particular variable. To implement this idea, a new data structure S^{sup} is introduced. S^{sup} contains variables in $future(c)$ (uninstantiated variables) whose domain contains at least one value for which no support has been found. In the STR2 algorithm (Algorithm 3 in Appendix A), lines 1, 6 and 8 initialize S^{sup} to be the same as $future(c)$, while line 20 removes any variable x for which $|gacValues(x)| \leftarrow |dom(x)|$ (i.e., all values of $dom(x)$ are supported) from S^{sup}. As a result, we only iterate over variables in S^{sup} at lines 16 and 26.

The second improvement aims to avoid unnecessary validity checks. At the end of an invocation of STR2 on a constraint c, we know that for every variable $x \in scp(c)$, every tuple τ where $\pi_\tau(x) \notin dom(x)$, τ has been removed from $table(c)$. If there is no backtrack and $dom(x)$ has not changed between this invocation and the next invocation, then, at the time of the next invocation, it is certain that $\pi_\tau(x) \in dom(x)$ for every alive tuple τ in $table(c)$. Therefore, there is no need to check whether $\pi_\tau(x) \in dom(x)$. To implement this improvement, we introduce S^{val}, which is the set of all the uninstantiated variables of c whose domains have been reduced since the previous invocation of STR2. When STR2 is called on a constraint c, the last assigned variable, denoted $lastAssignedVariable$, is added to S^{val} if it is in the scope of c. After any variable x has been instantiated (e.g., $x \leftarrow a$), some tuples may become invalid due to the removal of the other values from $dom(x)$. The last assigned variable

is the only instantiated variables for which validity operations must be performed. In STR2 (Algorithm 3 in Appendix A), S^{val} is first initialized at lines 2 through 5. At line 9, $getLastRemovedValue(dom(x))$ is the last value that was removed from the domain of x, and $LastRemoved(c, x)$ is the last value that was removed from the domain of x while processing constraint c. If these two values are different, then it means that the domain of x has changed since the previous invocation of STR2 on the constraint c, and then x is added to S^{val} at line 10.

We illustrate the operation of STR2 on the simple example of Figure 3.4. Suppose



$S^{sup} = \{x_2, x_3\}$

$S^{val} = \{x_2, x_3\}$

table(c_{23})

	x_2	x_3	*status*
0	2	1	0
1	3	1	0
2	3	2	0
3	4	1	1
4	4	2	1
5	4	3	1

Figure 3.4: An example using STR2

we have removed value 4 from $dom(x_2)$ for some reason, and then call STR2 on the constraint c_{23} over x_2 and x_3. Assume that no variable has been yet instantiated (e.g., we are still at the preprocessing stage), the sets S^{sup} and S^{val} are shown in Figure 3.4, and the *gacValues*, *lastRemoved*, and *lastRemovedvalue* are as follows:

- $lastRemoved(c_{23}, x_2) \leftarrow NIL$
- $lastRemovedvalue(x_2) \leftarrow 4$
- $gacValues(x_2) \leftarrow \{2,3\}$
- $lastRemoved(c_{23}, x_3) \leftarrow NIL$

- $lastRemovedvalue(x_3) \leftarrow NIL$
- $gacValues(x_3) \leftarrow \{1,2\}$

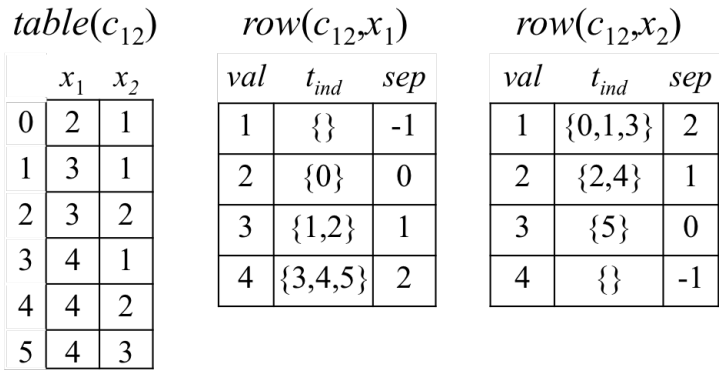
Because $lastRemoved(c_{23}, x_2) \neq lastRemovedvalue(x_2)$, and $lastRemoved(c_{23}, x_3) \neq lastRemovedvalue(x_3)$, both variables x_2 and x_3 are added to S^{val}. After validity operations have been performed on the variables in S^{val}, the domains of x_2 and x_3 are updated to {2,3} and {1.2}, respectively.

The worst-case time complexity of STR2 (Algorithm 3) is $\mathcal{O}(r'(d + t'))$, where $r' \leq r$. Similar to STR1, the worst-case space complexity of STR2 is $\mathcal{O}(n + rt)$ per constraint, and $lastRemoved$ takes $\mathcal{O}(r)$ space. S^{sup} and S^{val} also take $\mathcal{O}(r)$ space, but may be shared by all constraints. (Reminder, r is the constraint arity, and d is the maximum domain size of the variables, t the maximum size of a relation, and n the number of variables in the CSP).

3.2.3 STR3

Lecoutre *et al.* [2012] introduced STR3, which has a complex representation of the table constraints. In STR3, the representation focuses on domain values, associating a set of tuple identifiers with each value in a variable, indicating the tuples where the value appears. Figure 3.5 illustrates an example of the standard table representation, $table(c_{12})$, and the equivalent representation used by STR3 for the variables x_1 and x_2 in $row(c_{12}, x_1)$ and $row(c_{12}, x_2)$, respectively. The required data structures are as follows:

- $row(c, x)$ is a table with three columns of length $|dom(x)|$. It stores, for each value in the domain of x (column val), the list of tuple indices of c where the value appears (column t_{ind}), and an integer (column sep) initialized to the



	x_1	x_2
0	2	1
1	3	1
2	3	2
3	4	1
4	4	2
5	4	3

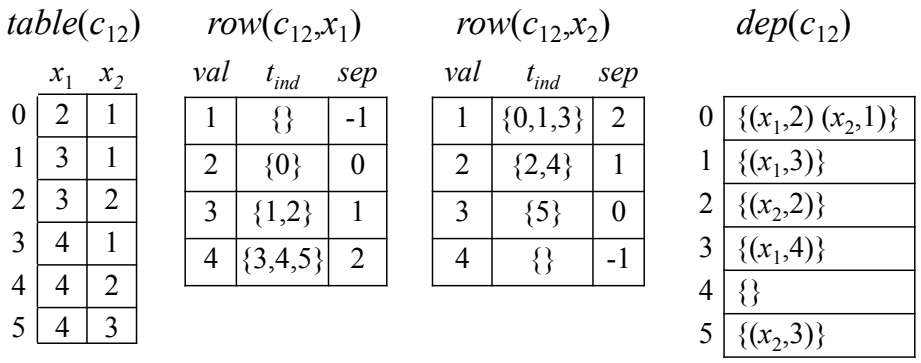
val	t_{ind}	*sep*
1	{}	-1
2	{0}	0
3	{1,2}	1
4	{3,4,5}	2

val	t_{ind}	*sep*
1	{0,1,3}	2
2	{2,4}	1
3	{5}	0
4	{}	-1

Figure 3.5: Equivalent table representation

size of the corresponding $t_{ind} - 1$. This integer is called the *separator* of the corresponding t_{ind}, and indicates the position of the tuple in t_{ind} that is the last known support the corresponding value of x in c. As STR3 progresses, and tuples are deleted, the value of *sep* is decremented. An index that appears after *sep* corresponds to a deleted tuple.The tuples whose index appears

- *dep*(c) is called the *dependency list* of constraint c. This structure is used to determine whether a tuple of c is an active support for some variable-value pair. It is implemented as an array of size t of sets. *dep*(c)[i] is the set of variable-



	x_1	x_2
0	2	1
1	3	1
2	3	2
3	4	1
4	4	2
5	4	3

val	t_{ind}	*sep*
1	{}	-1
2	{0}	0
3	{1,2}	1
4	{3,4,5}	2

val	t_{ind}	*sep*
1	{0,1,3}	2
2	{2,4}	1
3	{5}	0
4	{}	-1

0	{(x_1,2) (x_2,1)}
1	{(x_1,3)}
2	{(x_2,2)}
3	{(x_1,4)}
4	{}
5	{(x_2,3)}

Figure 3.6: Dependency list of the table representation

value pairs (x, a), such that the i^{th} tuple in c is an active support of (x, a) on

c. $dep(c)[i]$ is initialized by adding the variable-value pair (x, a) to $dep(c)[i]$, When the tuple at index i is deleted, all the variable-value pairs in $dep(c)[i]$ must find a new active support. *dep* is updated during propagation and search as tuples are deleted. However, it needs not be updated upon backtracking, because backtracking during search does not invalidate supports. Figure 3.6 illustrates this data structure for the constraint c_{12}.

- $invalid(c)$ is a *sparse set* containing all the invalid tuples for constraint c. Sparse sets were introduced by Briggs and Torczon [1993] to represent a set in a compact manner. $invalid(c)$ contains at most t elements. It is represented by two arrays (*dense* and *sparse*) and a counter (*members*). *dense* contains all the elements in the set and *sparse* contains pointers to the position of the elements in *dense*. *members* store the number of elements in the set. Initially, $members \leftarrow 0$ and the two arrays are initialized to be empty. Figure 3.7 illustrates an example of a sparse set that contains $\{0, 2, 4, 7\}$. To add an element $0 \leq k < n$ to a sparse set that does not already contain k, we set $dense[members] \leftarrow k$, $sparse[k] \leftarrow members$, and increment *members* by 1. An element k is in the set iff $sparse[k] < members$ and $dense[sparse[k]] = k$. To remove an element, we replace it with the last element in *dense* and decrement *members*. We denote

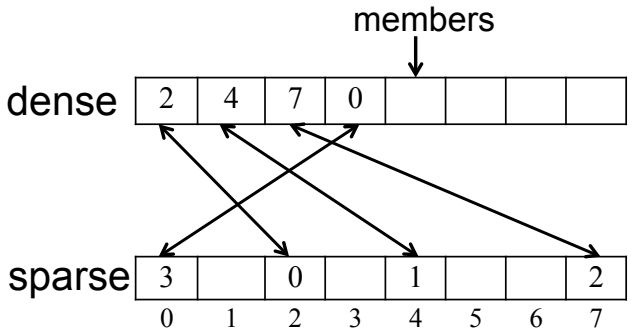


Figure 3.7: An example of the sparse set representation

$dense(invalid(c))$, $sparse(invalid(c))$ and $members(invalid(c))$ the accessors to the internal data-structure of the sparse set.

Unlike other STR algorithms, tuples are not explicitly discarded from the tables of STR3, rather the value *sep* partitions the tuples into forbidden and allowed/unvisited tuples.

Central to the implementation of STR3 (Algorithm 5 in Appendix A) is the relationship between the separator (*sep*) and the dependency list (*dep*). A variable-value pair (x, a) is GAC on c, either because

- The position of *sep* in the row where $val = a$ points to a tuple that is not stored in $invalid(c)$, thus, is an active support, or
- (x, a) appears at some index of $dep(c)$ that corresponds to a tuple that is not in $invalid(c)$.

Only one of the above two conditions is required for (x, a) to be GAC on c. When both conditions hold, the tuple of c they point to may be the same. When this situation occurs, we say that the separator and the dependency list are synchronized for (x, a).

STR3 operates differently at preprocessing and search, as we explain below and illustrate in Figure 3.8.

Preprocessing: Before search starts (at preprocessing), GACinit (Algorithm 4 in Appendix A) is called to remove all invalid tuples and initializes *sep* and *dep* so that they point to opposite ends of $row(c, x)$ for each value a in the domain of x. Both values of *sep* and *dep* are active supports of (x, a) because all invalid tuples have already been removed during preprocessing. When *dep* and *sep* are

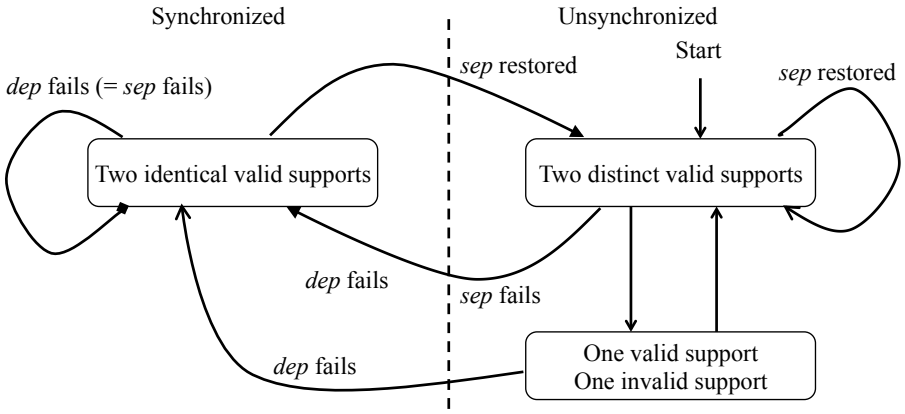


Figure 3.8: Relationship between separators and the dependency lists, [Lecoutre *et al.*, 2012]

synchronized at a particular value, we are provided with a single active support, and the dependency list simply mirrors what happens to the separator.

Search: During search, when a value v is assigned to a variable x, the values $dom(x) \setminus \{v\}$ are removed from $dom(x)$. Thus, STR3 is called to re-validate the lost support for all the constraints where $x \in scp(c)$. For each removed value a from x, every tuple in $row(c, x)$ where (x, a) appears (in column t_{ind}) becomes inactive, and STR3 appends that tuple to $invalid(c)$, if it is not already present. Whenever the tuple at index k becomes inactive (and the tuple is added to $invalid(c)$), all the variable-value pairs in $dep(c)[k]$ lose their active support. For each variable-value pair $(y, b) \in dep(c)[k]$ that has lost its active support, we look through the corresponding t_{ind} of the value b in $row(c, y)$ to check whether or not all its supporting tuples are in $invalid(c)$. If an active support is found, then the position *sep* is set to the position of the supporting tuple found in the corresponding t_{ind} and its previous value is stored for potential backtracking. If no active support is found, the value b is removed from the domain the variable y, and (y, b) is then added to the propagation queue. If a

domain wipe out occurs, STR3 returns inconsistency.

When STR3 succeeds during search, the separators and the dependency list remain synchronized. However, when search backtracks, only the *sep* associated with a value a in $row(c, x)$ is reverted, and $dep(c)[k]$ remains unchanged. Thus, the dependency list and separators may no longer be synchronized at (x, a). Whenever such unsynchronization occurs, the tuple at the position of *sep* for the value a in $row(c, x)$ and the tuple at index k are two distinct active supports of (x, a) in c. When there is loss of synchronization, and as long at least one of the two tuples (the one in position *sep* for the value a in $row(c, x)$, or the tuple at index k) remains active, we do not need to seek a new active support. In such a situation, we restore synchronization using the active support.

Figure 3.9 illustrates an example of the updates of the data structures when STR3 is called on (c_{12}, x_1) (for deleted values $a = 2$, $a = 3$, and $a = 4$) during search, where:

(a) Results after GACinit and the invocation of STR3 on the first removed value 2.

(b) Results after the invocation of STR3 on the second removed value 3.

(c) Results after the invocation of STR3 on the third removed value 4.

The worst-case accumulated cost of STR3 along a path in the search tree of length m is $\mathcal{O}(rt + m)$. The worst-case space complexity of STR3 is $\mathcal{O}(rd + t)$.

3.2.4 eSTR

Lecoutre *et al.* [2013] introduced the full pairwise consistency and STR algorithm (FPWC-STR or eSTR$*$ algorithm). That algorithm not only implements simple tabular reduction to enforce GAC, but also enforces pairwise consistency. The *

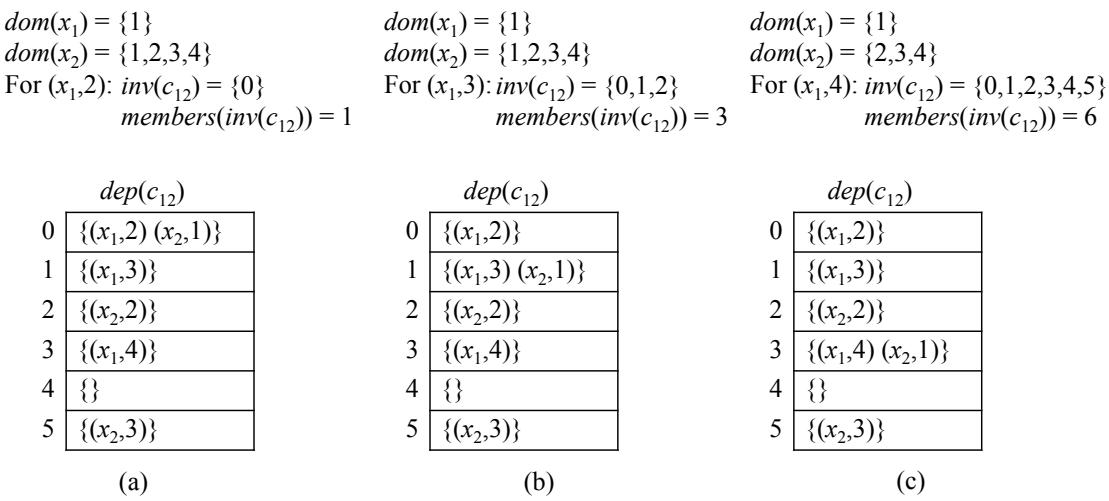


Figure 3.9: An example using STR3

indicates that the algorithm can be used in combination of any of the other STR algorithms (i.e., STR1 and STR2).

We denote I the set of variables at the intersection of the two constraints. We call a sub-tuple the projection on I of a tuple of any of the two constraints. The main idea of eSTR* is to store the number of times that each sub-tuple appears in the intersection of any two constraints. To this end, eSTR$*$ uses additional data structures that we explain with the example shown in Figure 3.10 for the two constraints c_{12} and c_{23}, and $I = \{x_2\}$:

- $ctrindices[c_i][c_j][i]$ is an array of size $|table(c_i)|$ that stores, for a tuple in $table(c_i)[i]$, the index in $table(c_i)$ of the first appearance of the value of the sub-tuple $\pi_I(table(c_i)[i])$. For example, for $ctrindices[c_{12}][c_{23}]$ and $I = \{x_2\}$, the values of the sub-tuple (x_2) are (1), (2), and (3). The first value, (1), appears for the first time in $table(c_{12})$ at index 0. Because this first value, (1), appears also in $table(c_{12})[1]$ and $table(c_{12})[3]$, we have $ctrindices[c_{12}][c_{23}][0] = ctrindices[c_{12}][c_{23}][1] = ctrindices[c_{12}][c_{23}][3] = 0$. Similarly, the second value,

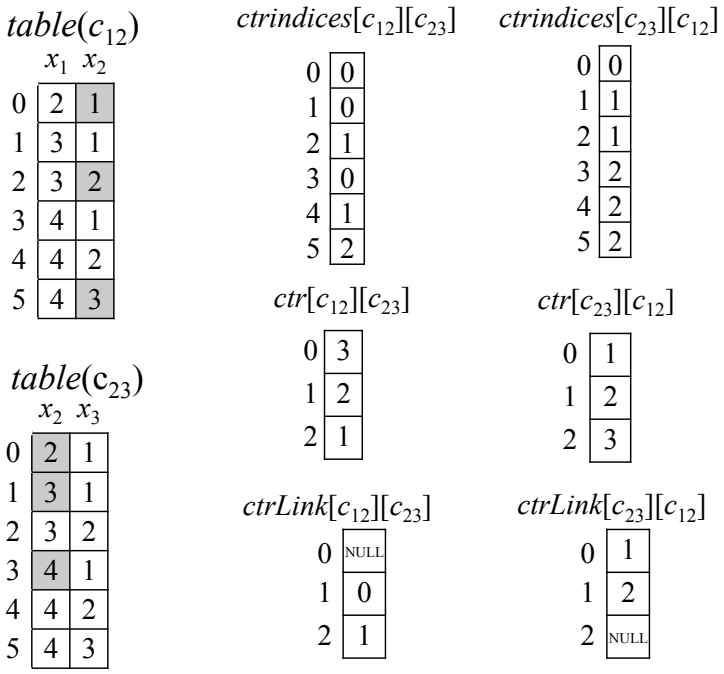


Figure 3.10: An example of the data structures used by eSTR

(2), appears for the first time in $table(c_{12})$ at index 2. Because this second value of the sub-tuple, (2), appears also in $table(c_{12})[4]$, we have $ctrindices[c_{12}][c_{23}][2] = ctrindices[c_{12}][c_{23}][4] = 1$.

- $ctr[c_i][c_j]$ is an array that stores, for each of the sub-tuples in I, the number of active tuples in which it appears in c_i. For example, assuming all the tuples of $table(c_{12})$ are active, in $ctr[c_{12}][c_{23}]$, the first value of the sub-tuple (x_2) is (1) and occurs three times in $table(c_{12})$, the second value is (2) and occurs twice, and the third value is (3) and occurs once.

- $ctrLink[c_i][c_j]$ is an array of size $|ctr[c_i][c_j]|$ that links $ctr[c_i][c_j]$ to $ctr[c_j][c_i]$. For each counter $ctr[c_i][c_j][k]$ corresponding to a value of the sub-tuple for the variables I, $ctrLink[c_i][c_j][k]$ holds the corresponding index value in the counter in $ctr[c_j][c_i]$ that is associated with that value of the sub-tuple. If the value of

the sub-tuple is not included in any sub-tuple, then $ctrLink[c_i][c_j][k]$ is set to $NULL$. For example, $ctrLink[c_{12}][c_{23}][0] = NULL$ because the value of the sub-tuple in $\pi_{x_2}(table(c_{12}))$ indexed at 0, which is (1), is not a member of $\pi_{x_2}(table(c_{23}))$. Similarly, $ctrLink[c_{12}][c_{23}][1] = 0$ and $ctrLink[c_{12}][c_{23}][2] = 1$ because the values of the sub-tuples in $\pi_{x_2}(table(c_{12}))$ indexed at 1 and 2, which are (2) and (3) respectively, appear $\pi_{x_2}(table(c_{23}))$ at the indices 0 and 1, respectively.

eSTR* makes use of two functions in addition to the functions used by the original the algorithms that they improve (i.e., STR1 and STR2):

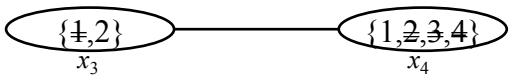
- *isPWconsistent* is a function called whenever a tuple $\tau \in table(c_i)$ is found to be valid. It iterates over each constraint c_j that intersects with c_i, and verifies whether τ has a PW-support in $table(c_j)$. It achieves this operation by using the structures $ctrindices[c_i][c_j]$ and $ctrLink[c_i][c_j]$ to locate the counter $ctr[c_i][c_j]$. Unless the function returns $NULL$ (indicating that there is no link) or 0 (indicating that there are no supports left), then τ has a PW-support in $table(c_j)$.

- *updateCtr* is called to update the counters associated with a constraint c_i whenever a tuple is removed from $table(c_i)$. For each constraint c_j that intersects with constraint c_i, we locate the index of the value of the sub-tuple k in $ctrindices[c_i][c_j]$. Then, we decrement the corresponding counter in $ctr[c_i][c_j][k]$ by 1. If this value becomes 0, then we know that some of the tuples in c_j have lost their PW-support, and that the two constraints need to be added to the propagation queue.

The data structures for the constraints c_{12} and c_{23} are illustrated in Figure 3.10.

eSTR1 algorithm is shown in Algorithm 9 in Appendix A. It calls *isPWconsistent* at line 7. After each call to *removeTuples*, it calls the function *updateCtr* at line 14.

Figure 3.11 illustrates an example of applying eSTR1. Suppose for some reason,



table(c_{34})

	x_3	x_4	*status*
0	2	1	0
1	3	1	1
2	3	2	1
3	4	1	1
4	4	2	1
5	4	3	1

Figure 3.11: An example using eSTR

we have reduced the domain of x_3 as shown. Next we run eSTR on the constraint c_{34} over variables x_3 and x_4. While iterating over the tuples, we find that the tuple at index 0 is valid and pw-consistent, but tuple 1 is not valid because the value 3 is no longer in the domain of x_3. Also, tuple at index 1 is not pw-consistent because $ctrLink(x_3)(x_4)[ctrindices(x_3)(x_4)(1)] = NULL$. As a result, we set the status at index 1 to 1 (i.e., tuple is deleted). We do the same for tuples at indices 2, 3, 4, and 5. Then, we update the domains of variables x_3 and x_4 accordingly.

Similarly to the previous STR algorithms, whenever backtracking occurs, appropriate values and tuples are restored. In addition, eSTR* restores all decremented counters when restoring the tuples.

The worst-case time complexity of eSTR1, is $\mathcal{O}(rd + max(r, g)t)$ where g is the number of intersecting constraints. The worst-case time complexity for restoring counters is $\mathcal{O}(gt)$. The worst-case space complexity of eSTR1 is $\mathcal{O}(n + max(r, g)t)$ per constraint.

3.3 Simple Tabular Reduction for Negative Table Constraints: STR-Ni

Li *et al.* [2013] introduces STR-N and STR-Ni, which implement simple tabular reduction on negative table constraints. Below, we discuss only STR-Ni, which improves STR-N. The main advantage of the STR-Ni algorithm over the above-discussed STR algorithms is that it operates directly on negative table constraints. Thus, we do not need to convert conflicts into supports, which results in better performance on loose constraints because the sizes of the tables of conflicts are smaller than that of the tables of supports.

Similarly to the STR1 and STR2 algorithms, STR-Ni dynamically maintains the constraint tables, and iterates over their tuples. The main difference between this method and the STR for positive tables is in the way validity checks are made (i.e., how a given variable-value pair is supported in a constraint). In STR-Ni, a tuple τ is a conflict[1] if and only if all the values in this tuple are present in the domains of their corresponding variables. That is, for a negative table constraint c and variable $x \in scp(c)$, if there exists a valid tuple τ involving a value (x, a) and $\tau \notin table(c)$, then there is at least one support for the value (x, a) on c.

STR-Ni (Algorithm 10 in Appendix A), uses the same data structures used for STR1 (a.k.a. GACstr, Algorithm 2 in Appendix A), except for the list of supported domain values *gacValues*. An additional data structure $count((x, a), c)$, which is the number of supports of (x, a) in $rel(c)$, is maintained for all variable-value pairs. $count((x, a), c)$ is initialized to the product of the current domain sizes of all the variables in $scp(c)$ except for x.

[1]The paper inaccurately refers to τ as a valid tuple, but we refer to it as a conflict because it is the exact nature.

When a tuple is found to be a conflict while iterating over the tuples of a $table(c)$, $count((x, a), c)$ of for each (x, a) listed in the tuple is decremented by 1. If $count((x, a), c)$ is greater than 0, then the value (x, a) has at least one support in the constraint c. In the paper, although the authors dynamically maintain the negative tables by removing tuples whose values have all been filtered out from the variables domains, they do not learn new conflicts and add them to the constraints.[2]

When the smallest value of $count((x, a), c)$ for all variable-value pairs (x, a) in a given constraint c is greater than the number of no-goods alive in the constraint, then we are guaranteed that all the (x, a) have at least one support, and we do not need to check the constraint at all.

The worst-case space and time complexities of GACstr also apply to STR-Ni, except that t is the size of the negative table. Because $count((x, a), c)$ is simply an integer, its space requirement is $\mathcal{O}(nd)$.

3.4 A Comparative Analysis of STR Algorithms

STR1 iterates over the tuples of a constraint represented as a positive table. Whenever a tuple is found to have a value that is no longer in the domain of its corresponding variable, the tuple is removed. Conversely, values that still appear in the variables may lose their support following some tuple deletions. Those values are then removed from the corresponding domains. The process iterates until a fixed point is reached.

STR2 improves on STR1 in two ways:

1. Firstly, while going through the tuples of a constraint, as soon as all the values in the domain of a variables are 'covered' by the active tuples, we stop checking

[2]Conflict learning can learn new nogoods, which can be exploited to prune away sub-trees where those conflicts show up again in the search tree.

the remaining tuples.

2. Secondly, if the domain of a variable has not changed between two subsequent consistency calls on a constraint on that variable, then the values in the variable's domain are guaranteed to be GAC by the first call and need not be checked at the second call.

STR3 makes use of a complex representation of the table constraint. The representation associates the domain values of variables to the tuples they appear in. Invalid tuples are partitioned into 'deleted' and 'unchecked or active,' rather than being removed from the table. This method of partitioning tuples helps us to avoid repeated consistency checks for the same values.

eSTR* uses the same techniques as STR1 and STR2 to achieve GAC. In addition, it also achieves pair-wise consistency. eSTR* stores the number of times that each sub-tuple appears in the intersection of any two constraints, by creating a counter structure that stores the number of active tuples in a constraint.[3]

STR-Ni uses the same technique as STR1, but operates on negative tables (conflicts). Unlike STR1, where an active tuple confirms that the value of a variable has a support, in STR-Ni it confirms that the value has one less support. STR-Ni is particularly advantageous for problems where the relation of a constraint has more conflicts than supports.

[3]This counter operation is reminiscent of AC-4 [Mohr and Henderson, 1986], which is theoretically optimal but overly costly in practice [Wallace, 1993].

3.5 A Comparative Analysis of STR Algorithms and GAC2001

Based on the attributes of the already listed algorithms, we are able to point out some obvious differences:

- While both GAC2001 and STR algorithms work with tuples, GAC2001 does not maintain tuples in tables like the STR algorithms do.
- While 'action' in GAC2001 is triggered by deletion of domains values, constraint propagation in STR-based algorithms is triggered by both domains values and relations tuples deletions.

3.6 GAC for MDD-Represented Constraints

STR-like algorithms operate on table constraints. However, the size of a table constraint is exponential in the arity of the constraint. Both Gent *et al.* [2007] and Katsileros and Walsh [2007] propose to use tries as a more compact representation than tables. Cheng and Yap [2010] notes that updating both table and trie representations (to remove deleted tuples) can be costly, and proposed to use instead *multi-valued decision diagram (MDD)* to represent constraints. They introduce the coarse-grained mddc algorithm (Algorithm 11 in AppendixA) to enforce GAC on this representation. They argue that mddc may remove exponential number of invalid tuples in polynomial or possibly constant time, thus achieving full incrementality of constraint representation in constant time.

The MDD representation of a constraint c is denoted $MDD(c)$. Each time it is applied on a constraint c, the function $mddcSeekSupports()$ (lines 1 and 6 in

Algorithm 11 in Appendix A) creates, for each variable x in the scope of c, a set S of domain values of x that have no support in constraint c. (The set S of variable x is similar to the structure *gacValues* in STR1 and STR2.) S is initialized with all the values in the current domain of x. *mddcSeekSupports* traverses the MDD constraint $MDD(c)$ recursively, updating S as it progresses. After the traversal of $MDD(c)$, we update $dom(x) \leftarrow dom(x) \setminus S$. In *mddcSeekSupports*, the function explores the sub-MDD rooted at a given node. If this node corresponds to a leaf-node or has already been explored, the node can be detected as supported or not. Otherwise, the children of the nodes are explored (*mddcSeekSupports* are called on them) as long as the value labeling the outgoing edge is still in the current domain of the corresponding variable. When a supported child node is found, both the parent node and the value labeling the arc are supported.

For example, suppose we have already reduced the domain of variable x_2 as shown in Figure 3.12, and we then invoke mddc on $MDD(c_{23})$. The resulting sets S are as follows:

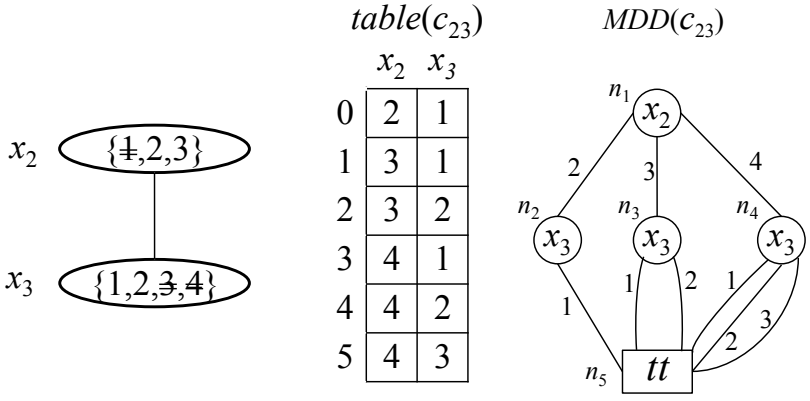


Figure 3.12: An example showing a constraint c_{12}, $table(c_{12})$, and MDD(c_{12}

- $S(x_2) \leftarrow \{1\}$
- $S(x_3) \leftarrow \{3, 4\}$

The data structures used for caching visited MDD nodes and that support the incrementality of mddc during search are as follows:

- Σ^{YES} contains nodes of $MDD(c)$ such that sub-MDD rooted at each node is satisfiable. From the example in Figure 3.12, we have $\Sigma^{YES} \leftarrow \{n_1, n_2, n_3\}$ because the sub-tuples rooted n_2 and n_3 are supported. Note that even though the edge labeled with the value 4 and point out of n_1 correspond to a value that is no longer in the domain of x_2, it remains supported because it has at least one child node that is supported.

- Σ^{NO} contains nodes of $MDD(c)$ such that the sub-MDD rooted at each node is unsatisfiable. From the example in Figure 3.12, we have $\Sigma^{NO} \leftarrow \{n_4\}$. n_4 is not supported because the value labeling the edge incident to n_4 is no longer in the domain of its variable x_2.

  Because unsatisfiable values remain unsatisfiable as more variables are assigned, mddc achieves incrementality by maintaining a sequence of sets $\Sigma_1^{NO}, \ldots, \Sigma_d^{NO}$ during search, where $1, \ldots, d$ denotes the search depth.

The worst-case time complexity of enforcing mddc on a constraint c is $\mathcal{O}(e_{mdd(c)} + \lambda)$ where $e_{mdd(c)}$ is the number of edges in $MDD(c)$ and λ the number of values detected GAC-inconsistent.

3.7 Our Contribution: A Hybrid STR (STR-h)

All of the studied STR algorithms either work on positive tables (e.g., STR1, STR2, STR3, and eSTR*) or negative tables (e.g., STR-N and STR-Ni), without taking into consideration what is the best representation of each individual relation.

We propose to mix the constraints representations as supports or conflicts depending on the initial tightness of each individual constraint relation. To this end, we check the size of each relation as provided in the problem description, and choose the representation with the smallest number of tuples given the maximum number of tuples in the relation, which is equal to the size of the Cartesian product of the domain sizes. Note that, during propagation, the tables of both positive and negative tables decrease monotonically. Thus, once an initial representation is chosen, this choice does not need to be revised during problem solving.

For example, suppose we have a constraint c, and the total number of all possible tuples in c is 200. Now, suppose the constraint depicts a positive table $table(c)$ of size 150, it is immediately visible that the negative table of the constraint c will have a table size of 50. Obviously, in this case, it is more efficient to use a negative-table representation and use an algorithm that operates on negative tables, such as is STR-Ni.

We build our hybrid algorithm (STR-h) as a combination of an STR algorithm that operates on positive tables and one that operates on negative tables. We choose to use STR1 and STR-Ni because the data structures on which they rely are compatible, which facilitates interoperability.

The worst-case time complexity of the selection for the representation of the table constraint is $\mathcal{O}(e)$. The time and space complexity of STR-h remains the same as that for STR1.

Summary

In this chapter, we introduced and examined the data structures and mechanisms of algorithms for achieving Generalized Arc Consistency (GAC2001, STR1, STR2,

STR3, eSTR, STR-N, and mddc). We also introduced a hybrid algorithm that advantageously combines STR1 and STR-Ni based on the size of the most appropriate representation.

Chapter 4

Empirical Evaluations of GAC Algorithms

In this chapter, we empirically evaluate the performance of the algorithms discussed in Chapter 2. Below, we describe our experimental set-up, conduct a comparative empirical analysis on randomly generated CSPs including the phase-transition, then discuss in detail the results on individual benchmark problems. Lastly, we summarize our results.

We state a warning from the outset regarding our implementation of the STR3 and mddc algorithms. Indeed, when comparing our results for those algorithms to those reported in the literature [Cheng and Yap, 2010; Lecoutre, 2011; Lecoutre *et al.*, 2012], we realized that our implementation of STR3 and mddc is far from reaching the efficacy reported in those papers. Our implementation is guaranteed correct because it does the same amount of filtering as the other GAC algorithms tested and visits the same number of nodes in the tree. However, the CPU time of our implementation is significantly larger than the CPU time reported in the literature for the same problem instance. we believe that our implementation *must be re-examined to improve its*

efficiency before we can confidently state any conclusions about the performance of STR3 or mddc. Examining the reasons of this poor performance requires more effort than allowed in the time span of this thesis, and is left out for future investigation.

4.1 Experimental Setup

We evaluate and compare the performance of the following algorithms to enforce real full look-ahead in a backtrack search that finds the first solution of a CSP:

- GAC2001
- GACstr and its variations: STR1, STR2, STR3, STR-Ni, eSTR1 and eSTR2.
- mddc

We report and compare performance in terms of CPU time.

4.2 Randomly Generated Non-Binary CSPs

We compare the performance of the above-listed algorithms on randomly generated non-binary CSPs. Generators of random CSPs generate uniform instances while maintaining some parameters of the CSP constant (e.g., number of variables and domain size) and varying other parameters (e.g., constraint tightness and constraint density). In particular, they allow us to generate instances that 'traverse' the phase-transition region known to characterize the difficulty of solving CSPs (see Section 2.6 and Figure 2.7). We use the random generator RBGenerator 2.0 [Xu *et al.*, 2007], which is based on the modelRB.[1] In this generator, a random instance of a non-binary CSP is characterized by the following input parameters:

[1]The RBGenerator is available from `http://www.cril.univ-artois.fr/~lecoutre/software.html\#`

- n: number of variables
- d: uniform domain size
- k: uniform arity of the constraints
- p: constraint density, i.e., the ratio between the number of constraints in the problem and the number of all possible constraints involving k variables.
  e: the number of constraints in the problem. (e and p are redundant.)
- q: uniform constraint looseness, i.e., the ratio of allowed tuples to d^k, which is the maximum number of tuples of a constraint.
  t: the constraint tightness and is uniform for all constraints. Note that $q = 1-t$.

Randomly generated instances are denoted with the tuple $(n, d, k, p(e), t)$. We maintain all parameters constant and vary the constraint tightness $t \in [0.1, 0.2, \ldots, 0.9]$, generate 30 instances per parameter configuration, and average our results over the 30 instances.

In Table 4.1, we report the CPU performance of all nine algorithms on random CSPs generated with ($n = 60$,$d = 5$,$k = 3$,$p = 0.4$,t), averages over 30 instances.

Table 4.1: CPU time (in milliseconds) for solving randomly generated CSPs, averaged over 30 instances with ($n = 60$,$d = 5$,$k = 3$,$p = 0.4$,t)

Tightness	**Algorithms**								
	GAC2001	STR1	STR2	STR3	eSTR1	eSTR2	STR-Ni	STR-h	mddc
$t = 0.1$	**0.00**	0.71	**0.00**	20.36	81.79	81.43	**0.00**	1.43	13.57
$t = 0.2$	1.00	4.33	4.00	16.33	78.00	86.33	**0.00**	2.67	19.33
$t = 0.3$	3.33	6.33	6.00	18.33	87.67	94.00	**0.67**	6.33	21.00
$t = 0.4$	18.33	19.33	22.00	39.33	59.67	89.00	**17.00**	19.33	38.00
$t = 0.5$	5887.50	5809.29	6381.43	8788.93	**49.64**	50.36	6687.14	5908.93	7877.86
$t = 0.6$	52.00	56.33	59.33	95.67	**40.67**	41.00	84.00	58.33	110.00
$t = 0.7$	**5.00**	5.33	6.00	11.00	37.00	36.67	11.67	5.33	15.00
$t = 0.8$	**0.00**	**0.00**	**0.00**	9.00	25.33	30.33	6.33	**0.00**	9.67
$t = 0.9$	**0.00**	**0.00**	**0.00**	**0.00**	19.67	18.67	0.33	**0.00**	2.67

We discuss the above results from the following perspectives:

- Comparing STR* algorithms against GAC2001
- Comparing eSTR1 and eSTR2 against STR1 and STR2
- Comparing STR-h against STR1 and STR-Ni
- Comparing mddc against STR2 and eSTR1

4.2.1 Comparing STR* algorithms against GAC2001

In Figure 4.1, we compare the performances of GAC2001, STR1, STR2, and STR3.

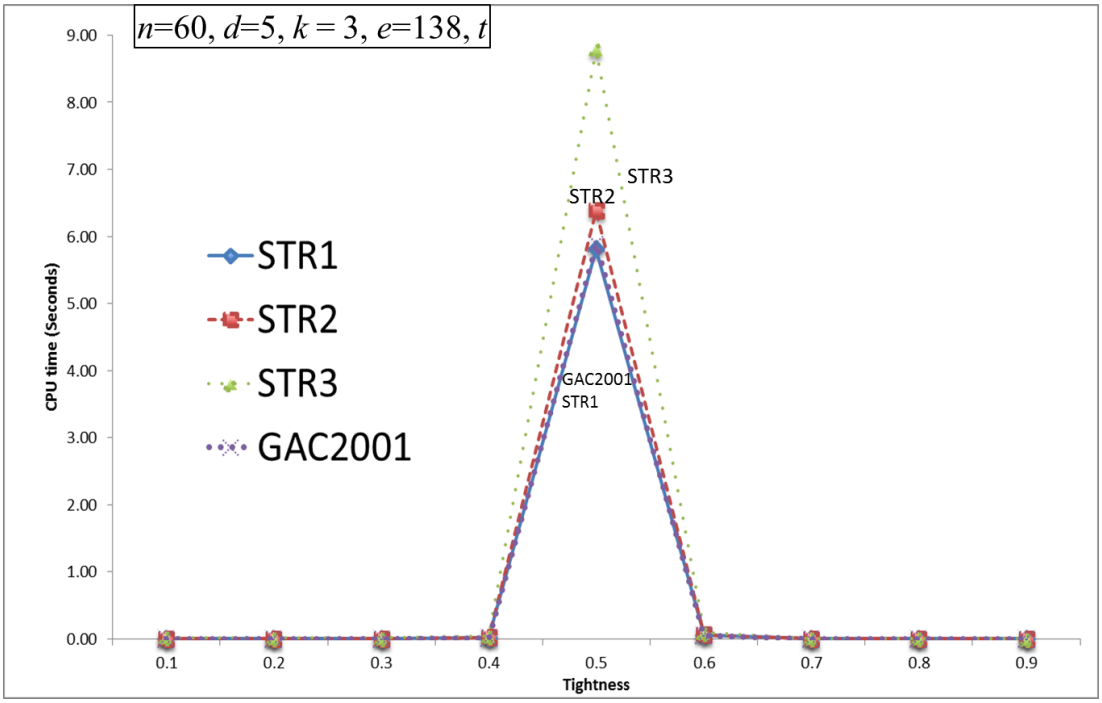


Figure 4.1: Phase-transition chart with parameters ($n = 60$,$d = 5$,$k = 3$,$p = 0.4$,t)

- STR3 shows an excessive cost around the phase transition but comparable cost to STR1 and STR2 outside the phase transition. We attribute the poor per-

formance of STR3 at the phase transition to our own implementation, which requires revision as stated at the beginning of this chapter.

- STR2 does not perform as well as STR1 or GAC2001. Although STR2 was presented as an improvement over STR1, when neither of the two conditions for improvements hold (see Section 3.2.2), STR2 does more work than STR1 because of the additional data structures that STR2 maintains.

- GAC2001 and STR1 exhibit similar performances in Figure 4.1. While they both slightly outperform STR2 around the phase transition area, none of the algorithms statistically outperforms the other.[2]

In an effort to reproduce the results provided by Lecoutre [2011], we compare GAC2001, STR1, and STR2 on random problems generated with the same CSP parameters (i.e., $n = 60$,$d = 2$,$k = 13$,$e = 20$,t). Those instances have larger constraints arity than the instances shown in Figure 4.1. The detailed numerical results are given in Table 4.2 and the chart is shown in Figure 4.2.

Table 4.2: CPU time (in milliseconds) for solving randomly generated CSPs, averaged over 30 instances with ($n = 60$,$d = 2$,$k = 13$,$e = 20$,t)

Tightness	**Algorithms**		
	GAC2001	STR1	STR2
$t = 0.80$	757.33	413	**407.33**
$t = 0.82$	2,376.33	914.67	**900.00**
$t = 0.84$	8,395	3,446.67	**3,383.67**
$t = 0.86$	28,725.17	10,946.55	**9,407.50**
$t = 0.88$	47,577.08	20,828.33	**18,833.04**
$t = 0.90$	36,823.21	18,510.00	**12,638.00**
$t = 0.92$	15,812	7,042.67	**6,534.33**
$t = 0.94$	4,444.33	**2039.00**	2064.00
$t = 0.96$	980.33	546.00	**494.00**

[2]According to a paired t-test with 95% confidence.

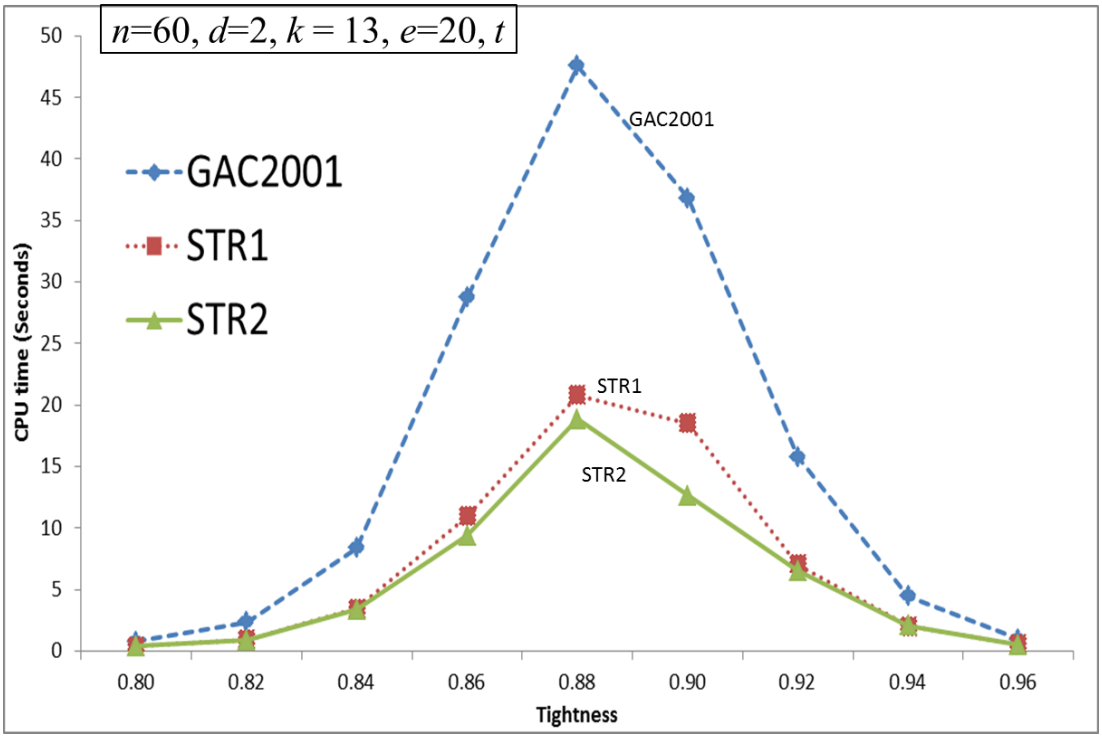


Figure 4.2: Phase-transition chart with parameters ($n = 60$,$d = 2$,$k = 13$,$e = 20$,t)

STR2 clearly outperforms both GAC2001 and STR1. Further, STR1 largely outperforms GAC2001. Those results are consistent with the ones reported by Lecoutre [2011].

In conclusion, our observations suggest that the advantages of STR1 over GAC2001 and those of STR2 over STR1 become more significant as the size of the problem or the arity of the constraints increases.

4.2.2 Comparing eSTR1 and eSTR2 against STR1 and STR2

In Figure 4.3, we compare the performances of the extended STR algorithms (eSTR1 and eSTR2) to STR1, which is cheaper than STR2 on these instances: Although not visible in Figure 4.3, but can clearly seen in Table 4.1, STR1 is cheaper than eSTR1 and eSTR2 outside the phase transition because of the additional data structures

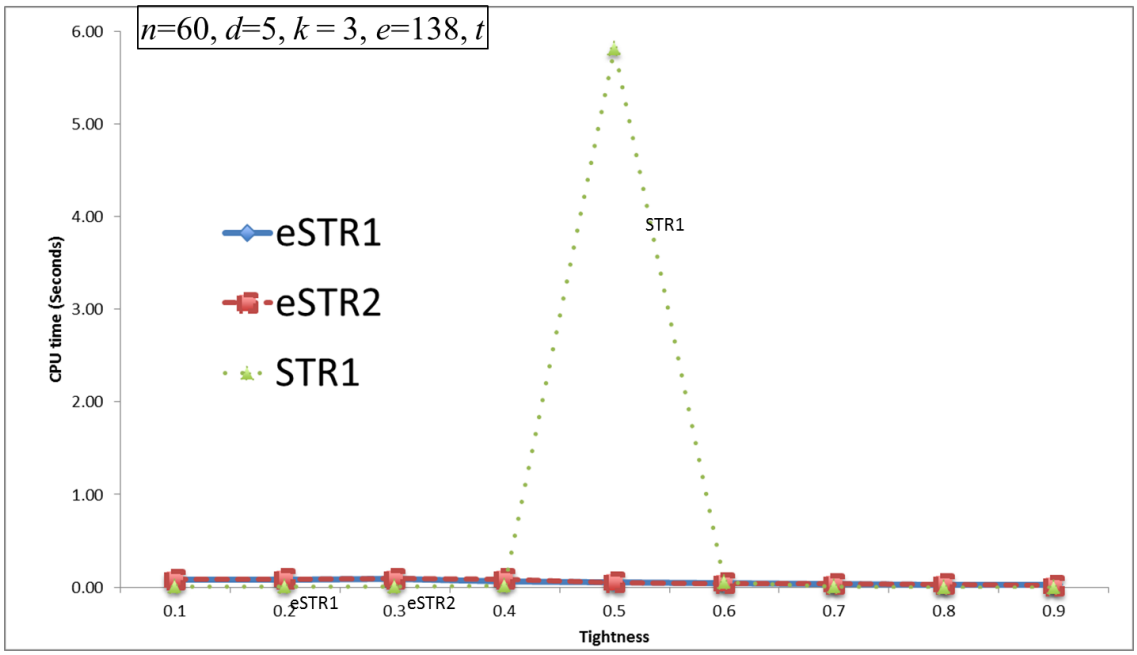


Figure 4.3: Phase-transition chart with parameters ($n = 60$,$d = 5$,$k = 3$,$e = 138$,t)

that eSTR1 and eSTR2 need to initialize and maintain. The advantages of eSTR* become visible as the problem becomes harder around the area of the phase transition. Indeed, eSTR* outperforms STR1 by a huge margin (>5000 milliseconds).

In order to show the advantages of eSTR1 and eSTR2 over STR1 and STR2 on difficult problems, we test those four algorithms on the following randomly generated problems, increasing the domain and constraint table size: $n = 60$,$d = 15$,$k = 3$,$e = 228$,$t \in \{0.1, 0.2, \ldots, 0.9\}$. Table 4.3 reports report the CPU performance averaged over 30 instances, and Figure 4.4 shows the corresponding charts.

From Table 4.3 and Figure 4.4, we clearly see that eSTR1 eSTR2 easily solve the instances at the phase transition, where STR1 and STR2 cannot even complete with the two-hours time window per instance. Further, eSTR1 outperforms eSTR2 between tightness $t = 0.2$ to $t = 0.7$ for the same reason that STR1 outperforms STR2 (i.e., neither of the two conditions for improvement hold as discussed in Section 3.2.2).

Table 4.3: CPU time (in milliseconds) for solving randomly generated CSPs, averaged over 30 instances with ($n = 60$,$d = 15$,$k = 3$,$e = 228$,t)

Tightness	**Algorithms**			
	STR1	STR2	eSTR1	eSTR2
$t = 0.1$	79.33	**61.33**	6,508.67	6,320.33
$t = 0.2$	100.67	**67.33**	5,813.45	5,650.00
$t = 0.3$	**56.67**	68.67	4,921.43	9,790.33
$t = 0.4$	16,712.00	16,436.00	**7,272.50**	18,010.00
$t = 0.5$	timed out	timed out	**3,264.12**	12,530.00
$t = 0.6$	timed out	timed out	**2,780.00**	7,442.50
$t = 0.7$	timed out	timed out	2,138.33	**2,136.33**
$t = 0.8$	16,760.00	32,071.33	**1,324.00**	1,325.33
$t = 0.9$	**50.67**	66.00	697.00	698.67

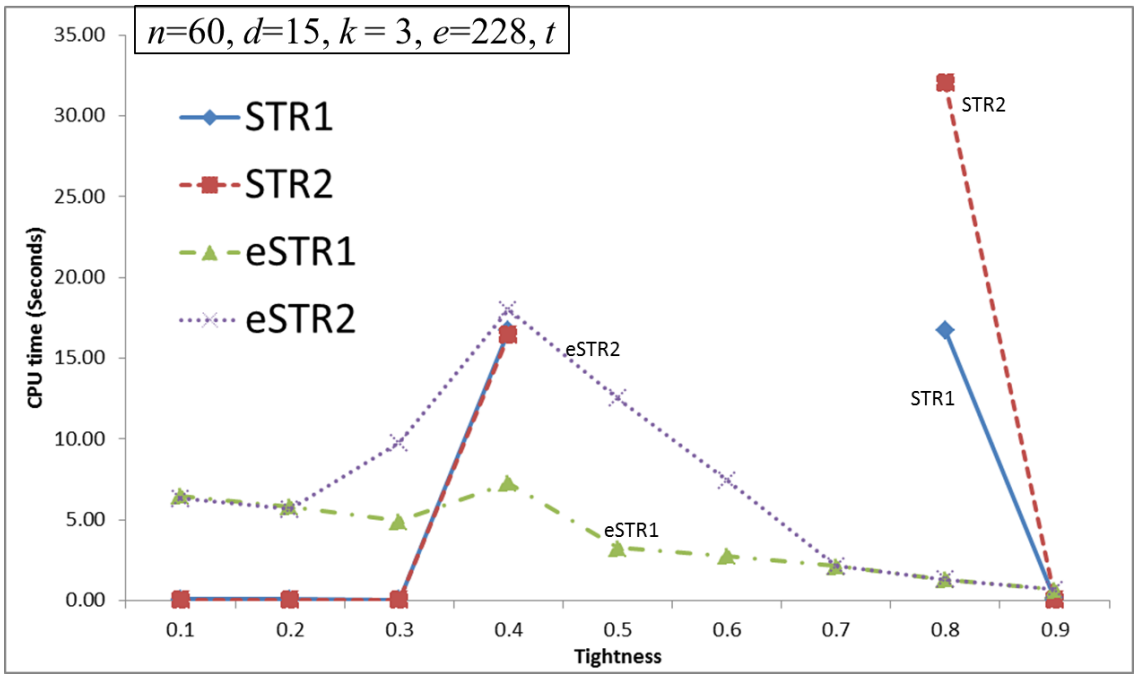


Figure 4.4: Phase-transition chart with parameters ($n = 60$,$d = 15$,$k = 3$,$e = 228$,t). Note that STR1 and STR2 time out for $t = 0.5, 0.6, 0.7$

4.2.3 Comparing STR-h against STR1 and STR-Ni

In Figure 4.5, we compare our hybrid algorithm STR-h to the algorithms that it combines (i.e., STR1 and STR-Ni). STR-h and STR1 exhibit comparable performances in Figure 4.5. Around the phase-transition (around $t = 0.5$), STR-h outperforms

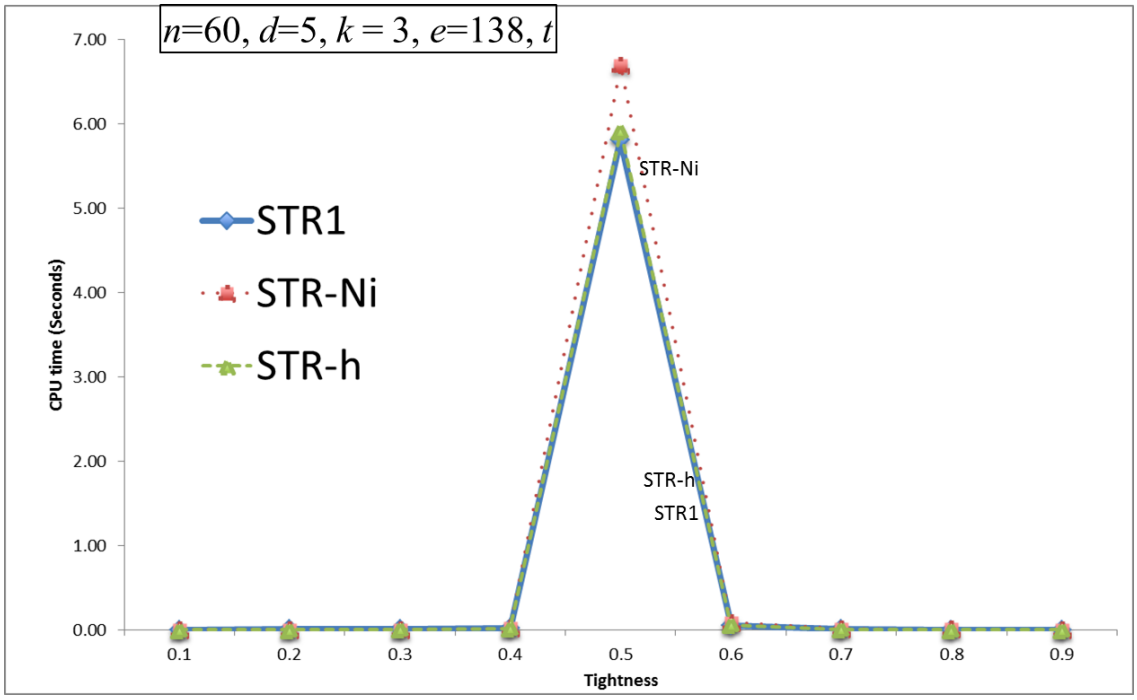


Figure 4.5: Phase-transition chart with parameters ($n = 60$,$d = 5$,$k = 3$,$e = 138$,t)

STR-Ni. This experiment and the other experiments conducted in this thesis show that STR-h typically performs better than the worse of STR1 and STR-Ni when the performances of STR1 and STR-Ni differ. In all other cases, the three algorithms have comparable performances.

4.2.4 Comparing mddc against STR2 and eSTR1

In Figure 4.6, we compare the mddc algorithm to a basic STR algorithm (STR2) and an extended algorithm (eSTR1). We choose STR2 and eSTR1 because their performances on those instances are worse than STR1 and eSTR2, respectively. We conclude that, while mddc outperforms eSTR1 outside the phase transition area (see Table 4.1), it performs worse than both STR2 and eSTR1 around the phase-transition area. We attribute this poor performance at the phase transition to our own imple-

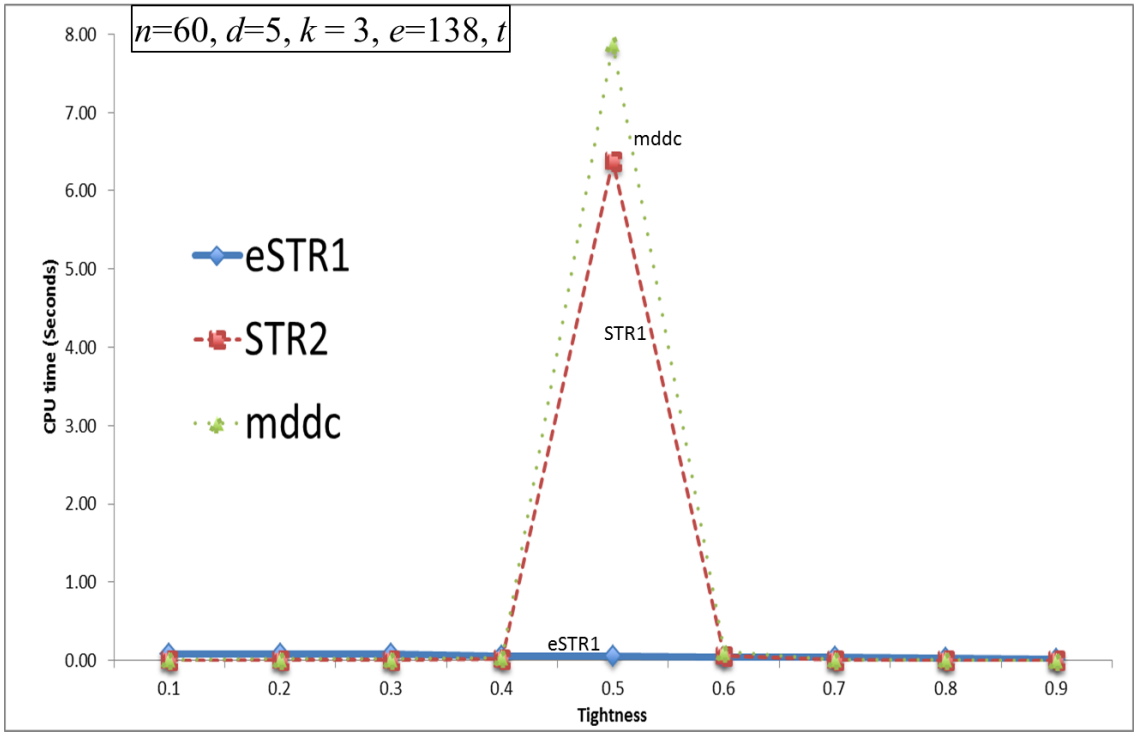


Figure 4.6: Phase-transition chart with parameters ($n = 60$,$d = 5$,$k = 3$,$e = 138$,t)

mentation, as stated at the beginning of this chapter.

4.3 Benchmark Problems

We run our experiments on 2875 benchmark instances of extensionally-defined constraints taken from the CSP Solver Competition.[3] In [2010], Lecoutre gives a description of the benchmark problems. We limit the CPU time to two hours per instance and the memory to 8GB. The 2875 instances comprise 1915 binary instances and 960 non-binary instances. They are organized as follows:

- The instances are grouped in 86 benchmarks.
- Each benchmark class has a number of instances (between 4 to 100).

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

- Each benchmark is further classified into one of the following 'qualitative' categories proposed by Lecoutre.[4] Those categories are: academic, assignment, Boolean, crossword, Latin square, quasi-random (random benchmarks that have some structure), random, and TSP.

We do not report results of experiments on benchmarks that did not complete because of:

1. *Insufficient memory.* When the size of the relations/tuples is too large to store given the available memory. This situation arises for the case of the following benchmarks: bddSmall, lard.

2. *Insufficient time.* The tested algorithms could not solve in the allocated time enough instances (only 1 or 2 instances for rand-2-27, rand-3-28-28, and rand-3-28-28-fcd), or any instances at all (BH-4-13, BH-4-4, BH-4-7, bqwh-15-106, bqwh-18-141, frb53-24, frb56-25, frb59-26, QCP-20, QCP-25, QWH-20, QWH-25, rand-3-24-24, renault). Typically, those are difficult problems that require the use of higher-level consistency algorithms [Karakashian *et al.*, 2010; Woodward *et al.*, 2011; Karakashian, 2013; Karakashian *et al.*, 2013; Schneider *et al.*, 2014].

Below, we first report a *summary* ranking of the performance of the algorithms on *each* tested benchmark class, then discuss ranking of three individual representative benchmark classes.

The tables with the detailed ranking of all algorithms on each benchmark can be found in Tables B.1 to Tables B.54 in Appendix B.

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

4.3.1 Method for statistical analysis

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

To compute the significance classes between the algorithms, we generate a generalized linear mixed-model for each algorithm on a given benchmark. While generalized linear mixed models do not require that the data be normally distributed, they do not take into account censored data. The models assume that random effects are normally distributed. We use those models to construct an approximate t-test (Wilcoxon) between each pairs of algorithms. Even if the random effects assumption may not hold for our data, our analysis yielded consistent results on the various benchmarks, thus supporting the correctness of our conclusions. For computing the significance of the CPU measurements, the CPU time of each algorithm on a given instance is given as input to the model. We assume all censored data points finished at the maximum cutoff time.

4.3.2 Comparison criteria used

Below, we list the criteria used to compare the various algorithms in Tables 4.4, 4.5, and 4.6:

Category denotes the category of the benchmark as listed in the provider's site.

Table gives the reference of the table where the detailed results of the benchmark are reported in Appendix B.

#I gives the (original) number of instances in the benchmark.

Best CPU lists the algorithms that statistically outperform all others in terms of CPU time (i.e., the algorithms are ranked to be in the same statistical class).

Fastest denotes the algorithms that solved the largest number of instances the fastest.

#Comp stands for 'completed,' denotes the algorithms that solved the largest number of instances.

#BTF denotes the algorithm that solved the largest number of instances in a backtrack-free manner.

All in any column indicates that all of the algorithms are equivalent with respect to that metric.

4.3.3 Ranking all benchmark classes

Using the criteria listed in Section 4.3.2, we rank the algorithms in Tables 4.4 and 4.5 for binary CSPs and Table 4.6 for non-binary CSPs. We make the following observations based on those results:

4.3.3.1 GAC2001

GAC2001 performs best on a variety of *binary* random-instances in terms of number of instances completed (#Comp), CPU time (Best CPU), and number of instances solved fastest (Fastest), see Tables 4.4 and 4.5. Although GAC2001 typically has a good performance on many non-binary instances, for example, it completes more instances (#Comp) than most of the basic STR algorithms (e.g., STR1, STR2, STR3, STR-Ni), it is mostly not the best in terms CPU (i.e., Best CPU and Fastest).

Table 4.4: Overview of the binary benchmarks tested (Part A)

Category	Benchmark	Table	#I	Best CPU	Fastest	#Comp	#BTF
Academic	coloring	B.31	22	eSTR1 eSTR2 STR-Ni	STR-Ni	GAC2001, STR1, STR3, eSTR1, eSTR2, STR-Ni, mddc	All
	hanoi	B.33	5	All	STR2	GAC2001, STR1, STR2, eSTR1, eSTR2, STR-h	All
	langford	B.12	4	All	STR-Ni	GAC2001, STR1, STR3, STR-h, STR-Ni	All
Assignment	driver	B.7	7	STR-Ni	STR-Ni	GAC2001, STR-Ni	All
Latin square	QCP-10	B.6	15	GAC2001, STR1, STR2, STR-Ni, STR-h, mddc	STR-Ni	GAC2001, STR-Ni	All
	QCP-15	B.14	15	All except STR3	GAC2001	GAC2001	None
	QWH-10	B.34	10	GAC2001, STR1, STR2, STR-Ni, STR-h, mddc	GAC2001	All	All
	QWH-15	B.35	10	GAC2001, STR-Ni	STR-Ni	GAC2001, STR-Ni	None
Quasi-random	composed-25-1-80	B.1	10	All	GAC2001	GAC2001	
	composed-25-10-20	B.2	10	All	STR-Ni	GAC2001, STR-Ni	None
	composed-75-1-80	B.3	10	All	STR-Ni, GAC2001	All	None
	ehi-85	B.4	100	GAC2001	GAC2001	GAC2001	None
	ehi-90	B.5	100	GAC2001	GAC2001	GAC2001	None
	geom	B.11	100	GAC2001, STR-Ni	GAC2001	GAC2001	All

4.3.3.2 STR-based algorithms

The basic STR algorithms (STR1, STR2, STR3, STR-Ni) perform well in terms of all criteria on the following binary benchmarks: Academic, Assignment, Latin

Table 4.5: Overview of the binary benchmarks tested (Part B)

Category	Benchmark	Table	#I	Best CPU	Fastest	#Comp	#BTF
Random	frb30-15	B.32	10	All except eSTR2	GAC2001	All	None
	frb35-17	B.8	10	GAC2001, STR-Ni	GAC2001, STR-Ni	GAC2001, STR-Ni	None
	frb40-19	B.10	10	GAC2001, STR-Ni	GAC2001, STR-Ni	GAC2001	None
	frb45-21	B.9	10	GAC2001	GAC2001	GAC2001	None
	marc	B.13	10	GAC2001, STR1, STR2, STR-Ni, STR-h	GAC2001, STR2	GAC2001, STR1, STR2, STR-Ni, STR-h	All
	rand-2-23	B.15	10	GAC2001	GAC2001	GAC2001	None
	rand-2-24	B.16	10	GAC2001, STR-Ni	GAC2001	GAC2001	None
	rand-2-25	B.17	10	GAC2001	GAC2001	GAC2001	None
	rand-2-26	B.18	10	GAC2001	GAC2001	GAC2001	None
	rand-2-30-15-fcd	B.20	50	GAC2001, STR-Ni	GAC2001	All	None
	rand-2-30-15	B.21	50	GAC2001, STR-Ni	GAC2001	GAC2001, STR1, STR2, STR3, eSTR2, STR-h, STR-Ni	None
	rand-2-40-19-fcd	B.22	50	GAC2001	GAC2001	GAC2001	None
	rand-2-40-19	B.23	50	GAC2001	GAC2001	GAC2001	None
	tightness0.1	B.24	100	GAC2001, STR-Ni	STR-Ni	GAC2001, STR-Ni	None
	tightness0.2	B.25	100	GAC2001, STR-Ni	STR-Ni	GAC2001, STR-Ni	None
	tightness0.35	B.26	100	GAC2001, STR-Ni	STR-Ni	GAC2001	None
	tightness0.5	B.27	100	GAC2001	GAC2001	GAC2001	None
	tightness0.65	B.28	100	GAC2001	GAC2001	GAC2001	None
	tightness0.8	B.29	100	GAC2001	GAC2001	GAC2001	None
	tightness0.9	B.30	100	GAC2001	GAC2001	GAC2001	None

square, and Quasi-random instances (Table 4.4). Among those algorithms, STR-Ni and STR2 occasionally stand out, outperforming the other STR-based algorithms: For example, STR-Ni and STR2 solve the largest number of instances the fastest (see column 'Fastest' in Table 4.4). STR2 outperforms STR-Ni when the positive table is smaller than the negative table (i.e., tight constraint), for example the benchmark hanoi. Conversely, STR-Ni outperforms STR2 when the negative table is smaller than the positive table (i.e., loose constraint), for example on the benchmarks coloring,

Table 4.6: Overview of the non-binary benchmarks tested

Category	Benchmark	Table	#I	Best CPU	Fastest	#Comp	#BTF
Assignment	modifiedRenault	B.54	50	All except STR3, STR-Ni, mddc	GAC2001	eSTR1, eSTR2	eSTR1, eSTR2
Boolean	aim-100	B.37	24	eSTR1, eSTR2	eSTR1	eSTR1, eSTR2	eSTR1, eSTR2
	aim-200	B.38	24	eSTR1, eSTR2	eSTR1	eSTR2	eSTR1, eSTR2
	aim-50	B.36	24	STR-Ni, eSTR1, eSTR2	eSTR1	All	eSTR1, eSTR2
	dubois	B.39	13	GAC2001, STR-Ni	GAC2001	GAC2001	None
	jnhSat	B.42	16	GAC2001, STR2, STR-Ni, STR-h	STR-Ni	All except STR1, STR-h, mddc	eSTR1, eSTR2
	jnhUnsat	B.43	34	STR-Ni	STR-Ni	All except STR1, STR-h, mddc	eSTR1, eSTR2
	pret	B.50	8	GAC2001, STR3, STR-Ni, STR-h	GAC2001	GAC2001, STR3, STR-Ni, STR-h	None
	ssa	B.40	8	All	mddc	eSTR1, eSTR2	eSTR1, eSTR2
	varDimacs	B.53	9	All	STR-Ni	GAC2001	All
Crossword	lexVg	4.9	63	GAC2001, STR1, STR2, eSTR1, eSTR2	STR2	GAC2001	All
	ogdVg	B.47	65	GAC2001, STR1, STR2, eSTR1, eSTR2	STR2	GAC2001	All
	ukVg	B.48	65	STR1, eSTR1, STR-Ni, STR-h, mddc	STR-Ni	GAC2001, STR-Ni	None
	wordsVg	B.49	65	GAC2001, STR2, eSTR1, eSTR2	STR2	GAC2001	All
Quasi-random	dag-rand	4.8	25	eSTR1, eSTR2	None	GAC2001, STR1, STR2, eSTR1, eSTR2	eSTR1, eSTR2
Random	rand-10-20-10	B.51	20	STR1, STR2, eSTR1, eSTR2	STR2	GAC2001, STR1, STR2, STR3, mddc	eSTR1, eSTR2
	rand-3-20-20-fcd	B.44	50	GAC2001, eSTR1, eSTR2	eSTR1	GAC2001	eSTR1
	rand-3-20-20	B.45	50	GAC2001, eSTR1, eSTR2	eSTR1	GAC2001	eSTR1
	rand-3-24-24-fcd	B.46	50	eSTR1, eSTR2	eSTR1	eSTR1	eSTR1
	rand-8-20-5	B.52	20	eSTR1	eSTR1	GAC2001	eSTR1
TSP	travellingSalesman-20	4.10	15	All except STR3, STR-Ni, mddc	GAC2001	GAC2001	All
	travellingSalesman-25	B.41	15	GAC2001	GAC2001	GAC2001	None

langford, QCP-10, QWH-15, composed-25-10-20, and composed-75-1-80. STR-Ni performs best in terms of CPU time on some instances (e.g., composed-25-10-20 of Table B.2), which can be attributed to the constant-time operation of using *count* as opposed to the linear-time traversal of $gacValues$. Out of all the basic STR algorithms (STR1, STR2, STR3,

Out of all the basic STR algorithms (STR1, STR2, STR3, STR-Ni), only STR2 and STR-Ni stand out on the non-binary benchmarks of Table 4.6. In particular, STR2 performs well on the non-binary *crossword* category and the rand-10-20-10 benchmark ('Fastest' column of Table 4.6). STR-Ni solves the following non-binary benchmarks fastest: jnhSat, jnhUnsat, varDimacs, and ukVg ('Fastest' column of Table 4.6).

STR3 performs the worst of all the algorithms tested, in particular STR1 and STR2, which are similar in nature. Indeed, STR3 does not stand out in Tables 4.4, 4.5, or 4.6. STR3 becomes effective when the reduction rate of the constraint table during propagation remains high (i.e., when the tables are not reduced in size). Lecoutre *et al.* [2012] argues that STR3 performs well when the tables of the constraint remain really large (e.g., tables with more that 1,000 tuples) during search. In our experiments, we did not come across large problems that did not time out within 2 hours.

4.3.3.3 eSTR-based algorithms

The eSTR algorithms (eSTR1 and eSTR2) perform relatively well in terms of completed instances (#Comp) on the binary problems (Table 4.4 and 4.5). However, they do not outperform any of the other algorithms in terms of CPU time (i.e., Best CPU and Fastest) on those binary problems. The eSTR algorithms outperform all other algorithms on non-binary benchmarks (e.g., Assignment, Boolean, Quasi-random and

Random in Table 4.6) by solving more instances backtrack free (#BTF), which is directly traceable to the fact that they enforce a higher-level consistency (i.e., pair-wise consistency).

4.3.3.4 mddc

The only benchmark where mddc solves the largest number of instances fastest (column 'Fastest') is the 'ssa' benchmark of Table B.40 in Appendix B. Because the ssa benchmark is made up of Pseudo-Boolean instances, the MDD representation of such constraints is compact and advantageous.

4.3.3.5 Implementation efficacy

Table 4.7 shows the number of completed instances and those not completed because they run out of time or memory. The best performances are shown in boldface and the worse two performances are grey out. This table shows that GAC2001 is the best algorithm for binary CSPs and eSTR1 is the best algorithm for non-binary CSPs. The goal of this table is to demonstrate that our implementation of STR3 and mddc is indeed problematic as those two algorithms perform the worse. *We insist that the value of the STR3 and mddc cannot and should not be judged based on our study* but we need to re-examine our implementation in terms of our results. Such an effort is beyond the time constraint of this thesis.

4.3.4 Qualitative analysis of three representative benchmarks

In this section, we look at the individual results of three representative benchmarks of non-binary CSPs. We do not analyze the results on binary CSPs because all the

Table 4.7: Number of instances completed or not completed (memory or time out) out of 1915 binary instances and 960 non-binary instances

Algorithms	Binary				Nonbinary			
	mem out	time out	mem/time out	Completed	mem out	time out	mem/time out	Completed
GAC2001	349	176	**525**	**1,390**	149	219	368	592
STR1	1,065	140	1,205	710	383	129	512	448
STR2	1,074	148	1,222	693	355	147	502	458
STR3	1,202	47	1,249	666	602	47	649	311
eSTR1	1,153	109	1,262	653	197	85	**282**	**678**
eSTR2	1,149	107	1,256	659	329	46	375	585
STR-Ni	828	15	843	1,072	431	69	500	460
mddc	1,326	0	1,326	589	635	73	708	252
STR-h	1,058	13	1,071	844	454	38	492	468

algorithms perform the same amount of backtracks and node visited (i.e., for binary CSPs, GAC is equivalent to PWC [Bessière *et al.*, 2008]). On the other hand, there is a difference in the performance of the algorithms on the non-binary instances in terms of backtracks, nodes visited, and CPU time.

4.3.4.1 Measured parameters

To compare performance in terms of CPU time, we measure the following parameters:

- For each benchmark category, we report the number of instances existing in the category, with the number completed by all algorithms in parenthesis, and the range of the number of constraints e.

- **Time**: The CPU time in milliseconds. It is seen that some data points are missing because some of the algorithms timed-out (i.e., did not terminate within the time window of 120 minutes). Due to the fact that some instances could not be completed, we conducted a survival data-analysis [Lee, 1992]. The survival

data analysis does not make any assumption about the distribution of the data and yields a calculated mean CPU time for each algorithm. For the algorithms that did not terminate on enough instances within the group, we report a '-' sign instead of the mean value.

- **S**: The equivalence classes of CPU performance. To compute the statistically significant categories, we perform a simple effects comparison between every two algorithms for a significance level of 0.05. This comparison requires a normal distribution of the non-censored data. For this analysis, we assume that all censored data points finished at the maximum cutoff time.

- **#C**: The number of instances completed by a given algorithm.

- **#F**: The number of instances on which the given algorithm is the fastest among all tested ones, where ties are awarded to all parties.

- **#BF**: The number of instances solved by a given algorithm in a backtrack-free manner.

- **#NV**: The average number of nodes visited by the corresponding search. The averages are computed over only the instances completed by all tested algorithms, which is the number in parenthesis in the problem description. Thus, the values reported in **#NV** should be considered in light of the number of completed instances.

In Section 4.3.4.2, we discuss our results on the highly dense dag-rand benchmark where the eSTR* algorithms outperform all other algorithms. In Section 4.3.4.3, we discuss our results on the lexVg crossword-benchmark, where STR2 performs the best. In Section 4.3.4.4, we discuss our results on the Traveling-Salesman benchmark, where

GAC2001 outperforms all other algorithms. Finally in Section 4.3.4.5, we discuss how the remaining benchmarks map into the three benchmarks that we singled out.

4.3.4.2 The dag-rand benchmark

Table 4.8 shows a benchmark where the eSTR* algorithms perform the best. We

Table 4.8: Performance summary for the dag-rand benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
dag-rand: 25(0) instances, $e \in$[16,16]						
GAC2001	2,934,000.00	0	B	**25**	0	
STR1	1,348,394.00	0	B	**25**	0	
STR2	1,331,438.00	0	B	**25**	0	
STR3	-	-	-	0	-	
eSTR1	108,003.20	0	**A**	**25**	**25**	
eSTR2	**101,985.20**	0	**A**	**25**	**25**	
STR-Ni	-	-	-	0	-	
STR-h	-	-	-	0	-	
mddc	-	-	-	0	-	

immediately see the effectiveness of enforcing a higher level of consistency (pair-wise consistency) when the problem is highly dense in that the eSTR* algorithms solve all instances in a backtrack-free manner (#BF) and have competitive CPU time in comparison with (GAC2001, STR1 and STR2).

4.3.4.3 The lexVg benchmark

While Table 4.8, eSTR* largely outperforms all other algorithms in most criteria (Time, #F, S, and #BF), in Table 4.9, STR algorithms outperform the eSTR* algorithms in terms of CPU time (although eSTR* might solve more instances backtrack free or remain in the same significance class as shown in the tables). STR2 solves the most instances fastest.

Table 4.9: Performance summary for the lexVG benchmark: STR2 outperforms all other algorithms in terms of CPU time

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
lexVg: 63(40) instances, $e \in$[8,36]						
GAC2001	38,838.41	**32**	**A**	**63**	26	18.00
STR1	10,414.50	23	**A**	60	26	18.00
STR2	**8,313.83**	**57**	**A**	60	26	18.00
STR3	120,270.34	0	B	58	26	18.00
eSTR1	5,074.00	3	**A**	55	26	18.00
eSTR2	10,202.50	22	**A**	60	26	18.00
STR-Ni	25,370.00	0	B	2	2	18.00
STR-h	136,883.33	0	B	6	2	18.00
mddc	214,408.41	0	B	44	26	18.00

4.3.4.4 The traveling-salesman-20 benchmark

In the traveling salesman benchmark like the one in Table 4.10, GAC2001 outperforms every other algorithm in CPU time (Time), and in the number of completed instances (#C).

Table 4.10: Performance summary for the traveling-salesman-20 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
travellingSalesman-20: 15(0) instances, $e \in$[230,230]						
GAC2001	**67,176.67**	**10**	**A**	**15**	1	
STR1	13,179.17	1	**A**	12	1	
STR2	13,635.83	3	**A**	12	1	
STR3	7,440.00	0	B	1	1	
eSTR1	19,076.67	0	**A**	12	1	
eSTR2	13,820.83	0	**A**	12	1	
STR-Ni	-	-	-	0	-	
STR-h	15,281.67	0	**A**	12	1	
mddc	989,813.33	0	B	3	1	

4.3.4.5 All remaining benchmarks

The 103 instances reported in Tables 4.8, 4.9, and 4.10 above are representative of the results obtained in our experiments on non-binary benchmark problems (which comprise 960 non-binary instances). Below, we classify the remaining tested instances into the three qualitative categories identified above. The fourth category below lists benchmarks that yielded inconclusive results in that no single algorithm outperforms all others. All tables for individual results can be found in Tables B.1 to Tables B.54 in Appendix B.

1. Similar to Table 4.8: aim-50 (Table B.36), aim-100 (Table B.37), aim-200 (Table B.38), modifiedRenault (Table B.54), rand-10-20-10 (Table **??**)

2. Similar to Table 4.9: wordsVg (Table B.49), ogdVg (Table B.47), ukVg (Table B.48), jnhSat (Table B.42), jnhUnsat (Table B.43)

3. Similar to Table 4.10: travelingSalesman-25 (Table B.41), dubois (Table B.39)

4. Inconclusive benchmarks: pret (Table B.50), rand-10-20-10 (Table B.51), rand-8-20-5 (Table B.52), varDimacs (Table B.53).

Summary

In this chapter, we empirically evaluated the performance of the studied algorithms on randomly generated problems and on benchmark problems. We listed the conclusions we drew from our experiments.

Chapter 5

Conclusions and Future Work

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

5.1 Conclusions and Summary of Contributions

Generalized arc-consistency (GAC) is an important consistency property. The GAC2001 algorithm is available in almost all constraint solvers and widely used in practice Bessière *et al.* [2005]. Ullmann [2007] introduced another technique of achieving GAC known as Simple Tabular Reduction (STR). More recently, other algorithms (i.e., STR2, STR3, and STR-Ni) have been proposed to improve the performance of the first STR-based algorithm (STR1). We propose a hybrid, STR-h, that combines two forms of STR (i.e., STR1 and STR-Ni) in order to adapt to the most compact table representation of each constraint. We show that the performance of STR-h is always better than the worse of those two algorithms.

From our experiments, we draw the following conclusions about our algorithms:

1. GAC2001 is particularly effective on binary CSPs but also occasionally on non-

binary CSPs (e.g., the traveling-salesman problem). GAC2001 is the easiest algorithm to implement and its performance in general is quite good and robust. It does not require the use of additional data structures for bookkeeping (other than a pointer for the support of each variable-value pair). Thus, it has no overhead. On large tables, it is outperformed by the STR-based algorithms.

2. STR1 and STR2 : are effective on many binary and non-binary CSPs, especially when the positive tables are smaller than the negative tables (i.e., tight constraints). STR1 and STR2 outperform GAC2001 on most non-binary CSPs. Further, when variable-value pairs have many supports in a given table, STR2 is more effective than STR1 because it stops after finding the first support. Otherwise, their performances are comparable. The implementation of the STR1 and STR2 is slightly more involved because they require a more complex 'handling' of the table constraints. However, the investment is well worth the effort on non-binary CSPs.

3. STR3 requires more memory than all other STR-based algorithms that we tested. Further, our implementation of it fails to show the improvements promised in the original publications. While our implementation is correct in that it does exactly the required filtering, its performance is a significant concern and requires a more careful analysis, an effort for which we lack the time in this thesis.

4. STR-Ni is effective on problems where the negative tables are smaller than the positive tables (i.e., loose constraints).

5. STR-h combines both STR1 and STR-NI, and does not perform worse than the worst of the two of them. It is effective when a CSP has a mixture of tight and

loose constraints. It selectively decides, for each constraint, the most effective constraint representation and filtering algorithm.

6. eSTR* enforces not only GAC, but also pair-wise consistency. Thus, it is effective on non-binary CSPs, especially when the problem is difficult or dense. When the constraints are loose and the problem is easy to solve, preparing the data structures for the eSTR* algorithms may be an overkill.

7. mddc is advantageous for CSPs that the constraints can be represented compactly as an MDD (e.g., pseudo-boolean CSPs). Although we follow the implementation described in the original paper, we fail to replicate the results reported in the literature. Like for STR3, our implementation is correct but its performance is a major concern and prevents us from drawing the correct conclusions.

5.2 Directions for Future Research

Below we identify directions for further research:

1. The performance of our implementation of STR3 and mddc is a major concern. It requires a more careful examination than allowed in the time frame of this thesis, which may yield the derivation of the appropriate advice for future users who may want to include those algorithms in their solvers.

2. Since we started our study, new developments have been reported in the literature, which need to be studied in light of our effort. Those developments target STR algorithms [Xia and Yap, 2013; Gharbi *et al.*, 2014; Jefferson and Nightingale, 2013] as well as MDD-represented constraints [Perez and Régin, 2014].

3. Most importantly, we believe that it is important to extend our idea of a hybrid algorithm that selectively chooses the most appropriate representation for each individual constraint and, given the constraint's representation and size, the most appropriate GAC algorithm. We also believe that the major challenge in the design and implementation of such a hybrid algorithm is the conception of data structures that allow the transparent and efficient interoperability of the various algorithms.

5.3 Final Note

Generalized Arc Consistency is an important property of Constraint Satisfaction Problems. Understanding the different algorithms that are used to enforce GAC and knowing how the algorithms perform for different problems allow us to improve the performance of problem solving.

Appendix A

Algorithms

Algorithm 1: REVISE2001: Enforcing GAC2001

Input: A constraint c from a queue of all constraints, a variable $x \in scp(c)$ and an integer d that represents the current depth of search

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

```
1  DELETE ← false
2  foreach value a ∈ dom(x) do
3      τ ← LastGAC((x, a), c)
4      if ∃k \ τ[x′] ∉ dom(x′) then
5          τ ← succ(τ, rel(c))
6          while (τ ≠ NIL) and (¬c′(τ)) do
7              τ ← succ(τ, rel(c))
8          if tau ≠ NIL then
9              LastGAC((x, a), c) ← τ
10         else
11             delete a from dom(x)
12             DELETE ← true
13 return DELETE
```

Algorithm 2: Enforcing GACstr

Input: A constraint c from a queue of all constraints and an integer d which represents the current depth of search

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

```
1  foreach variable X ∈ future(c) do
2      gacValues(x) ← ∅
3  prev ← −1
4  curr ← first(c);  while curr ≠ −1 do
5      τ ← table(c)[curr]
6      if isValid(c, τ) then
7          foreach variable x ∈ future(c) do
8              if τ(x) ∉ gacValues(x) then
9                  add τ(x) to gacValues(x)
10         prev ← curr
11         curr ← next(c)[curr]
12     else
13         next ← next(c)[curr]
14         removeTuple(c, prev, curr, depth)
15         curr ← next
16 foreach variable x ∈ future(x) do
17     if |gacValues(x)| ≠ |dom(x)| then
18         if gacValues(x) ← ∅ then
19             return false
20         dom(x) ← gacValues(x)
21         add x to propagationQueue
22 return true
```

Algorithm 3: Enforcing GACstr2

Input: A constraint c from a queue of all constraints and an integer d which represents the current depth of search

Output: $true$ if the problem is GACstr2, $false$ otherwise

```
1  S^sup ← ∅
2  if lastAssignedVariable ∉ scp(c) then
3      S^val ← ∅
4  else
5      S^val ← {lastAssignedVariable}
6  foreach variable x ∈ future(c) do
7      gacValues(x) ← ∅
8      S^sup ← S^sup ∪ {x}
9      if getLastRemovedValue(dom(x)) ≠ lastRemoved(c)(x) then
10         S^val ← S^val ∪ {X}
11         lastRemoved(c)(x) ← getLastRemovedValue(dom(x))
12 prev ← −1
13 curr ← first(c);  while curr ≠ −1 do
14     τ ← table(c)[curr]
15     if isValid(c, τ) then
16         foreach variable x ∈ S^sup do
17             if τ(x) ∉ gacValues(x) then
18                 add τ(x) to gacValues(x)
19             if |gacValues(x)| = |dom(x)| then
20                 S^sup ← S^sup \ {x}
21         prev ← curr ; curr ← next(c)[curr]
22     else
23         next ← next(c)[curr]
24         removeTuple(c, prev, curr, depth)
25         curr ← next
26 foreach variable x ∈ S^sup do
27     if gacValues(x) ← ∅ then
28         return false
29     dom(x) ← gacValues(x)
30     lastRemoved(c)(x) ← getLastRemovedValue(dom(x))
31     add x to propagationQueue
32 return true
```

Algorithm 4: Preprocessing with GACinit

Input: A constraint c

```
1  remove invalid tuples from rel(c)
2  invalid(c) ← ∅
3  foreach x ∈ scp(c) and a ∈ dom(x) do
4      row(c, x)(a)[↑] ← |row(c, x)(a)| − 1
5      dep(c)[row(c, x)(a)[0]] ← {(x, a)}
```

Algorithm 5: Enforcing STR3

Input: A constraint c, a variable x, a value a and an integer d which represents the current depth of search

Output: $true$ if the problem is STR3-GAC, $false$ otherwise

```
1  prevMembers ← members(invalid(c))
2  for k ← 0 to row(c, x)(a)[↑] do
3      if row(c, x)(a)[k] ∉ invalid(c) then
4          add row(c, x)(a)[k] to invalid(c)
5  if prevMembers = members(inv(c)) then
6      return true
7  save(c, prevMembers, stateI)
8  foreach i ∈ [prevMembers + 1, ... members(invalid(c))] do
9      k ← dense(inv(c))[i]
10     foreach (y, b) ∈ c.dep[k] such that b ∈ dom(y) do
11         p ← row(c, y)(b)[↑]
12         while p ≥ 0 and row(c, y)(b)[p] ∈ invalid(c) do
13             p ← p − 1
14         if p < 0 then
15             removeValue(y, b)
16             if dom(y) = ∅ then
17                 return false
18         else
19             if p ≠ row(c, y)(b)[↑] then
20                 save((c, y, b), row(c, y)(b)[↑], stateR)
21                 row(c, y)(b)[↑] ← p
22             move (y, b) from dep(c)[k] to dep(c)[row(c, y)(b)[p]]
23 return true
```

Function `save`

Input: key,newData,store

if $(key, oldData) \notin top(store)$ *for any* $oldData$ **then**
  insert $(key, newData)$ *to* $top(store)$

Function `RestoreR`

$list \leftarrow pop(stateR)$
foreach $((c,(x,a)),k) \in list$ **do**
  $row(c,x)(a)[\uparrow] \leftarrow k$

Function `RestoreI`

$list \leftarrow pop(stateR)$
foreach $(c,k) \in list$ **do**
  $members(inv(c)) \leftarrow k$

Algorithm 9: Enforcing eSTR

Input: A constraint c from a queue of all constraints and an integer d which represents the current depth of search

Output: *true* if the problem is eSTR, $false$ otherwise

```
1  foreach variable x ∈ future(c) do
2      gacValues(x) ← ∅
3  prev ← −1
4  curr ← first(c)
5  while curr ≠ −1 do
6      τ ← table(c)[curr]
7      if isValid(c, τ) and isPWconsistent(c, τ) then
8          foreach variable x ∈ future(c) do
9              if τ(x) ∉ gacValues(x) then
10                 add τ(x) to gacValues(x)
11         prev ← curr
12         curr ← next(c)[curr]
13     else
14         next ← next(c)[curr]
15         removeTuple(c, prev, curr, depth)
16         updateCtr(c, curr)
17         curr ← next
18 foreach variable x ∈ future(c) do
19     if |gacValues(x)| ≠ |dom(x)| then
20         if gacValues(x) = ∅ then
21             return false
22         dom(x) ← gacValues(x)
23         add X to propagationQueue
24 return true
```

Algorithm 10: Enforcing STR-N

Input: A constraint c from a queue of all constraints and an integer d which represents the current depth of search

Output: *true* if the problem is STR-N, $false$ otherwise

```
1  if lastIndex(c) = 0 then
2      return false
3  foreach variable x ∈ future(c) do
4      compute count(x, a, c) for x
5  prev ← −1
6  curr ← first(c)
7  while curr ≠ −1 do
8      τ ← table(c)[curr]
9      if isValid(c, τ) then
10         foreach variable x ∈ future(c) do
11             count(x, a, c)getscount(x, a, c) − 1
12         prev ← curr
13         curr ← next(c)[curr]
14     else
15         next ← next(c)[curr]
16         removeTuple(c, prev, curr, depth)
17         curr ← next
18 foreach variable x ∈ future(c) do
19     if count(x, a, c) = 0 then
20         remove (x, a) from dom(x)
21         if |dom(x)| = 0 then
22             return false
23         add x to propagationQueue
24 return true
```

Algorithm 11: Enforcing mddc

Input: An MDD constraint G from a queue of all constraints and an integer d which represents the current depth of search

Output: *true* if the constraint is mdds-gac , $false$ otherwise

```
1  Σ^YES ← ∅
2  restore(Σ^NO, d)
3  for i ← 1 to r do
4      S_i ← dom(x_i)
5  δ ← r + 1
6  mddsSeekSuppports(G, 1)
7  for i ← 1 to δ − 1 do
8      dom(x_i) ← dom(x_i) \ S_i
9  save the state of σ^NO
10 if ∃S_i ∈ S such that S_i ≠ ∅ then
11     return true
12 else
13     return false
14 mddcSeekSupports(G, i)
15 if G = tt then
16     if i < δ then
17         δ ← i
18     return true
19 if G = ff then
20     return false
21 if G ∈ Σ^YES then
22     return true
23 if G ∈ Σ^NO then
24     return false
25 res ← false
26 for k ← 1 to n //n is domain size do
27     if a_k ∈ dom(x_i) then
28         res ← true
29         S_i ← S_i \ a_k
30         if i + 1 = δ and S_i = ∅ then
31             δ ← i
32             Break
33 Σ^res ← Σ^res ∪ [G]
34 return true
```

Appendix B

Results of Experiments on Benchmark Problems

Below are the tables with the detailed experimental results on benchmark problems omitted from Chapter 4 in order to improve readobility. We report the results first for binary then for non-binary benchmark problems.

The summary analysis can be found in Section 4.3.3 and three representative benchmarks are discussed in Section 4.3.4.

B.1 Comparison criteria

Below we list the comparison criteria used in all the tables in this appendix.

- A '-' signifies that all algorithms did not complete on the instances of the category.
- For each benchmark category, we report the number of instances existing in the category, with the number completed by all algorithms in parenthesis, and the range of the number of constraints e.

- **Time**: The CPU time in milliseconds. It is seen that some data points are missing because some of the algorithms timed-out (could not finish within the given time-window of 120 minutes). Due to the fact that some instances could not be completed, we conducted a survival data-analysis [Lee, 1992]. The survival data analysis does not make any assumption about the distribution of the data and yields a calculated mean CPU time for each algorithm. There is no mean (but rather a '-') calculated for algorithms that did not terminate on enough instances within the group.

- **S**: The equivalence classes of CPU performance. To compute the statistically significant categories, we perform a simple effects comparison between every two algorithms for a significance level of 0.05. This comparison requires a normal distribution of the non-censored data. For this analysis, we assume that all censored data points finished at the maximum cutoff time.

- **#C**: The number of instances completed by a given algorithm.

- **#F**: The number of instances on which the given algorithm is the fastest among all tested ones, where ties are awarded to all parties.

- **#BF**: The number of instances solved by a given algorithm in a backtrack-free manner.

- **#NV**: The average number of nodes visited by the corresponding search. The averages are computed over only the instances completed by all tested algorithms, which is the number in parenthesis in the problem description. Thus, the values reported in **#NV** should be considered in light of the number of completed instances.

B.2 Binary benchmark problems

Below we summarize the content of the tables in this section:

- GAC performs well: composed-25-1-80 (Table B.1), frb35-17 (Table B.8), frb40-19 (Table B.10), frb45-21 (Table B.9), geom (Table B.11), marc (Table B.13), QCP-15 (Table B.14), rand-2-23 (Table B.15), rand-2-24 (Table B.16), rand-2-25 (Table B.17), rand-2-26 (Table B.18), rand-2-30-15-fcd (Table B.20), rand-2-30-15 (Table B.21), rand-2-40-19-fcd (Table B.22), rand-2-40-19 (Table B.23), tightness0.5 (Table B.27), tightness0.65 (Table B.28), tightness0.8 (Table B.29), tightness0.9 (Table B.30).

- STR-N (where the negative tables are largely smaller) performs best: driver (Table B.7), frb35-17 (Table B.8), langford (Table B.12), tightness0.1 (Table B.24), tightness0.2 (Table B.25), tightness0.35 (Table B.26), composed-25-10-20 (Table B.2), QWH-15 (Table B.35)

- STR2 performs well: marc (Table B.13), hanoi (Table B.33).

- Results are inconclusive on: coloring (Table B.31), frb30-15 (Table B.32), QWH-10 (Table B.34), composed-75-1-80 (Table B.3).

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

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
composed-25-1-80: 10(1) instances, $e \in$[302,302]						
GAC2001	438,692.50	**4**	**A**	**4**	0	1.00
STR1	10.00	2	**A**	2	0	1.00
STR2	15.00	1	**A**	2	0	1.00
STR3	40.00	0	**A**	2	0	1.00
eSTR1	325.00	0	**A**	2	0	1.00
eSTR2	340.00	0	**A**	1	0	1.00
STR-Ni	10.00	2	**A**	2	0	1.00
STR-h	15.00	1	**A**	2	0	1.00
mddc	40.00	0	**A**	2	0	1.00

Table B.2: Statistical analysis of the composed-25-10-20 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
composed-25-10-20: 10(3) instances, $e \in$[620,620]						
GAC2001	26,323.33	1	**A**	**6**	0	123.60
STR1	40.00	0	**A**	5	0	123.60
STR2	28.00	3	**A**	5	0	123.60
STR3	168.00	0	**A**	5	0	123.60
eSTR1	524.00	0	**A**	5	0	123.60
eSTR2	560.00	0	**A**	5	0	123.60
STR-Ni	20,265.00	**5**	**A**	**6**	0	123.60
STR-h	30.00	1	**A**	5	0	123.60
mddc	90.00	0	**A**	5	0	123.60

Table B.3: Statistical analysis of the composed-75-1-80 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
composed-75-1-80: 10(3) instances, $e \in$[702,702]						
GAC2001	**13.33**	**3**	**A**	3	0	1.00
STR1	26.67	0	**A**	3	0	1.00
STR2	16.67	2	**A**	3	0	1.00
STR3	103.33	0	**A**	3	0	1.00
eSTR1	666.67	0	**A**	3	0	1.00
eSTR2	643.33	0	**A**	3	0	1.00
STR-Ni	**13.33**	**3**	**A**	3	0	1.00
STR-h	26.67	1	**A**	3	0	1.00
mddc	83.33	0	**A**	3	0	1.00

Table B.4: Statistical analysis of the ehi-85 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
ehi-85: 100(36) instances, $e \in$[4081,4137]						
GAC2001	995,500.82	**51**	**A**	**97**	0	845.42
STR1	1,752.50	0	B	36	0	845.42
STR2	1,852.50	0	B	36	0	845.42
STR3	7,743.33	0	C	36	0	845.42
eSTR1	4,444.72	0	C	36	0	845.42
eSTR2	6,166.67	0	C	36	0	845.42
STR-Ni	29,024.13	15	B	46	0	845.42
STR-h	1,180.83	34	B	36	0	845.42
mddc	1,611.94	0	B	36	0	845.42

Table B.5: Statistical analysis of the ehi-90 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
ehi-90: 100(26) instances, $e \in$[4343,4400]						
GAC2001	1,579,893.02	**55**	**A**	**96**	0	1,342.96
STR1	1,049.52	2	B	42	0	1,342.96
STR2	1,323.81	0	B	42	0	1,342.96
STR3	12,189.23	0	C	26	0	1,342.96
eSTR1	4,620.71	0	C	42	0	1,342.96
eSTR2	4,699.05	0	C	42	0	1,342.96
STR-Ni	6,399.77	6	B	43	0	1,342.96
STR-h	917.14	41	B	42	0	1,342.96
mddc	982.38	0	B	42	0	1,342.96

Table B.6: Statistical analysis of the QCP-10 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
QCP-10: 15(13) instances, $e \in$[822,822]						
GAC2001	11,784.67	3	**A**	**15**	4	673.15
STR1	138.46	3	**A**	13	4	673.15
STR2	145.38	3	**A**	13	4	673.15
STR3	29,960.00	0	B	14	4	673.15
eSTR1	636.92	0	B	13	4	673.15
eSTR2	615.38	0	B	13	4	673.15
STR-Ni	5,349.33	**14**	**A**	**15**	4	673.15
STR-h	150.77	2	**A**	13	4	673.15
mddc	170.00	0	**A**	13	4	673.15

Table B.7: Statistical analysis of the driver benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
driver: 7(4) instances, $e \in$[217,17447]						
GAC2001	80,294.29	1	B	**7**	1	11,667.50
STR1	3,555.00	2	B	4	1	11,667.50
STR2	7,447.50	1	B	4	1	11,667.50
STR3	94,082.50	1	B	4	1	11,667.50
eSTR1	12,852.50	0	B	4	1	11,667.50
eSTR2	13,607.50	0	B	4	1	11,667.50
STR-Ni	50,475.71	**6**	**A**	**7**	1	11,667.50
STR-h	4,275.00	1	B	4	1	11,667.50
mddc	4,852.50	1	B	4	1	11,667.50

Table B.8: Statistical analysis of the frb35-17 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
frb35-17: 10(4) instances, $e \in$[260,273]						
GAC2001	27,548.00	**5**	**A**	**10**	0	12,556.00
STR1	25,673.33	0	B	6	0	12,556.00
STR2	28,618.33	0	B	6	0	12,556.00
STR3	51,078.75	0	B	8	0	12,556.00
eSTR1	27,045.00	0	B	4	0	12,556.00
eSTR2	42,408.33	0	B	6	0	12,556.00
STR-Ni *	26,227.00	**5**	**A**	**10**	0	17,405.75
STR-h *	24,106.67	0	B	6	0	17,405.75
mddc	17,050.00	0	B	4	0	12,556.00

* STR-Ni and STR-h visit a different number of nodes because of the way the constraints are provided in the xml file (i.e., merged versus un-merged)

Table B.9: Statistical analysis of the frb45-21 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
frb45-21: 10(0) instances, $e \in$[369,394]						
GAC2001	3,438,888.89	**9**	**A**	**9**	0	
STR1	-	-		0	-	
STR2	-	-		0	-	
STR3	-	-		0	-	
eSTR1	-	-		0	-	
eSTR2	-	-		0	-	
STR-Ni	-	-		0	-	
STR-h	-	-		0	-	
mddc	-	-		0	-	

Table B.10: Statistical analysis of the frb40-19 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
frb40-19: 10(0) instances, $e \in$[308,326]						
GAC2001	155,414.00	**5**	**A**	**10**	0	
STR1	-	-		0	-	
STR2	-	-		0	-	
STR3	-	-		0	-	
eSTR1	-	-		0	-	
eSTR2	-	-		0	-	
STR-Ni	72,068.00	**5**	**A**	5	0	
STR-h	-	-		0	-	
mddc	-	-		0	-	

Table B.11: Statistical analysis of the geom benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
geom: 100(85) instances, $e \in$[339,555]						
GAC2001	50,173.10	**75**	**A**	**100**	19	1,024.31
STR1	3,941.72	1	B	87	19	1,024.31
STR2	5,107.01	1	B	87	19	1,024.31
STR3	7,774.25	0	C	87	19	1,024.31
eSTR1	7,342.02	0	C	84	19	1,024.31
eSTR2	8,468.16	0	C	87	19	1,024.31
STR-Ni	6,260.77	52	A	91	19	1,024.31
STR-h	4,919.89	0	B	87	19	1,024.31
mddc	3,126.35	0	B	85	19	1,024.31

Table B.12: Statistical analysis of the langford benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
langford: 4(2) instances, $e \in$[28,528]						
GAC2001	13,377.50	1	**A**	**4**	1	223.00
STR1	57,052.50	1	**A**	**4**	1	223.00
STR2	530.00	1	**A**	2	1	223.00
STR3	52,027.50	1	**A**	**4**	1	223.00
eSTR1	2,340.00	1	**A**	2	1	223.00
eSTR2	2,505.00	0	**A**	2	1	223.00
STR-Ni	8,507.50	**4**	**A**	**4**	1	223.00
STR-h	48,227.50	1	**A**	**4**	1	223.00
mddc	445.00	1	**A**	2	1	223.00

Table B.13: Statistical analysis of the marc benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
marc: 10(0) instances, $e \in$[3160,4560]						
GAC2001	3,696.00	**5**	**A**	**10**	5	
STR1	3,747.00	0	**A**	**10**	5	
STR2	2,948.00	**5**	**A**	**10**	5	
STR3	-	0	-	0	-	
eSTR1	-	0	-	0	-	
eSTR2	960,505.00	0	B	2	1	
STR-Ni	5,401.00	0	**A**	**10**	5	
STR-h	3,615.00	0	**A**	**10**	5	
mddc	11,757.50	0	B	4	4	

Table B.14: Statistical analysis of the QCP-15 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
QCP-15: 15(3) instances, $e \in$[2519,2520]						
GAC2001	1,387,868.89	**5**	**A**	**9**	0	9,978.00
STR1	6,406.67	0	**A**	3	0	9,978.00
STR2	10,770.00	0	**A**	3	0	9,978.00
STR3	329,485.00	0	B	4	0	9,978.00
eSTR1	12,606.67	0	**A**	3	0	9,978.00
eSTR2	16,433.33	0	**A**	3	0	9,978.00
STR-Ni	10,152.50	4	**A**	4	0	9,978.00
STR-h	6,113.33	0	**A**	3	0	9,978.00
mddc	5,680.00	0	**A**	3	0	9,978.00

Table B.15: Statistical analysis of the rand-2-23 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-2-23: 10(0) instances, $e \in$[253,253]						
GAC2001	238,473.00	**6**	**A**	**10**	0	
STR1	38,120.00	0	B	1	0	
STR2	56,570.00	0	B	1	0	
STR3	73,300.00	0	B	1	0	
eSTR1		0	-	0	-	
eSTR2	129,470.00	0	B	1	0	
STR-Ni	110,460.00	4	B	4	0	
STR-h	50,050.00	0	B	1	0	
mddc	33,910.00	0	B	1	0	

Table B.16: Statistical analysis of the rand-2-24 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-2-24: 10(0) instances, $e \in$[276,276]						
GAC2001	646,879.00	**7**	**A**	**10**	0	
STR1		-	-	0	-	
STR2		-	-	0	-	
STR3		-	-	0	-	
eSTR1		-	-	0	-	
eSTR2		-	-	0	-	
STR-Ni	96,820.00	3	**A**	3	0	
STR-h		-	-	0	-	
mddc		-	-	0	-	

Table B.17: Statistical analysis of the rand-2-25 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-2-25: 10(0) instances, $e \in$[]						
GAC2001	2,058,016.00	**9**	**A**	**10**	0	
STR1		-	-	0	-	
STR2		-	-	0	-	
STR3		-	-	0	-	
eSTR1		-	-	0	-	
eSTR2		-	-	0	-	
STR-Ni	110,300.00	1	B	1	0	
STR-h		-	-	0	-	
mddc		-	-	0	-	

Table B.18: Statistical analysis of the rand-2-26 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-2-26: 10(0) instances, $e \in$[]						
GAC2001	4,063,000.00	**10**	**A**	**10**	0	
STR1		-	-	0	-	
STR2		-	-	0	-	
STR3		-	-	0	-	
eSTR1		-	-	0	-	
eSTR2		-	-	0	-	
STR-Ni	-	-	-	0	-	
STR-h		-	-	0	-	
mddc		-	-	0	-	

Table B.19: Statistical analysis of the rand-2-27 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-2-27: 10(0) instances, $e \in$[]						
GAC2001	4,780,000.00	**2**	**A**	**2**	0	
STR1		-	-	0	-	
STR2		-	-	0	-	
STR3		-	-	0	-	
eSTR1		-	-	0	-	
eSTR2		-	-	0	-	
STR-Ni	-	-	-	0	-	
STR-h		-	-	0	-	
mddc		-	-	0	-	

Table B.20: Statistical analysis of the rand-2-30-15-fcd benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-2-30-15-fcd: 50(50) instances, $e \in$[208,230]						
GAC2001	**2,391.80**	**49**	**A**	**50**	0	3,974.20
STR1	4,186.20	0	B	**50**	0	3,974.20
STR2	4,283.80	0	B	**50**	0	3,974.20
STR3	6,286.40	0	C	**50**	0	3,974.20
eSTR1	7,053.60	0	C	**50**	0	3,974.20
eSTR2	6,441.40	0	C	**50**	0	3,974.20
STR-Ni	2,591.20	4	**A**	**50**	0	3,974.20
STR-h	4,234.40	0	B	**50**	0	3,974.20
mddc	5,265.60	0	B	**50**	0	3,974.20

Table B.21: Statistical analysis of the rand-2-30-15 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-2-30-15: 50(49) instances, $e \in$[208,230]						
GAC2001	4,386.20	**40**	**A**	**50**	0	6,936.88
STR1	8,243.20	0	B	**50**	0	6,936.88
STR2	10,079.20	0	B	**50**	0	6,936.88
STR3	11,742.80	0	C	**50**	0	6,936.88
eSTR1	11,897.96	0	C	49	0	6,936.88
eSTR2	13,455.80	0	C	**50**	0	6,936.88
STR-Ni	**4,385.60**	12	**A**	**50**	0	6,936.88
STR-h	7,378.80	0	B	**50**	0	6,936.88
mddc	7,126.94	0	B	49	0	6,936.88

Table B.22: Statistical analysis of the rand-2-40-19-fcd benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-2-40-19-fcd: 50(3) instances, $e \in$[325,351]						
GAC2001	471,552.00	**45**	**A**	**50**	0	5,489.00
STR1	28,492.50	0	B	4	0	5,489.00
STR2	18,147.50	0	B	4	0	5,489.00
STR3	68,065.00	0	B	6	0	5,489.00
eSTR1	37,426.67	0	B	3	0	5,489.00
eSTR2	61,137.50	0	B	4	0	5,489.00
STR-Ni	137,128.00	5	B	5	0	5,489.00
STR-h	18,205.00	0	B	4	0	5,489.00
mddc	34,747.50	0	B	4	0	5,489.00

Table B.23: Statistical analysis of the rand-2-40-19 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-2-40-19: 50(1) instances, $e \in$[325,351]						
GAC2001	1,039,856.80	**50**	**A**	**50**	0	
STR1	33,650.00	0	B	1	0	
STR2	25,950.00	0	B	1	0	
STR3	43,890.00	0	B	1	0	
eSTR1	66,070.00	0	B	1	0	
eSTR2	42,590.00	0	B	1	0	
STR-Ni	133,465.00	0	B	2	0	
STR-h	44,250.00	0	B	1	0	
mddc	43,500.00	0	B	1	0	

Table B.24: Statistical analysis of the tightness0.1 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
tightness0.1: 100(19) instances, $e \in$[746,753]						
GAC2001	65,837.60	0	**A**	**100**	0	12,021.95
STR1	25,465.17	0	B	29	0	12,021.95
STR2	29,772.22	0	B	27	0	12,021.95
STR3	51,316.75	0	B	40	0	12,021.95
eSTR1	39,576.32	0	B	19	0	12,021.95
eSTR2	41,358.57	0	B	28	0	12,021.95
STR-Ni	31,794.70	**100**	**A**	**100**	0	12,021.95
STR-h	30,530.00	0	B	28	0	12,021.95
mddc	19,495.91	0	B	22	0	12,021.95

Table B.25: Statistical analysis of the tightness0.2 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
tightness0.2: 100(21) instances, $e \in$[414,414]						
GAC2001	62,246.20	0	**A**	**100**	0	15,680.38
STR1	30,685.00	0	B	30	0	15,680.38
STR2	27,097.78	0	B	27	0	15,680.38
STR3	51,269.77	0	B	44	0	15,680.38
eSTR1	36,856.67	0	B	21	0	15,680.38
eSTR2	40,408.28	0	B	29	0	15,680.38
STR-Ni	36,772.30	**100**	**A**	**1**00	0	15,680.38
STR-h	27,514.48	0	B	29	0	15,680.38
mddc	23,865.00	0	B	22	0	15,680.38

Table B.26: Statistical analysis of the tightness0.35 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
tightness0.35: 100(26) instances, $e \in$[250,250]						
GAC2001	65,352.20	1	**A**	**100**	0	12,173.08
STR1	29,401.91	0	B	47	0	12,173.08
STR2	20,890.77	0	B	39	0	12,173.08
STR3	49,357.12	0	B	52	0	12,173.08
eSTR1	24,382.31	0	B	26	0	12,173.08
eSTR2	30,829.23	0	B	39	0	12,173.08
STR-Ni	43,469.29	**99**	**A**	99	0	12,173.08
STR-h	24,720.26	0	B	39	0	12,173.08
mddc	21,298.62	0	B	29	0	12,173.08

Table B.27: Statistical analysis of the tightness0.5 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
tightness0.5: 100(17) instances, $e \in$[180,180]						
GAC2001	100,192.90	**100**	**A**	**100**	0	9,551.50
STR1	38,664.17	0	C	36	0	9,551.50
STR2	27,422.73	0	C	33	0	9,551.50
STR3	52,675.24	0	C	42	0	9,551.50
eSTR1	25,547.65	0	C	17	0	9,551.50
eSTR2	37,678.48	0	C	33	0	9,551.50
STR-Ni	101,740.10	0	B	96	0	9,551.50
STR-h	27,910.59	0	B	34	0	9,551.50
mddc	24,290.00	0	C	23	0	9,551.50

Table B.28: Statistical analysis of the tightness0.65 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
tightness0.65: 100(23) instances, $e \in$[40,40]						
GAC2001	74,495.60	**100**	**A**	**100**	0	7,268.96
STR1	30,112.92	0	B	48	0	7,268.96
STR2	27,163.83	0	B	47	0	7,268.96
STR3	58,473.96	0	C	48	0	7,268.96
eSTR1	23,017.83	0	C	23	0	7,268.96
eSTR2	43,147.71	0	C	48	0	7,268.96
STR-Ni	207,437.50	0	D	96	0	7,268.96
STR-h	35,699.79	0	B	48	0	7,268.96
mddc	25,177.65	0	C	34	0	7,268.96

Table B.29: Statistical analysis of the tightness0.8 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
tightness0.8: 100(36) instances, $e \in$[103,103]						
GAC2001	174,146.30	**100**	**A**	**100**	0	2,992.56
STR1	33,153.70	0	B	54	0	2,992.56
STR2	29,848.00	0	B	55	0	2,992.56
STR3	54,783.00	0	B	50	0	2,992.56
eSTR1	21,356.94	0	B	36	0	2,992.56
eSTR2	35,602.73	0	B	55	0	2,992.56
STR-Ni	646,332.58	0	D	89	0	2,992.56
STR-h	59,335.79	0	B	57	0	2,992.56
mddc	33,252.65	0	C	49	0	2,992.56

Table B.30: Statistical analysis of the tightness0.9 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
tightness0.9: 100(36) instances, $e \in$[84,84]						
GAC2001	307,169.49	**68**	**A**	**99**	0	751.71
STR1	61,666.00	1	C	65	0	751.71
STR2	49,936.42	30	B	67	0	751.71
STR3	26,658.00	0	D	40	0	751.71
eSTR1	19,120.28	0	D	36	0	751.71
eSTR2	70,502.99	0	C	67	0	751.71
STR-Ni	1,655,783.53	0	D	85	0	751.71
STR-h	177,208.59	0	D	64	0	751.71
mddc	58,870.18	0	D	56	0	751.71

Table B.31: Statistical analysis of the coloring benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
coloring: 22(18) instances, $e \in$[78,5714]						
GAC2001	17,869.09	14	**A**	**22**	**4**	281.67
STR1	107.00	12	**A**	20	**4**	281.67
STR2	105.56	11	**A**	18	3	281.67
STR3	184.44	9	**A**	18	**4**	271.72
eSTR1	336.50	8	B	20	**4**	281.67
eSTR2	415.50	0	B	20	**4**	281.67
STR-Ni	8,920.91	**19**	**A**	**22**	**4**	281.67
STR-h	84.21	14	**A**	19	3	281.67
mddc	115.50	13	**A**	20	**4**	281.67

Table B.32: Statistical analysis of the frb30-15 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
frb30-15: 10(10) instances, $e \in$[208,217]						
GAC2001	**1,577.00**	**10**	**A**	**10**	0	2,717.80
STR1	3,050.00	0	**A**	**10**	0	2,717.80
STR2	2,839.00	0	**A**	**10**	0	2,717.80
STR3	4,675.00	0	**A**	**10**	0	2,717.80
eSTR1	5,020.00	0	**A**	**10**	0	2,717.80
eSTR2	6,218.00	0	B	**10**	0	2,717.80
STR-Ni	2,566.00	0	**A**	**10**	0	4,348.20
STR-h	5,002.00	0	**A**	**10**	0	4,348.20
mddc	3,182.00	0	**A**	**10**	0	2,717.80

Table B.33: Statistical analysis of the hanoi benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
hanoi: 5(3) instances, $e \in$[5,125]						
GAC2001	792.00	2	**A**	**5**	**5**	16.67
STR1	582.00	4	**A**	**5**	**5**	16.67
STR2	544.00	**5**	**A**	**5**	**5**	16.67
STR3	176.67	1	**A**	3	3	16.67
eSTR1	828.00	2	**A**	**5**	**5**	16.67
eSTR2	774.00	0	**A**	**5**	**5**	16.67
STR-Ni	12,890.00	1	**A**	4	4	16.67
STR-h	1,022.00	2	**A**	**5**	**5**	16.67
mddc	356.67	1	**A**	3	3	16.67

Table B.34: Statistical analysis of the QWH-10 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
QWH-10: 10(10) instances, $e \in$[756,756]						
GAC2001	19.00	7	**A**	**10**	**3**	146.30
STR1	27.00	2	**A**	**10**	**3**	146.30
STR2	27.00	4	**A**	**10**	**3**	146.30
STR3	156.00	0	B	**10**	**3**	146.30
eSTR1	358.00	0	B	**10**	**3**	146.30
eSTR2	378.00	0	B	**10**	**3**	146.30
STR-Ni	**17.00**	**9**	**A**	**10**	**3**	146.30
STR-h	32.00	2	**A**	**10**	**3**	146.30
mddc	51.00	0	**A**	**10**	**3**	146.30

Table B.35: Statistical analysis of the QWH-15 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
QWH-15: 10(7) instances, $e \in$[2324,2324]						
GAC2001	54,952.00	0	**A**	**10**	0	25,435.57
STR1	18,908.75	0	B	8	0	25,435.57
STR2	22,384.29	0	B	7	0	25,435.57
STR3	256,483.75	0	B	8	0	25,435.57
eSTR1	30,757.14	0	B	7	0	25,435.57
eSTR2	34,522.86	0	B	7	0	25,435.57
STR-Ni	26,533.00	**10**	**A**	**10**	0	25,435.57
STR-h	17,291.43	0	B	7	0	25,435.57
mddc	12,191.43	0	B	7	0	25,435.57

B.3 Non-binary benchmark problems

Below are the tables for the tested non-binary CSPs that were omitted from Section 4.3.4:

- eSTR* where a higher level of consistency is enforced performs best: aim-50 (Table B.36), aim-100 (Table B.37), aim-200 (Table B.38), jnhSat (Table B.42), jnhUnsat (Table B.43), ssa (Table B.40), rand-3-20-20-fcd (Table B.44), rand-3-20-20 (Table B.45), rand-3-24-24-fcd (Table B.46), rand-8-20-5 (Table B.52)rand-10-20-10 (Table B.51), modifiedRenault (Table B.54).
- GAC performs well: dubios (Table B.39), pret (Table B.50), travellingSalesman-20 (Table 4.10), travellingSalesman-25 (Table B.41).
- STR2 performs well: ogdVg (Table B.47), wordsVg (Table B.49).
- STR-N performs well: ukVg (Table B.48).
- Results are inconclusive on: varDimacs (Table B.53).

Table B.36: Statistical analysis of the aim-50 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
aim-50: 24(24) instances, $e \in$[69,289]						
GAC2001	512.92	3	B	**24**	1	42,937.92
STR1	705.00	4	B	**24**	1	42,937.92
STR2	786.25	5	B	**24**	1	42,937.92
STR3	795.83	3	B	**24**	1	42,937.92
eSTR1	**10.00**	**17**	**A**	**24**	**19**	25.00
eSTR2	10.42	0	**A**	**24**	**19**	25.00
STR-Ni	379.17	8	**A**	**24**	2	42,937.92
STR-h	709.17	3	B	**24**	2	42,937.92
mddc	727.08	3	B	**24**	1	42,937.92

Table B.37: Statistical analysis of the aim-100 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
aim-100: 24(8) instances, $e \in$[150,570]						
GAC2001	229,864.67	0	B	15	1	2,268.13
STR1	197.50	0	B	8	1	2,268.13
STR2	195.00	1	B	8	1	2,268.13
STR3	12,468.89	0	B	9	1	2,268.13
eSTR1	28.33	**11**	**A**	**24**	**16**	50.00
eSTR2	28.75	0	**A**	**24**	14	50.00
STR-Ni	7,853.00	4	B	10	1	2,268.13
STR-h	177.50	3	B	8	1	2,268.13
mddc	190.00	0	B	8	1	2,268.13

Table B.38: Statistical analysis of the aim-200 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
aim-200: 24(0) instances, $e \in$[302,1169]						
GAC2001	433,005.00	0	B	8	0	
STR1	17,220.00	0	B	4	0	
STR2	17,320.00	0	B	4	0	
STR3	-	-	-	0	-	
eSTR1	68.26	**8**	**A**	23	**23**	
eSTR2	68.33	0	**A**	**24**	19	
STR-Ni	14,150.00	0	B	5	0	
STR-h	13,760.00	0	B	4	0	
mddc	16,602.50	0	B	4	0	

Table B.39: Statistical analysis of the dubois benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
dubois: 13(0) instances, $e \in$[40,200]						
GAC2001	107,920.00	**2**	**A**	**2**	0	
STR1	-	-	-	0	-	
STR2	-	-	-	0	-	
STR3	-	-	-	0	-	
eSTR1	-	-	-	0	-	
eSTR2	-	-	-	0	-	
STR-Ni	90,530.00	0	**A**	1	0	
STR-h	-	-	-	0	-	
mddc	-	-	-	0	-	

Table B.40: Statistical analysis of the ssa benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
ssa: 8(4) instances, $e \in$[177,22141]						
GAC2001	745.00	2	**A**	6	4	62,507.50
STR1	1,023.33	1	**A**	6	4	62,507.50
STR2	1,897.50	2	**A**	4	2	62,507.50
STR3	39,115.00	0	**A**	4	2	62,507.50
eSTR1	430.00	2	**A**	**8**	**6**	1,246.00
eSTR2	278.75	0	**A**	**8**	**6**	1,246.00
STR-Ni	616.67	2	**A**	6	2	62,507.50
STR-h	1,257.50	0	**A**	4	0	62,507.50
mddc	873.33	**4**	**A**	6	4	62,507.50

Table B.41: Statistical analysis of the travellingSalesman-25 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
travellingSalesman-25: 15(0) instances, $e \in$[350,350]						
GAC2001	955,896.00	**15**	**A**	**15**	0	
STR1	24,311.67	0	B	6	0	
STR2	18,625.00	0	B	6	0	
STR3	-	-	-	0	-	
eSTR1	38,461.67	0	B	6	0	
eSTR2	33,378.33	0	B	6	0	
STR-Ni	-	-	-	0	-	
STR-h	27,296.67	0	B	6	0	
mddc	-	-	-	0	-	

Table B.42: Statistical analysis of the jnhSat benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
jnhSat: 16(4) instances, $e \in$[726,819]						
GAC2001	5,765.63	1	**A**	**16**	1	1,524.21
STR1	4,430.67	0	B	15	1	1,524.21
STR2	6,629.38	1	**A**	**16**	1	1,524.21
STR3	86,557.50	0	B	**16**	1	1,524.21
eSTR1	17,257.50	1	B	**16**	**16**	100.00
eSTR2	17,729.38	0	B	**16**	**16**	100.00
STR-Ni	1,713.75	**14**	**A**	**16**	0	1,524.21
STR-h	1,938.00	0	**A**	15	0	1,524.21
mddc	4,776.43	0	B	14	1	1,524.21

Table B.43: Statistical analysis of the jnhUnsat benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
jnhUnsat: 34(14) instances, $e \in$[714,834]						
GAC2001	5,765.63	1	B	**16**	1	1946.26
STR1	4,430.67	0	B	15	1	1,946.26
STR2	6,629.38	1	B	**16**	1	1,946.26
STR3	86,557.50	0	C	**16**	1	1,946.26
eSTR1	17,257.50	1	C	**16**	**16**	0.00
eSTR2	17,729.38	0	C	**16**	**16**	0.00
STR-Ni	1,713.75	**14**	**A**	**16**	0	1,946.26
STR-h	1,938.00	0	B	15	0	1946.26
mddc	4,776.43	0	C	14	1	1,946.26

Table B.44: Statistical analysis of the rand-3-20-20-fcd benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-3-20-20-fcd: 50(12) instances, $e \in$[55,60]						
GAC2001	235,679.80	1	**A**	**50**	0	10,462.73
STR1	78,321.38	0	B	29	0	10,462.73
STR2	79,446.90	0	B	29	0	10,462.73
STR3	336,209.55	0	C	22	0	10,462.73
eSTR1	1,187.76	**28**	**A**	49	**48**	7,144.27
eSTR2	39,284.39	0	**A**	41	17	7,144.27
STR-Ni	444,185.63	0	C	48	0	10,462.73
STR-h	157,619.31	0	C	29	0	10,462.73
mddc	65,903.33	0	C	12	0	10,462.73

Table B.45: Statistical analysis of the rand-3-20-20 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-3-20-20: 50(5) instances, $e \in$[55,60]						
GAC2001	405,241.00	3	**A**	**50**	0	16,641.50
STR1	113,266.19	0	B	21	0	16,641.50
STR2	100,038.50	0	B	20	0	16,641.50
STR3	485,491.67	0	B	12	0	16,641.50
eSTR1	1,157.66	**32**	**A**	47	**47**	6,547.75
eSTR2	23,687.07	0	**A**	41	17	6,547.75
STR-Ni	741,572.92	0	B	48	0	16,641.50
STR-h	219,327.14	0	B	21	0	16,641.50
mddc	63,034.00	0	B	5	0	16,641.50

Table B.46: Statistical analysis of the rand-3-24-24-fcd benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-3-24-24-fcd: 50(0) instances, $e \in$[72,76]						
GAC2001	2,196,606.92	4	B	26	0	
STR1	115,700.00	0	C	2	0	
STR2	-	-	-	0	-	
STR3	676,630.00	0	C	2	0	
eSTR1	2,745.85	**16**	**A**	**41**	**41**	
eSTR2	5,378.00	0	**A**	15	9	
STR-Ni	3,255,730.00	0	C	3	0	
STR-h	127,585.00	0	C	2	0	
mddc	156,840.00	0	C	1	0	

Table B.47: Statistical analysis of the ogdVg benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
ogdVg: 65(2) instances, $e \in$[8,36]						
GAC2001	592,518.60	14	**A**	**43**	**11**	18.00
STR1	206,458.25	0	**A**	40	**11**	18.00
STR2	129,209.50	**32**	**A**	40	**11**	18.00
STR3	266,023.60	0	B	25	11	18.00
eSTR1	15,112.81	0	**A**	32	**11**	18.00
eSTR2	135,021.75	0	**A**	40	**11**	18.00
STR-Ni	14,765.00	0	B	2	2	18.00
STR-h	30,513.00	0	B	10	7	18.00
mddc	1,653,245.00	0	B	12	9	18.00

Table B.48: Statistical analysis of the ukVg benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
ukVg: 65(7) instances, $e \in$[8,36]						
GAC2001	54,952.00	0	B	**10**	0	25,435.57
STR1	18,908.75	0	**A**	8	0	25,435.57
STR2	22,384.29	0	B	7	0	25,435.57
STR3	256,483.75	0	C	8	0	25,435.57
eSTR1	30,757.14	0	**A**	7	0	25,435.57
eSTR2	34,522.86	0	B	7	0	25,435.57
STR-Ni	26,533.00	**10**	**A**	**10**	0	25,435.57
STR-h	17,291.43	0	**A**	7	0	25,435.57
mddc	12,191.43	0	**A**	7	0	25,435.57

Table B.49: Statistical analysis of the wordsVg benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
wordsVg: 65(0) instances, $e \in$[8,36]						
GAC2001	665,027.86	9	**A**	**42**	**3**	
STR1	305,310.00	0	C	1	0	
STR2	44,399.43	**34**	**A**	35	**3**	
STR3	-	-	-	0	-	
eSTR1	18,098.15	0	**A**	27	**3**	
eSTR2	63,174.29	0	**A**	35	**3**	
STR-Ni	17,415.00	0	B	2	2	
STR-h	49,941.25	0	B	8	**3**	
mddc	-	-	-	0	-	

Table B.50: Statistical analysis of the pret benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
pret: 8(0) instances, $e \in$[40,100]						
GAC2001	39,442.50	**4**	**A**	**4**	0	
STR1	-	-	-	0	-	
STR2	-	-	-	0	-	
STR3	71,945.00	0	**A**	**4**	0	
eSTR1	-	-	-	0	-	
eSTR2	-	-	-	0	-	
STR-Ni	41,557.50	0	**A**	**4**	0	
STR-h	50,350.00	0	**A**	**4**	0	
mddc	-	-	-	0	-	

Table B.51: Statistical analysis of the rand-10-20-10 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-10-20-10: 20(0) instances, $e \in$[5,5]						
GAC2001	3,595.00	0	B	**20**	0	
STR1	318.00	6	**A**	**20**	0	
STR2	308.50	**14**	**A**	**20**	0	
STR3	218,435.00	0	B	**20**	0	
eSTR1	578.95	0	**A**	19	**19**	
eSTR2	500.53	0	**A**	19	**19**	
STR-Ni	-	-	-	0	-	
STR-h	-	-	-	0	-	
mddc	3,208,000.00	0	B	**20**	0	

Table B.52: Statistical analysis of the rand-8-20-5 benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
rand-8-20-5: 20(0) instances, $e \in$[18,18]						
GAC2001	1,469,519.00	1	B	**20**	0	
STR1	745,625.56	1	B	18	0	
STR2	810,090.00	0	B	18	0	
STR3	-	-	-	0	-	
eSTR1	20,350.00	**17**	**A**	18	**18**	
eSTR2	767,518.46	0	B	13	0	
STR-Ni	2,599,279.38	0	B	16	0	
STR-h	1,792,535.79	0	B	19	0	
mddc	-	-	-	0	-	

Table B.53: Statistical analysis of the varDimacs benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
varDimacs: 9(3) instances, $e \in$[133,1737]						
GAC2001	127,220.00	1	**A**	**5**	1	44,366.00
STR1	10,592.50	0	**A**	4	1	44,366.00
STR2	1,293.33	1	**A**	3	1	44,366.00
STR3	35,152.50	0	**A**	4	1	44,366.00
eSTR1	12,830.00	0	**A**	4	1	44,042.67
eSTR2	1,546.67	0	**A**	3	1	44,042.67
STR-Ni	6,192.50	**4**	**A**	4	1	44,366.00
STR-h	10,325.00	0	**A**	4	1	44,366.00
mddc	1,013.33	0	**A**	3	1	44,366.00

Table B.54: Statistical analysis of the modifiedRenault benchmark

Algorithm	**Time** (ms)	**#F**	**S**	**#C**	**#BF**	**#NV**
modifiedRenault: 50(0) instances, $e \in$[125,137]						
GAC2001	371,500.00	**12**	**A**	27	5	
STR1	3,989.05	0	**A**	21	5	
STR2	3,228.57	10	**A**	21	5	
STR3	141,972.00	0	B	20	5	
eSTR1	18,592.00	7	**A**	**50**	**50**	
eSTR2	19,334.40	0	**A**	**50**	**50**	
STR-Ni	-	-	-	0	-	
STR-h	786.67	2	**A**	18	4	
mddc	376,944.50	0	B	20	5	

Appendix C

Code Documentation

This documentation gives an overview of the algorithms and their corresponding helper functions.

Below is the file structure and hierarchy of the code files:

```
scsp
|-- include
|   |-- FikayoGAC_algs
|   |   |-- mddc.h
|   |   |-- str1.h
|   |   |-- str2.h
|   |   |-- str3.h
|   |   |-- strn.h
|   |   '-- fikayoinclude.h
|   '-- strinit.h
|-- src
|   |-- FikayoGAC_algs
|   |   |-- mddc.c
```

```
|   |   |-- str1.c
|   |   |-- str2.c
|   |   |-- str3.c
|   |   |-- strn.c
|   |   '-- fikayoinclude.c
|   '-- strinit.c
```

Below, we document the source files used to implement our algorithms, which were added to the scsp-code package created by Shant Karakashian. The code repository is located on the Department of Computer Science and Engineering of the University of Nebraska-Lincoln SVN server located at `https://cse.unl.edu/svn/scsp`.[1]

C.1 File Documentation

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

C.1.1 scsp/src/Fikayo_GACalgs/mddc.c File Reference

Functions

- mdd_node * make_node (mdd_node *node, variable *var, llist *list)
- mdd_link * new_mdd_link (void)
- timestamp * new_timestamp (void)
- llist * link_list (variable *pred)
- mdd_node * mddify (constraint *cons, int type, main_structure *m_s)
- mdd_node * mddReduce (mdd_node *T)
- int all_same_terminal (mdd_node *node, int k)

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

- mdd_node * G_prime_found (mdd_node *node, llist *list)
- llist_node * get_node_of_value (llist *list, int k)
- void print_mdd (constraint *cons, mdd_node *node)
- llist * mddc (constraint *cons, int time, variable *cur_var, main_structure *m_s)
- int mddcSeekSupports (mdd_node *cons, int time, int i, main_structure *m_s, llist *G_No, llist *G_Yes)
- llist * restore_G_no (llist *list, int time)
- void remove_G_no (llist *list, int time)
- int mddc_gac_filter (set *undo_set, variable *cur_var, main_structure *m_s, int time)
- llist * llist_dif (llist *llist1, llist *llist2)

Variables

- int accept
- int reject
- int delta
- llist * G_list

C.1.2 scsp/src/Fikayo_GACalgs/str.c File Reference

Functions

- int str_filter (set *undo_set, variable *cur_var, int time, main_structure *m_s)
- int str1_filter (set *undo_set, variable *cur_var, int time, main_structure *m_s)
- int removeValues_from_domain (llist *list1, variable *vars, variable *var, main_structure *m_s)

- int removeValues (llist_node *val_node, variable *vars, variable *var, main_structure *m_s)
- llist * llist3 (llist *llist1, llist *llist2)
- void removeTuples (set *undo_set, constraint *cons, int tups, variable *var, int time)
- int isValid (constraint *cons, int tau, main_structure *m_s)
- int isContainedIn (int valsing, llist *valarray)
- int isContainedIn2 (int valsing, llist *valarray)
- int isContainedIn3 (int valsing, llist *valarray)
- int isEqual_domain (llist *gacvals, llist *doms)
- int isEqual_domain2 (llist *gacvals, llist *doms)
- int isFuture (variable *var)
- int estr1_filter (set *undo_set, variable *cur_var, int time, main_structure *m_s)

C.1.3 scsp/src/Fikayo_GACalgs/str2.c File Reference

Functions

- int str2_filter (set *undo_set, variable *cur_var, int time, main_structure *m_s)
- int isScopeMember (variable *cur_var, int *variables, main_structure *m_s)
- int isValid2 (constraint *cons, int tau, main_structure *m_s)
- int isS_sup (variable *var, llist *list)
- int isPWConsistent (constraint *constr, int tau)
- llist * UpdateCtr (constraint *constr, int tau)
- int estr2_filter (set *undo_set, variable *cur_var, int time, main_structure *m_s)

C.1.4 scsp/src/Fikayo_GACalgs/str3.c File Reference

Functions

- int str3_filter (set *undo_set, variable *cur_var, main_structure *m_s, int time)
- void move (constraint *cons, var_val *vvp, int k, int l)
- void save (constraint *key, llist *store, variable *var)
- void save2 (con_var_val *cvv, int new_val, llist *store, int var_p, int val_p, variable *var)
- st_I * new_st_I (void)
- st_R * new_st_R (void)
- state_save * new_state_save (void)
- void restoreI (main_structure *m_s, variable *var)
- void restoreR (main_structure *m_s, variable *var)
- int str3_GAC (main_structure *m_s, variable *c_var, set *undo_set, int time)
- int get_var_position (variable *var, constraint *con, main_structure *m_s)
- int get_val_position (int val, llist *list)
- void print_row (constraint *cons, main_structure *m_s)

C.1.5 scsp/src/Fikayo_GACalgs/strn.c File Reference

Functions

- int strn_filter (set *undo_set, variable *cur_var, int time, main_structure *m_s)
- int isValid_neg (constraint *cons, int tau, main_structure *m_s)
- void ComputeCount (variable *var, constraint *con, main_structure *m_s)
- void UpdateCount (constraint *con, int tup, variable *cur_var, main_structure *m_s)

Bibliography

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

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

[Briggs and Torczon, 1993] Preston Briggs and Linda Torczon. An Efficient Representation for Sparse Sets. *ACM Lett. Program. Lang. Syst.*, 2(1-4):59–69, March 1993.

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

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

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

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

[Gent *et al.*, 2007] Ian P. Gent, Chris Jefferson, Ian Miguel, and Peter Nightingale1. Data Structures for Generalised Arc Consistency for Extensional Constraints. In *22nd AAAI Conference on Artificial Intelligence (AAAI 07)*, pages 379–393, 2007.

[Gharbi *et al.*, 2014] Nebras Gharbi, Fred Hemery, Christophe Lecoutre, and Olivier Roussel. Sliced Table Constraints: Combining Compression and Tabular Reduction. In *International Conference on the Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2014)*, volume LNCS 8451, pages 120–135. Springer, 2014.

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

[Jefferson and Nightingale, 2013] Christopher Jefferson and Peter Nightingale. Extending Simple Tabular Reduction with Short Supports. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, volume LNCS 8451, pages 573–579. Springer, 2013.

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

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

[Karakashian, 2013] Shant Karakashian. *Practical Tractability of CSPs by Higher Level Consistency and Tree Decomposition*. PhD thesis, University of Nebraska-Lincoln, 2013.

[Katsileros and Walsh, 2007] George Katsileros and Toby Walsh. A Compression Algorithm for Large Arity Extensional Constraints. In *International Conference on Principles and Practice of Constraint Programming (CP 2007)*, volume 4741 of *Lecture Notes in Computer Science*, pages 379–393. Springer, 2007.

[Lecoutre *et al.*, 2012] Christophe Lecoutre, Chavalit Likitvivatanavong, and Ronald Yap. A Path-Optimal GAC Algorithm for Table Constraints. In *20th European Conference on Artificial Intelligence (ECAI 2012)*, pages 510–515, 2012.

[Lecoutre *et al.*, 2013] Christophe Lecoutre, Anastasia Paparrizou, and Kostas Stergiou. Extending STR to a Higher-Order Consistency. In *Proceedings of the twenty-seventh AAAI Conference on Artificial Intelligence (AAAI 2013)*, pages 576–582, 2013.

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

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

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

[Li *et al.*, 2013] Hongbo Li, Yanchun Liang, Jinsong Guo, and Zhanshan Li. Making Simple Tabular Reduction Works on Negative Table Constraints. In *Proceedings of the twenty-seventh AAAI conference on Artificial Intelligence (AAAI 2013)*, pages 1629–1630, 2013.

[Mackworth, 1977] Alan K. Mackworth. On Reading Sketch Maps. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 598–606, 1977.

[Mohr and Henderson, 1986] Roger Mohr and Thomas C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Perez and Régin, 2014] Guillaume Perez and Jean-Charles Régin. Improving GAC-4 for Table and MDD Constraints. In *International Conference on Principles and Practice of Constraint Programming (CP 2014)*, volume LNCS 8656, pages 606–621. Springer, 2014.

[Schneider *et al.*, 2014] Anthony Schneider, Robert J. Woodward, Berthe Y. Choueiry, and Christian Bessiere. Improving Relational Consistency Algorithms Using Dynamic Relation Partitioning. In *International Conference on Principles and Practice of Constraint Programming (CP 2014)*, volume LNCS 8656, pages 688–704. Springer, 2014.

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

[Wallace, 1993] Richard J. Wallace. Why AC-3 is Almost Always Better than AC-4 for Establishing Arc Consistency in CSPs. In *Proceedings of the 13 th International Joint Conference on Artificial Intelligence*, pages 239–245, Chambéry, France, 1993.

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

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

[Xia and Yap, 2013] Wei Xia and Roland H. C. Yap. Optimizing STR Algorithms with Tuple Compression. In *International Conference on the Principles and Practice of Constraint Programming (CP 2013)*, volume LNCS 8124, pages 724–732. Springer, 2013.

[Xu *et al.*, 2007] Ke Xu, Frédéric Boussemart, Fred Hemery, and Christophe Lecoutre. Random Constraint Satisfaction: Easy Generation of Hard (Satisfiable) Instances. *Artificial Intelligence*, 171(8-9):514–534, 2007.