University of the Aegean
Department of Information and Communication Systems Engineering
Artificial Intelligence Laboratory

Adaptive Strategies for Solving Constraint Satisfaction Problems

Thanasis Balafoutis

Submitted in total fulfilment of the requirements
of the degree of Doctor of Philosophy

June 2011

Abstract

A major challenge in constraint programming is to develop efficient generic approaches to solve instances of the constraint satisfaction problem (CSP). In recent years, adaptive approaches for solving CSPs have attracted the interest of many researchers. General speaking, a strategy that uses the results of its own search experience to modify its subsequent behavior does adaptive search.

In this dissertation we explore adaptive strategies for backtracking search on various levels. First, we investigate adaptive search-guiding heuristics for ordering variables in CSPs. These adaptive heuristics learn and use information from every node explored in the search tree, whereas traditional static and dynamic heuristics only use information about the initial and current nodes. We then perform a wide empirical evaluation of the proposed variable ordering heuristics and compare them with the current state-of-the-art variable ordering strategies.

Concerning constraint propagation which is used as an inference mechanism in order to simplify a problem so as to make it easier to solve, we explore adaptive strategies for ordering the different revisions performed when enforcing arc consistency algorithms.

Next, we propose adaptive branching heuristics for splitting the search tree. The application of these heuristics results in an adaptive branching scheme. Experiments with instantiations of the proposed generic heuristics confirm that search with adaptive branching outperforms search with a fixed branching scheme on a wide range of problem.

Finally, we propose a new a generic approach for branching where the variable's domains are grouped into sets by using the scores assigned to values by a value ordering heuristic, and a clustering algorithm from machine learning.

In general, this dissertation contributes to the design and implementation of adaptive and autonomous constraint solvers that have the ability to advantageously modify modelers decisions that typically in mainstream CP solvers are taken prior to search.

Acknowledgements

First of all, I would like to express my deeply felt gratitude to my supervisor Professor Kostas Stergiou for his warm encouragement, thoughtful guidance and friendship. I feel blessed to have worked with him and I have greatly benefited from his knowledge, experience and personality.

My special thanks go also to Professor Efstathios Stamatatos for his kindly help on many procedural issues concerning my PhD studies and for his helpful discussion on clustering algorithms.

I am grateful to Professors fr. George Anagnostopoulos, George Pavlos and Emmanuel Sarris from Democritus University of Thrace for their unselfish hospitality in their laboratories during my stay in Xanthi.

I also warmly thank my fellow student Anastasia Paparrizou for her help and support.

Finally, I thank the Greek Ministry of Education for their generous support.

This work is dedicated to my wife Christina and to our children. Without their love and support it would not have been possible.

Declarations

Part of the material presented in this thesis has been previously published in conference, journal and workshop papers. We now give details.

Parts of **Chapter 3** are included in the following papers:

[1] T. Balafoutis and K. Stergiou. On conflict-driven variable ordering heuristucs. In *Proceedings of the ERCIM workshop - CSCLP*, 2008.

[2] T. Balafoutis and K. Stergiou. Conflict directed variable selection strategies for constraint satisfaction problems. In *Proceedings of the 6th Hellenic Conference on Artificial Intelligence (SETN)*, pages 29–38, 2010.

Parts of **Chapter 4** are included in the following papers:

[3] T. Balafoutis and K. Stergiou. Experimental evaluation of modern variable selection strategies in constraint satisfaction problems. In *Proceedings of the 15th RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion.*, 2008.

[4] T. Balafoutis and K. Stergiou. Evaluating and Improving Modern Variable and Revision Ordering Strategies in CSPs. *Fundamenta Informaticae*, 102(3-4):229–261, 2010.

Parts of **Chapter 5** are included in the following papers:

[5] T. Balafoutis and K. Stergiou. Exploiting constraint weights for revision ordering in arc consistency algorithms. In *Proceedings of the ECAI-2008 workshop on Modeling and Solving Problems with Constraints*, 2008.

[4] T. Balafoutis and K. Stergiou. Evaluating and Improving Modern Variable and Revision Ordering Strategies in CSPs. *Fundamenta Informaticae*, 102(3-4):229–261, 2010.

Parts of **Chapter 6** can be found in:

[6] T. Balafoutis and K. Stergiou. Adaptive branching for constraint satisfaction problems. In *Proceedings of the ECAI'10*, pages 855–860, 2010.

Parts of **Chapter 7** can be found in:

[7] T. Balafoutis, A. Paparrizou, and K. Stergiou. Experimental Evaluation of Branching Schemes for the CSP. In *Proceedings of the TRICS workshop at CP-2010*, pages 1–12, 2010.

LIST OF KNOWN CITATIONS

Citations for paper [1]:

1. J.Vion and S. Piechowiak. Handling Heterogeneous Constraints in Revision Ordering Heuristics. In *Proceedings of the TRICS workshop at CP-2010*, 2010.

2. B. Bontoux. Techniques hybrides de recherche exacte et approcheé: application à des problèmes de transport. PhD Thesis, Universiti D'Avignon, France, 2008.

3. C. Lecoutre. Constraint Networks Techniques and Algorithms. ISTE/Wiley 2009.

Citations for paper [5]:

1. A. Zanarini. Exploiting global constraints for search and propagation. PhD Thesis, Université de Montréal, Canada, 2010.

2. M. D. Moffitt. On the modelling and optimization of preferences in constraint-based temporal reasoning. Artificial Intelligence, 2010.

3. C. Lecoutre. Constraint Networks Techniques and Algorithms. ISTE/Wiley 2009.

Citation for paper [3]:

1. A. Zanarini. Exploiting global constraints for search and propagation. PhD Thesis, Université de Montréal, Canada, 2010.

Other Publications

1. T. Balafoutis, and K. Stergiou. Algorithms for stochastic CSPs. In *Proceedings of CP-2006*, pages 44–58, 2006.

2. T. Balafoutis, A. Paparrizou, K. Stergiou, and T. Walsh. Improving the Performance of maxRPC. In *Proceedings of CP-2010*, pages 69–83, 2010.

3. T. Balafoutis, A. Paparrizou, K. Stergiou, and T. Walsh. New Algorithms for of max Restricted Path Consistency. Constraints (accepted with minor revisions).

Contents

List of Figures

List of Tables

If your daily life seems poor, do not blame it; blame yourself, tell yourself that you are not poet enough to call forth its riches.

R. M. Rilke

1 Introduction

Constraint Satisfaction Problems (CSPs) involve finding a value for each one of a set of problem variables where constraints specify that some subsets of values cannot be used together. They can model a wide range of combinatorial problems and have many applications in artificial intelligence, operations research, programming languages, databases and other areas of computer science. Common applications include scheduling, verification, design, configuration and games [38].

As a simple example of constraint satisfaction, consider a sports league scheduling problem, where we try to build the schedule of matches between teams (e.g. football teams). In this problem various constraints are naturally revealed: i) Each team must play each other exactly twice (once home and once away), ii) No team can play more than two consecutive home or away matches, iii) The number of times that a team plays two consecutive home or away matches must be minimum, iv) Teams that use the same stadium cannot play home games at the same date, v) Games between top teams must occur at certain dates (due to TV coverage).

Constraint solvers take a real-world problem like this, represented in terms of decision variables and constraints, and search for a solution. A *solution* is an assignment of a single value from its domain to each variable such than no constraint is violated. A problem may have one, many, or no solutions. A problem that has one or more solutions is *satisfiable* or *consistent*. If there is no possible assignment of values to variables that satisfies all the constraints, then the problem is *unsatisfiable* or *inconsistent*.

It has been shown that the CSP, in its general form, is NP-hard [59].

This means that it is unlikely that an efficient general-purpose algorithm exists that does not have a worst-case time complexity exponential in the size of the problem. However, in many practical applications, the instances that arise have special structure that enable them to be solved more efficiently [23].

Once a CSP has been identified and modeled there is a whole host of problem solving techniques that have been developed for solving it [83]. In general, a CSP can be solved either systematically, as with backtracking, or using forms of local search which are typically incomplete. A backtracking search algorithm performs a depth-first traversal of a search tree, where the branches out of a node represent alternative choices that may have to be examined in order to find a solution, and the constraints are used to prune subtrees containing no solutions. Basic backtrack search builds up a partial solution by choosing values for variables until it reaches a dead end, where the partial solution cannot be consistently extended. When it reaches a dead end it undoes the last choice it made and tries another. This is done in a systematic manner that guarantees that all possibilities will be tried. It improves on simply enumerating and testing of all candidate solutions by brute force in that it checks to see if the constraints are satisfied each time it makes a new choice, rather than waiting until a complete solution candidate containing values for all variables is generated. The backtrack search process is often represented as a search tree, where each node (below the root) represents a choice of a value for a variable, and each branch represents a candidate partial solution. Discovering that a partial solution cannot be extended then corresponds to pruning a subtree from consideration. Backtracking search algorithms typically are complete. That is they guarantee that a solution will be found if one exists, and can be used to show that a CSP does not have a solution or to find a provably optimal solution.

Since backtracking search is not guaranteed to terminate within polynomial time - in general there is no polynomial algorithm for CSPs - the research community has spent a considerable amount of effort on maximizing the practical efficiency of backtracking search. Usually this is done by combining backtracking search with constraint propagation mecha-

nisms to filter inconsistent values, and by making use of effective heuristics to guide search. Regarding the latter, backtracking algorithms are typically guided by variable and value ordering heuristics and make use of a branching scheme to divide the search tree while the algorithm traverses it.

The objective of the work presented in this thesis is to investigate adaptive search strategies in order to increase the practical efficiency of backtracking search. General speaking, a strategy that uses the results of its own search experience to modify its subsequent behavior does adaptive search. In other words, a search-guiding strategy is said to be adaptive when it makes choices that depend on the current state of the problem instance an well as previous states. Thus, an adaptive strategy learns, in the sense that it takes account of information concerning the subtrees already been explored.

In problem solving, there exist several pure adaptive approaches which are based on computational intelligence methods and techniques like genetic algorithms, ant colony optimization, swarm intelligence, e.t.c. Also, for local search, there exist some adaptive meta-heuristic approaches like simulated annealing. But all these techniques are suitable and working well only for specific problem classes like scheduling and timetabling. None of these methods has been accepted as a robust general-purpose method for CSPs. Therefore, their applicability is limited to specific problems. In addition, these methods are incomplete. That is they cannot guarantee that a solution will be found even if one exists and prove insolvability.

On the other hand, general purpose constraint programming (CP) solvers are based on backtracking search and they can be efficiently used to solve a wide range of problems in AI and other areas of computer science. But the mainstream CP solvers (like Ilog Solver [48], Gecode [74], Choco [52], e.t.c.) do not include adaptive components in their search mechanisms. The notion of adaptiveness, in these solvers, is restricted only to the usage of certain variable ordering heuristics (like *dom/wdeg* and *impacts*).

Most of the CSP solvers are composed of three main components: i) a modeling language, ii) a set of filtering algorithms for specialized (global)

constraints and iii) search strategies (algorithms and heuristics). Modeling languages are used by the CSP solvers in order to provide a representation of CP problems. That is, defining problem variables and their values, expressing the constraints, handling symmetries, defining viewpoints, e.t.c. Filtering algorithms are based on properties of constraint networks. The idea is to exploit such properties in order to identify some nogoods, where a nogood corresponds to a partial assignment (i.e. a set of variable assignments) that can not lead to any solution. Properties that allow identifying nogoods of size 1 are called domain filtering consistencies. Search is used to traverse the search space of a CSP in order to find a solution. For most of the complete CSP solvers, it respectively corresponds to constraint propagation and depth-first search with backtracking guided by some heuristics. Our thesis is concerned with the third component of CSP solvers, namely search strategies.

1.1 Our contributions

As we already mentioned, a search strategy for solving a CSP, use variable and value ordering heuristics to guide search, and make use of a branching scheme to divide the search tree. Also, it interacts with a propagation queue for making inference and filtering the domains. These are, in a few words, the topics that our work is concerned (except value ordering).

In this dissertation we explore adaptive strategies for backtracking search on various levels. First, we investigate adaptive search-guiding heuristics for ordering variables in CSPs [6], [8]. These adaptive heuristics learn and use information from every node explored in the search tree, whereas traditional static and dynamic heuristics only use information about the initial and current nodes. We then perform a wide empirical evaluation of the proposed variable ordering heuristic and compare them with the current state-of-the-art variable ordering strategies [4], [9].

Concerning constraint propagation which is used as an inference mechanism in order to simplify a problem so as to make it easier to solve, we explore adaptive strategies for ordering the different revisions performed when performing constraint propagation [5], [9].

Next, we propose adaptive branching heuristics for splitting the search tree. The application of these heuristics results in an adaptive branching scheme. Experiments with instantiations of the proposed generic heuristics confirm that search with adaptive branching outperforms search with a fixed branching scheme on a wide range of problem [7].

Finally, we propose a new a generic approach for branching where the variable's domains are grouped into sets by using the scores assigned to values by a value ordering heuristic, and a clustering algorithm from machine learning [2].

In general, the work of this thesis contributes to the design and implementation of adaptive and autonomous constraint solvers that have the ability to advantageously modify modeler's decisions that typically in mainstream CP solvers are taken prior to search.

1.2 Structure and content

The remainder of this thesis is divided into six Chapters. Chapter 2 contains the basic core information for understanding this thesis. We formally introduce Constraint Satisfaction Problems with the formalism that surrounds them. We then introduce some basic constraint propagation concepts that will be used in the following Chapters. These are generalized arc consistency and max-restricted path consistency. Finally we present lookahead backtracking search algorithms that apply different levels of filtering after each decision.

In Chapter 3 we survey the literature on search-guiding heuristics. We start with the static, or fixed, variable ordering heuristics that keep the same ordering throughout search and then, we review dynamic variable ordering heuristics. These heuristics take account of the current state of the instance being solved. Next we present two adaptive heuristics that are widely considered to be state-of-the-art. The first uses the concept of impacts which measure the importance of a value assignment in terms of the search space trimming caused through propagation, while the second is based on recording information about failures in the form of constraint weights. Finally, we introduce new adaptive conflict-directed heuristics

for complete backtrack search algorithms. By noting the constraint responsible for each value deletion, it is possible to implement different weighting strategies. We also use an aging mechanism, as in some SAT solvers, which periodically divides the value of all weights by a constant, thereby giving greater importance to conflicts discovered recently. Our last heuristic tries to better identify contentious constraints by detecting all the possible conflicts after a failure.

In Chapter 4, we experimentally evaluate the most recent and powerful variable ordering heuristics, and new variants of them, over a wide range of benchmarks. This experimental analysis is divided in two parts. In the first part, we evaluate our new proposed conflict-driven adaptive heuristics. Results from various random, academic and real world problems show that some of the proposed heuristics are quite competitive compared to existing ones and in some cases they can increase efficiency. In the second part, we experimentally evaluate the performance of the most recent and powerful heuristics over a wide range of benchmarks, in order to reveal their strengths and weaknesses. All these new heuristics have been tested over a narrow set of problems in their original papers and they have been compared mainly with older heuristics. Hence, there is no comprehensive view of the relative strengths and weaknesses of these heuristics.

The main propagation method used by CP solvers is arc consistency. Coarse grained arc consistency algorithms operate by maintaining a list of arcs (or variables) that records the revisions that are still to be performed. It is well known that the performance of such algorithms is affected by the order in which revisions are carried out. Based on our observation concerning the interaction between conflict-driven variable ordering heuristics and revision ordering heuristics, in Chapter 5, we extend the use of failures discovered during search to devise new, efficient and adaptive revision ordering heuristics. We show that these adaptive heuristics can not only reduce the numbers of constraints checks and list operations, but also cut down the size of the explored search tree. Results from various benchmarks demonstrate that some of the proposed heuristics can boost the performance of the conflict-driven heuristics up to 5 times.

In Chapter 6, we consider branching schemes for the CSP. Branching decisions repeatedly split the search tree into two or more subtrees. We first make a detailed experimental comparison between the existing fixed branching schemes under a variety of different variable ordering heuristics. Next, we develop two generic heuristics that can be applied at successful right branches once the variable ordering heuristic chooses to branch on a variable other than the current one. At this point the heuristics are used to decide whether the advice of the variable ordering heuristic will be followed or not. The application of these heuristics results in an adaptive branching scheme that dynamically switches between the fixed branching schemes. Both of our heuristics can be used in tandem with any backtracking search algorithm and variable ordering heuristic. The first heuristic is based on measuring the difference between the scores that the variable ordering heuristic assigns to its selected variable and the current variable. The second heuristic is based on the use of a secondary advisor to decide if the variable ordering heuristic will be followed or not. Experiments with instantiations of the two generic heuristics confirm that search with adaptive branching outperforms search with a fixed branching scheme on a wide range of problems.

In Chapter 7, we propose and study a generic set branching method where the partition of a domain into sets is created using the scores assigned to values by a value ordering heuristic, and a machine learning clustering algorithm.

Finally, we conclude in Chapter 8 with a recapitulation of the contributions of this thesis and the opportunities it presents for further research.

Do not spoil what you have by desiring what you have not; but remember that what you now have was once among the things you only hoped for.

Epicurus

2 Formal Background

In this chapter we formally introduce Constraint Satisfaction Problems with the formalism that surrounds them. We then introduce some basic constraint propagation concepts that will be used throughout this thesis. These are generalized arc consistency and max-restricted path consistency. Finally we present lookahead backtracking search algorithms that apply different levels of filtering after each decision.

2.1 Constraint satisfaction problems

A *Constraint Satisfaction Problem* (CSP) is defined as a tuple (X, D, C) where: $X = \{x_1, \ldots, x_n\}$ is a set of n variables, $D = \{D(x_1), \ldots, D(x_n)\}$ is a set of domains, one for each variable, with maximum cardinality d, and $C = \{c_1, \ldots, c_e\}$ is a set of e constraints. Each constraint c is a pair $(var(c), rel(c))$, where $var(c) = \{x_1, \ldots, x_m\}$ is an ordered subset of X, and $rel(c)$ is a subset of the *Cartesian* product $D(x_1) \times \ldots \times D(x_m)$ that specifies the allowed combinations of values for the variables in $var(c)$. Each constraint c, has a *scope* which is the set of variables in the constraint. In this thesis, a binary constraint c with $var(c) = \{x_i, x_j\}$ will be denoted by c_{ij}, and $D(x_i)$ will denote the current domain of variable x_i. A *global constraint* is a constraint that captures a relation between a non-fixed number of variables. Each tuple $\tau \in rel(c)$ is an ordered list of values $(a_1, \ldots, a_m)$ such that $a_j \in D(x_j)$,$j = 1, \ldots, m$. A tuple $\tau \in rel(c_i)$ is *valid* iff none of the values in the tuple has been removed from the domain of the corresponding

variable.

The *arity* of a constraint is the number of variables in the scope of the constraint. The *degree* of a variable x_i, denoted by $\Gamma(x_i)$, is the number of constraints in which x_i participates.

A partial assignment is a tuple consisting pairs, each pair consisting of an instantiated variable and the value that is assigned to it in the current search node. A full assignment is one containing all *n* variables. A solution to a CSP is a full assignment such that no constraint is violated.

The CSP is a generalization of the propositional satisfiability (SAT) problem. SAT is the general problem of deciding whether or not a given conjunctive normal form (CNF), called a SAT instance, is satisfiable. This is one of the most studied problems because of its theoretical and practical importance. SAT was also the first problem shown to be NP-complete.

2.2 Constraint propagation

Techniques and algorithms for solving CSPs belong mainly to two main categories: inference and search [28, 34]. Inference methods aim to simplify a problem so as to make it easier to solve, while preserving its semantics, i.e. its set of solutions. Simplification can be achieved by transforming the set of variables and constraints, or by discarding incompatible combinations of values.

To apply inference methods on a CSP, local deductions are iteratively performed until a fixed point is reached or more generally, a certain stopping condition is met. Quite often in practice, a local inference is made possible by reasoning from a single constraint, and corresponds to the removal of a value belonging to the domain of a variable involved in this constraint. Interestingly, as soon as a local inference is performed, the conditions to trigger new inferences may hold, since variables are typically shared by several constraints. This mechanism of propagating the results of local inferences from constraints is called *constraint propagation* and is achieved by filtering algorithms.

At the core of a finite domain constraint programming solver is a constraint propagation engine. Classically, constraint propagation is guided

by events concerning variables or constraints. In the context of generic filtering, where a unique procedure is used no mater what the constraints are, the only kind of events considered are when the domain of a variable changes (i.e. when it loses one or more values). In the context of specialized filtering, CP solvers are using propagators for particular constraints. They are usually implemented as specialized algorithms. The constraint solver computes a fixed point of all propagators, maximizing the amount of inference they can contribute. It then splits the problem and solves the resulting smaller problems recursively.

2.2.1 Arc Consistency

A consistency, which is a general property of a constraint network, usually indicates a certain level of local coherence. In most cases, a local consistency is a property defined from particular subsets of variables and constraints. In contrast, global consistency is a precise property that refers to the entire network, guaranteeing in particular that a solution exists.

Arc consistency is the oldest and most well-known way of propagating constraints. This is indeed a very simple and natural concept that guarantees every value in a domain to be consistent with every constraint.

Given a binary CSP (X, D, C) and a variable $x_i \in X$, a value $a_i \in D(x_i)$ is *arc consistent* (AC) iff for every variable $x_j \in X$, s.t. $c_{ij} \in C$, there exists at least one value $a_j \in D(x_j)$ s.t. the pair (a_i,a_j) satisfies c_{ij}. In this case we say that a_j is a *support* of a_i on c_{ij}. That is, a *support* on a constraint is a tuple that is both valid (i.e. can be build from current domains) and allowed by this constraint. A variable is AC iff all values in its domain are AC. A problem is AC iff there is no empty domain in D and all variables are AC. Enforcing AC on a problem results in the removal of all non-supported values from the domains of the variables.

The definition of arc consistency for non-binary constraints, usually called *generalized arc consistency* (GAC) or *domain consistency*, is a direct extension of the definition of AC. In a non-binary CSP a value $a_i \in D(x_i)$ is GAC iff for every constraint c, s.t. $x_i \in vars(c)$, there exists a valid tuple $\tau \in rel(c)$ that includes the assignment of a_i to x_i [62, 60]. In this case τ

is a support of a_i on constraint c. A variable is GAC iff all values in its domain are AC. A problem is GAC iff there is no empty domain in D and all variables are GAC.

A *support check* (consistency check) is a test to find out if a tuple supports a given value. In the case of binary CSPs a support check simply verifies if two values support each other or not. The *revision* of a variable-constraint pair (c, x_i), with $x_i \in vars(c)$, verifies if all values in $D(x_i)$ have support on c. In the binary case the revision of an arc (x_i,x_j) verifies if all values in $D(x_i)$ have supports in $D(x_j)$. We say that a revision is *fruitful* if it deletes at least one value, while it is *redundant* if it achieves no pruning. A *DWO-revision* is one that causes a domain wipeout (*DWO*). That is, it removes the last remaining value(s) from a domain.

AC-3

The most well-known algorithm for arc consistency is the one proposed by Mackworth in [59] under the name AC3. It was initially proposed for binary normalized networks (networks in which all constraints that share the same scope are merged) but in [60] it was extended for GAC in arbitrary networks.

AC3 is the most commonly presented and used algorithm because of its simple and natural structure. Algorithm 1 depicts the main procedure. An arc is a variable pair (x_i, x_j) which corresponds to a directed constraint. Hence, for each binary constraint c_{ij} involving variables x_i and x_j there are two arcs, (x_i, x_j) and (x_j, x_i). Initially, the algorithm inserts all arcs in the revision list Q. This list is usually implemented as a FIFO queue. Then, each arc (x_i, x_j) is removed from the list and revised in turn. If any value in $D(x_i)$ is removed when revising (x_i, x_j), all arcs pointing to x_i (i.e. having x_i as second element in the pair), except (x_i, x_j), will be inserted in Q (if not already there) to be revised. Algorithm 2 depicts function *revise*(x_i, x_j) which seeks supports for the values of x_i in $D(x_j)$. It removes those values in $D(x_i)$ that do not have any support in $D(x_j)$. The algorithm terminates when the list Q becomes empty.

In all the modern CP solvers the constraint propagation engine maintains a propagation list. However, the elements of the list may not be arc,

Algorithm 1 AC3

1: $Q \leftarrow \{(x_i, x_j)\} \mid c_{ij} \in C$ or $c_{ji} \in C, i \neq j$
2: **while** $Q \neq \emptyset$ **do**
3: select and delete an arc (x_i, x_j) from Q
4: **if** REVISE(x_i, x_j) **then**
5: $Q \leftarrow Q \cup \{(x_k, x_i)\} \mid c_{ki} \in C, k \neq j$
6: **end if**
7: **end while**

Algorithm 2 REVISE-3(x_i, x_j)

1: DELETE $\leftarrow$ **false**
2: **for** each $a \in D(x_i)$ **do**
3: **if** $\nexists\, b \in D(x_j)$ such that *(a, b)* satisfies c_{ij} **then**
4: delete *a* from $D(x_i)$
5: DELETE $\leftarrow$ **true**
6: **end if**
7: **end for**
8: **return** DELETE

variables or constraints, like in the list Q presented in Algorithm 1. They may be propagators for particular constraints (i.e. in solvers ILog [48] and Gecode [74]). A propagator is a specialized filtering algorithm that may be different for each constraint.

2.2.2 Max Restricted Path Consistency

Various local consistencies stronger than (G)AC have been proposed for both binary and non-binary constraints [27, 11, 16]. One such example is *Max Restricted Path Consistency* (maxRPC), a strong local consistency for binary constraints introduced in 1997 by Debruyne and Bessiere [25]. A value $a_i \in D(x_i)$ is *max restricted path consistent* (maxRPC) iff it is AC and for each constraint $c_{ij} \in \mathcal{C}$ there exists a value $a_j \in D(x_j)$ that is an AC-support of a_i s.t. the pair of values (a_i, a_j) is *path consistent* (PC) [25]. A pair of values (a_i, a_j) is PC iff for any third variable x_k there exists a value

$a_k \in D(x_k)$ s.t. a_k is an AC-support of both a_i and a_j. In this case a_j is a *PC-support* of a_i in x_j and a_k is a *PC-witness* for the pair (a_i, a_j) in x_k. A variable is maxRPC iff all its values are maxRPC. A problem is maxRPC iff there is no empty domain in $\mathcal{D}$ and all variables are maxRPC.

Another consistency, which is build upon GAC and is stronger than maxRPC, is the Singleton Arc Consistency (SAC) [26]. In SAC, each value it turn is assigned to each variable in turn. After each assignment, GAC is enforced and the satisfiability of the resulting constraint network is checked.

2.3 Backtracking search algorithms

An attempt to solve a constraint satisfaction problem instance generally requires search. Complete solution methods for CSPs are based on depth-first backtracking search. These methods explore the search space in a systematic way and guarantee that a solution will be found if one exists, or that unsatisfiability will be proved, if no solution exists.

To solve CSP instances, backtrack search has become the standard approach, mainly because it requires only a polynomial amount of space. Backtrack search only needs to store the current search path being explored, because it seeks one solution at a time.

In the backtracking algorithm, the current variable is assigned a value from its domain. This assignment is then checked against the current partial solution; if any of the constraints between this variable and the past variables is violated, the assignment is abandoned and another value for the current variable is chosen. If all values for the current variable have been tried, the algorithm backtracks to the previous variable and assigns it a new value. If a complete solution is found, i.e. a value has been assigned to every variable, the algorithm may terminate. If there are no solutions, the algorithm terminates when all possibilities have been considered.

An example of a search tree built by the backtracking algorithm is shown in Figure 2.1, using the 4-queens problem. The n-queens problem requires placing n queens on an $n \times n$ chessboard in such a way that no queen can take any other. Hence no two queens can be on the same row, the same column or the same diagonal of the board. As a CSP, this problem

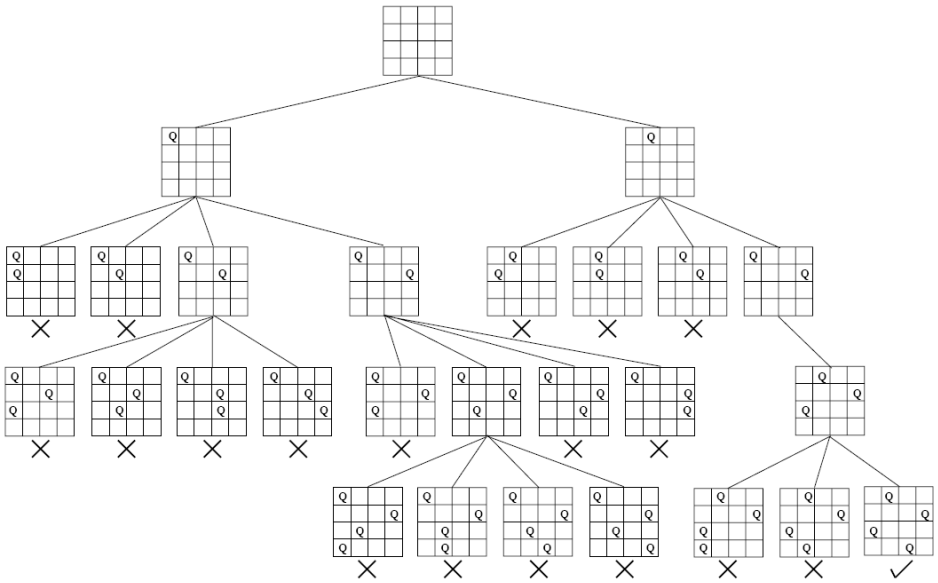

Figure 2.1: *Search tree for the 4-queens problem using simple backtrack.*

has 4 variables, representing the rows of the chessboard, and each variable has domain $\{1, .., 4\}$ representing the 4 columns. However, it is easier to follow the progress of the search if the chessboard representation is used: a Q on a particular square should be taken as meaning that the variable corresponding to that row has been assigned the value corresponding to that column. Deadends, where the algorithm has to backtrack to a previous choice, are marked by crosses, and the solution eventually found is marked by a tick.

The backtracking algorithm only checks the constraints between the current variable and the past variables.

An important technique for improving efficiency is to maintain a level of local consistency during the backtracking search by performing constraint propagation at each node in the search tree. Whenever a new subproblem is created, by removing values from the domains of future variables which are inconsistent with the current assignment, the subproblem is made arc consistent. This will remove further values from the domains of future variables. This has two important benefits. First, removing in-

consistencies during search can dramatically prune the search tree by removing many dead ends and by simplifying the remaining subproblem. In some cases, a variable will have an empty domain after constraint propagation; i.e., no value satisfies the unary constraints over that variable. In this case, backtracking can be initiated as there is no solution along this branch of the search tree. In other cases, the variables will have their domains reduced. If a domain is reduced to a single value, the value of the variable is forced and it does not need to be branched on in the future. Thus, it can be much easier to find a solution to a CSP after constraint propagation or to show that the CSP does not have a solution. Second, some of the most important variable ordering heuristics make use of the information gathered by constraint propagation to make effective variable ordering decisions. As a result of these benefits, it is now standard for a backtracking algorithm to incorporate some form of constraint propagation.

MAC [72], which is the backtrack search algorithm that maintains (generalized) arc consistency during search, is currently considered to be the most efficient complete general-purpose approach to solving CSP instances.

Within MAC the solution process proceeds by iteratively interleaving search phases and propagation phases. During the search phase a variable is instantiated to a value of its domain. Then, in the propagation phase, each constraint checks its consistency (i.e. whether it is feasible or not). In case the constraint is not satisfied, it fails and backtrack occurs; otherwise, constraint inference is performed and reflected on variable domains. Constraint inference removes values from the variables domains that are inconsistent with respect to the partial assignment built so far. Every time a constraint reduces a variable domain, the other constraints that include that variable have to propagate again until the fixed point is reached, that is, no further can be inferred [30]. If, while achieving the fixed point, one of the variables domains becomes empty, then the search fails and it backtracks to reconsider the branching decision. After achieving the fixed point, a new search step is performed. The solution process finishes when a solution is found, that is, a value is assigned to each variable, or when one of the following conditions is achieved: the tree has

been fully explored without finding a solution, a time or a backtrack limit has been reached.

Arc consistency is not the only propagation mechanism that can be maintained during search. Stronger level of consistencies can also be used. As an example, we will refer here the $MmaxRPC$ search algorithm which maintains $maxRPC$ throughout search.

2.4 Variable and value ordering heuristics

A tree search algorithm for constraint satisfaction requires the order in which variables are to be considered to be specified. Using different variable ordering heuristics can drastically effect the efficiency of algorithms solving a CSP instance. The ordering may be either a static ordering, in which the order of the variables is specified before the search begins, and is not changed thereafter, or a dynamic ordering, in which the choice of next variable to be considered at any point depends on the current state of the search.

A common variable ordering heuristic is based on what Haralick and Elliott [44] termed the "fail-first" principle, which they explained as "To succeed, try first where you are most likely to fail".

Having selected the next variable to assign a value to, a search algorithm has to select a value to assign. As with variable ordering, unless values are to be assigned simply in the order in which they appear in the domain of each variable, we should decide how to choose the order in which values should be assigned. A different value ordering will rearrange the branches emanating from each node of the search tree. This is an advantage if it ensures that a branch which leads to a solution is searched earlier than branches which lead to dead ends, provided that only one solution is required. If all solutions are required, or if the whole tree has to be searched because there are no solutions, then the order in which the branches are searched is immaterial.

In the example of Figure 2.1, where the search tree of the 4-queens problem is depicted, variable ordering heuristic select variables in lexicographic order. That is, initially put the first queen in the first row, then the

second queen in the second row and so on. The same lexicographic ordering is also used for ordering values. After a variable assignment, always the first column is tried for placing the queen. If a constraint is violated, the second column is tried next e.t.c. Using a different variable ordering heuristic, simply means that at the top of the search tree the first queen is tried to be placed in a row different than the first. Respectively, using a different value ordering heuristic, a column different than the first will be selected.

2.5 Branching schemes

From the early days of CSP research, search algorithms were usually implemented using either a d-way or a 2-way branching scheme.

In 2-way branching, after a variable x with domain $\{a_1, \ldots, a_d\}$ is chosen, its values are assigned through a sequence of binary choices [73]. The first choice point creates two branches, corresponding to the assignment of a_1 to x (left branch) and the removal of a_1 from the domain of x (right branch).

An alternative branching scheme which was extensively used in the past, and is still used by some solvers, is d-way branching. In this case, after variable x is selected, d branches are built, each one corresponding to one of the d possible value assignments of x. For example, the branching scheme that is used in the 4-queens problem depicted in Figure 2.1 is the d-way branching scheme.

Another technique that is sometimes used is dichotomic domain splitting [31]. This method proceeds by splitting the current domain of the selected variable into two sets, usually based on the lexicographical ordering of the values. In this way branching is performed on the two created sets and the branching factor is reduced to two. Domain splitting is mostly used on optimization problems and especially when the domains of the variables are very large. Although domain splitting drastically reduces the branching factor, it can result in a much deeper search tree since the effects of propagation after a branching decision may be diminished.

2.6 Restarts

Restarting the search is an effective strategy in which the search process is stopping and restarting from the scratch. A cutoff is usually used in order to specify the stopping process. This may be the number of backtracks, the number of wrong decisions, the number of seconds or any other relevant measure.

Restart has been used to solve Boolean satisfaction problems. In particular it is used in the SAT solver Chaff [64] where variable and value choices at the top of the tree are made randomly until a certain depth.

The basic idea behind the association of restart is to give equal chances to all parts of the search space to be explored at the beginning of the search.

2.7 Modern complete CP solvers

A CSP solver is a program which deals with satisfiability of CSP instances. It is said complete when it can prove that an instance is either satisfiable or unsatisfiable. Most of the CSP solvers are composed of two main components: Inference and Search. Inference is used to transform an instance into an equivalent form which is simpler than the original one, while search is used to traverse the search space of the instance in order to find a solution. For (most of the) complete CSP solvers, it respectively corresponds to constraint propagation and depth-first search with backtracking guided by some heuristics.

Modern complete CSP solvers like ILog solver [48], Gecode [74] and Choco [52] offer a high-level modeling language and rich libraries of filtering algorithms for specialized global constraints, search heuristics, symmetry breaking methods, etc. They also offer a clear separation between the model and the solving machinery (providing both modeling tools and innovative solving tools).

As an example of how a CSP problem can be modeled and solved with a modern complete CSP solver, we will describe the well known 4-queens problem using the Choco solver (which is written in Java). This model for the 4-queens problem only involves binary constraints of differences

between integer variables. One can immediately recognize the 4 main elements of any Choco code. First of all, create the model object. Then create the variables by using the Choco API (One variable per queen giving the row (or the column) where the queen will be placed). Finally, add the constraints and solve the problem.

Listing 2.1: *The 4-queens problem modeled with Choco solver*

```
int nbQ = 4;
//1- Create the model
CPModel m = new CPModel();
//2- Create the variables
IntegerVariable[] q = Choco.makeIntVarArray("Q", nbQ, 1, nbQ);
//3- Post constraints
for (int i = 0; i < nbQ; i++) {
  for (int j = i + 1; j < nbQ; j++) {
    int k = j - i;
    m.addConstraint(Choco.neq(q[i], q[j]));
    m.addConstraint(Choco.neq(q[i], Choco.plus(q[j], k)));
    m.addConstraint(Choco.neq(q[i], Choco.minus(q[j], k)));
  }
}
//4- Create the solver
CPSolver s = new CPSolver();
s.read(m);
s.setGeometricRestart(14, 1.5d);
s.setFirstSolution(true);
s.generateSearchStrategy();
s.attachGoal(new DomOverWDegBranchingNew(s, new IncreasingDomain()));
s.launch();
```

Taking a closer look at the Listing 2.1, we can mention that the decisions that modeler has to take are all static. They cannot change during search. For example the java class DomOverWDegBranchingNew() selects the $dom/wdeg$ as variable ordering heuristic and 2-way branching as the default branching scheme. These selections will be followed throughout search.

A way to solve this problem, which is also a novel contribution of our thesis, is to use adaptive methods that can adaptively change modeler decisions throughout search. Modern CSP solvers in general they do not use this kind of adaptive methods. The notion of adaptiveness is only restricted to the selection of adaptive variable ordering heuristics.

2.8 Our complete and generic CSP solver

The constraint problem solving library used in all the experimental studies presented in this thesis is not an adaption or extension of an existing solver, but a solver built from scratch. Various reasons led to the decision to implement a new constraint programming system.

The existing constraint solvers have implemented a wide set of existing techniques and algorithms for handling and solving CSPs. A list of constraint problem languages and solvers can be found at http://4c.ucc.ie /web/ archive/solver.jsp. But in order to experimentally test and try new research algorithms and ideas these solvers cannot directly be used. Therefore, any existing solver would have to be modified.

Modification of the existing code is only possible for the constraint problem solvers with an open source license. Most of those are very complex systems with large amounts of code and features which are not relevant to the requirements of our needs. To modify the code, one needs to be familiar with it. Gaining familiarity with the code of any of the existing implementations would be a huge and difficult task. Most of the surveyed solvers do not have comprehensive documentation of the employed concepts, algorithms, and implementation details. Often unit tests, which aid understanding and enable regression testing, do not exist. Sometimes, the documentation is outdated and the solver not actively maintained anymore.

Choosing a solver to extend to fulfil the requirements at hand is not a trivial task as well. This decision can only be made after evaluating the code of the implementations and estimating the effort of implementing the additional features. Gathering all the required information is a very labor-intensive and time-consuming task.

The reasons stated above had the biggest influence on the decision to implement a new solver from scratch. The basic data structures and algorithms are easy to implement and not all of the sophisticated and difficult algorithms are required. A custom implementation can furthermore be integrated into the overall system architecture more easily. Different algorithms can be evaluated and new concepts explored without needing to investigate possible hidden effects of local changes on the solver as a whole.

2.8.1 The architecture

Our solver is implemented as an object-oriented system, with classes representing the core entities and concepts. Only the methods which are directly needed to interact are exposed, everything else is kept private to the class. Where possible, interfaces are implemented to abstract concepts from particular instantiations of them. The complexity of performing a particular task is hidden in the implementation of the class performing it, other classes which need it done do not need to worry about how it gets done.

The object-oriented nature of the Java programming language, which we have used, makes it possible to take advantage of all the points mentioned above. It is not intended to give here an exhaustive overview of the classes or their methods, but rather to illustrate the general capabilities of the system.

Our CSP solver is a generic solver in the sense that it can handle constraints of any arity. This solver essentially implements the MAC and MmaxRPC search algorithms and supports a wide range of branching schemes, variable and value ordering heuristics.

We have implemented the three most important branching schemes, namely 2-way, d-way and dichotomic branching. Variable ordering can be done with the following heuristics: *dom*, *dom/ddeg*, the majority of the conflict-driven heuristics (dom/wdeg, random probing and our new heuristics described in Section 3.4). We also support the impacts variable ordering heuristic and the heuristic proposed by Correia and Barahona [24].

From the value ordering heuristics, except from the lexicographic ordering, we have also implemented the Geelen's promise heuristic [37]. Our solver also supports the geometric restart policy.

Concerning the local consistency algorithms, our solver includes two implementations of a generic GAC algorithm ($GAC3$ [60] and $GAC3^{rm}$ [55]) and an implementation of maxRPC ($maxRPC3^{rm}$ [3]). Also, it includes an efficient algorithm for handling table constraints of large arities [54].

Concerning the performance of our solver compared to two state-of-the-art solvers, like Abscon 109 [56] and Choco [52], some preliminary results showed that all three solvers visited roughly the same amount of nodes, our solver was consistently slower than Abscon, but sometimes faster than Choco. Note that the aim of our study is to fairly compare the various algorithms and heuristics within the same solver's environment and not to build a state-of-the-art constraint solver. Although our implementation is reasonably optimized for its purposes, it lacks important aspects of state-of-the-art constraint solvers such as specialized propagators for global constraints and intricate data structures.

Only those who will risk going too far can possibly find out how far one can go.

T. S. Eliot

3

Variable Selection Strategies

The order in which variables are assigned by a backtrack search algorithm has been recognized as a key issue for a long time. Using different variable ordering heuristics can drastically effect the efficiency of algorithms solving CSP instances.

In this chapter, we survey the literature on search-guiding heuristics. We start with the static, or fixed, variable ordering heuristics that keep the same ordering throughout search and then, we overview the dynamic variable ordering heuristics. Next we present two adaptive heuristics that can reasonably be considered to be state-of-the-art. The first uses the concept of impacts which measure the importance of a value assignment in terms of the search space trimming caused through propagation, while the second is based on constraint weighting. Finally, we introduce new adaptive conflict-directed heuristics for complete backtrack search algorithms.

3.1 Static ordering

Static, or fixed, variable ordering heuristics (SVOs) keep the same ordering throughout the search, using only structural information about the initial state of search. The simplest such heuristic is *lexico* which orders variables lexicographically. When variables are indexed by integers, *lexico* is usually implemented so as to order the variables according to the value of their index. If $vars(P) = \{x_1, x_2, ..., x_n\}$, then *lexico* will select first x_1, then x_2,...

and finally x_n.

The heuristic *deg*, which is also known as *max degree*, orders variables in sequence of decreasing degree [29]. So variables with the highest initial size of their neighborhood are selected first.

Other known static variable ordering heuristics are the *min width* heuristic which chooses an ordering that minimizes the width of the constraint network [35] and the *min bandwidth* heuristic which minimizes the bandwidth of the constraint graph [89].

Static variable ordering heuristics are weak heuristics and nowadays they are used very rarely.

3.2 Dynamic ordering

Dynamic variable ordering heuristics (DVOs) are considerably more efficient and have thus received much attention in the literature. These heuristics are dynamic because their choices take into account information about the current state of the problem at each point in search. They often obey the *fail-first principle* originally introduced by Haralick and Elliott in [44] i.e. "To succeed, try first where you are most likely to fail".

The first well known dynamic heuristic, introduced by Haralick and Elliott, was *dom* [44]. This heuristic chooses the variable with the smallest remaining domain. The dynamic variation of *deg*, called *ddeg* selects the variable with largest dynamic degree. That is, the variable that is constrained with the largest number of unassigned variables. This principle comes from the simple observation that to find a solution quickly, it is better to move at each step to the most promising subtree, primarily by selecting a value that is most likely to participate in a solution. It is preferable to avoid branching on a value that is globally inconsistent, because this implies exploration of a fruitless subtree, which is clearly a waste of time if there is a solution elsewhere.

By combining *dom* and *deg* (or *ddeg*), the heuristics called *dom/deg* and *dom/ddeg* [14, 80] were derived. These heuristics select the variable that minimizes the ratio of current domain size to static degree (dynamic degree) and can significantly improve the search performance. Other dy-

namic heuristics, based on measures such as the constrainedness of the problem, include the ones proposed in [40, 46]. These heuristics, although conceptually elegant, require extra computation and have only been tested on random problems.

When using variable ordering heuristics, it is a common phenomenon that ties can occur. A tie is a situation where a number of variables are considered equivalent by a heuristic. Especially at the beginning of search, where it is more likely that the domains of the variables are of equal size, ties are frequently noticed. A common tie breaker for the *dom* heuristic is *lexico*, (*dom+lexico* composed heuristic) which selects among the variables with smallest domain size the lexicographically first. Other known composed heuristics are *dom+deg* [36], *dom+ddeg* [19, 79] and *BZ3* [79].

Bessière et al. [12], have proposed a general formulation of DVOs which integrates in the selection function a measure of the constrainedness of the given variable. These heuristics (denoted as *mDVO*) take into account the variable's neighborhood and they can be considered as neighborhood generalizations of the dom and $dom/ddeg$ heuristics. For instance, the selection function for variable X_i is described as follows:

$$H_a^{\circledcirc}(x_i) = \frac{\sum_{x_j \in \Gamma(x_i)} (\alpha(x_i) \circledcirc \alpha(x_j))}{|\Gamma(x_i)|^2} \tag{3.1}$$

where $\alpha(x_i)$ can be any simple syntactical property of the variable such as $|D(x_i)|$ or $\frac{|D(x_i)|}{|\Gamma(x_i)|}$ and $\circledcirc \in \{+, \times\}$. Neighborhood based heuristics have shown to be quite promising.

Correia and Barahona [24] proposed variable orderings, by integrating Singleton Consistency propagation procedures with look-ahead heuristics. This heuristic computes the reduction in the search space after the application of Restricted Singleton Consistency (RSC) [69], for every value of the current variable. Although this heuristic was firstly introduced to break ties in variables with current domain size equal to 2, it can also be used as a tie breaker for any other variable ordering heuristic.

Cambazard and Jussien [21] went a step further by analyzing where the reduction of the search space occurs and how past choices are involved in this reduction. This is implemented through the use of *explanations*. An

explanation consists of a set of constraints C' (a subset of the set C of the original constraints of the problem) and a set of decisions $dc_1, ..., dc_n$ taken during search. An explanation of the removal of value a from variable v can be written as:

$$C' \wedge dc_1 \wedge dc_2 \wedge ... \wedge dc_n \Rightarrow v \neq a$$

Finally, Zanarini and Pesant [90] proposed *constraint-centered heuristics* which guide the exploration of the search space toward areas that are likely to contain a high number of solutions. These heuristics are based on solution counting information at the level of individual constraints. Although the cost of computing the solution counting information is in general high, it has been shown that for certain widely-used global constraints, such information can be computed efficiently.

3.3 Impact-based search strategies

Inspired by integer programming, Refalo introduced an *impact* measure with the aim of detecting choices which result in the strongest search space reduction [70]. An impact is an estimation of the importance of a value assignment for reducing the search space. Refalo proposes to characterize the impact of a decision by computing the Cartesian product of the domains before and after the considered decision. The impacts of assignments for every value can be approximated by the use of averaged values at the current level of observation. If K is the index set of impacts observed so far for assignment $x_i = \alpha$, $\overline{I}$ is the averaged impact:

$$\overline{I}(x_i = \alpha) = \frac{\sum_{k \in K} I^k(x_i = \alpha)}{|K|} \tag{3.2}$$

where I^k is the observed value impact for any $k \in K$.

The impact of a variable x_i can be computed by the following equation:

$$I(x_i) = \sum_{\alpha \in D(x_i)} 1 - \overline{I}(x_i = \alpha) \tag{3.3}$$

An interesting extension of the above heuristic is the use of "node impacts" to break ties in a subset of variables that have equivalent impacts. Node impacts are the accurate impact values which can be computed for any variable by trying all possible assignments.

3.4 Conflict-driven variable selection strategies

Dynamic weighting is an efficient mechanism for identifying hard parts of combinatorial problems. It was first introduced to improve the performance of local search methods. The breakout method [63], simply increases the weights of all current nogoods (tuples corresponding to unsatisfied constraints) whenever a local minimum is encountered, and then uses these weights to escape from local minima. Another method, devised independently [76], increments the weight of all clauses not satisfied by the current assignment. This weight strategy has been shown to enhance dramatically the applicability of a randomized greedy local search procedure (GSAT) for propositional satisfiability testing. Thornton [82] has studied constraint weighting in the context of applying local search to solve CSP instances, and has shown this weighting to be effective on structured problems.

3.4.1 Using constraint weighting

Boussemart et al. [18], inspired from SAT (satisfiability testing) solvers like Chaff [64], proposed conflict-driven variable ordering heuristics. In these heuristics, a weight is assigned on every constraint. These constraint weights are initialized to 1. And every time a constraint causes a failure (i.e. a domain wipeout) during search, its weight is incremented by one. Each variable has a *weighted degree*, which is the sum of the weights over all constraints in which this variable participates. Formally, the weighted degree of a variable is:

$$\alpha_{wdeg}(x_i) = \sum_{C \in \mathcal{C}} weight[C] \quad | \quad x_i \in Vars(C) \wedge |FutVars(C)| > 1 \quad (3.4)$$

where $FutVars(C)$ denotes the uninstantiated variables of a constraint C, $weight[C]$ is its weight and $Vars(C)$ the variables involved in C.

The weighted degree heuristic (*wdeg*) selects the variable with the largest weighted degree. The current domain of the variable can also be incorporated to give the domain-over-weighted-degree heuristic (*dom/wdeg*) which selects the variable with minimum ratio between current domain size and weighted degree.

Both of these heuristics (especially *dom/wdeg*) have been shown to be very effective on a wide range of problems. Their success is based on the increment of the weights of constraints that are involved in hard subproblems. Thus search will focus on the most important parts of the search space.

3.4.2 Random probing

Grimes and Wallace in [43] and later in [86] proposed alternative conflict-driven heuristics that consider value deletions as the basic propagation events associated with constraint weights. These alternatives include the following strategies:

- constraint weights are increased by the size of the domain reduction leading to a DWO (*alldel* heuristic).
- whenever a domain is reduced in size during constraint propagation, the weight of the constraint involved is incremented by 1.
- whenever a domain is reduced in size, the constraint weights are increased by the size of domain reduction.

The last two heuristics record constraints responsible for value deletions and use this information to increase weights.

They also used a sampling technique called *random probing* where several short runs of the search algorithm are made to initialize the constraint weights prior to the final run.

3.4.3 Discussion

As stated in Section 3.4.1, the *wdeg* and *dom/wdeg* heuristics associate a counter, called *weight*, with each constraint of a problem. These counters are updated during search whenever a DWO occurs. If, for example, the MAC algorithm is used for systematic search and AC-3 is applied at every step, a DWO for a variable x_i will be identified inside the *revise* procedure of Algorithm 3. In line 7, the weight of variable x_i will be increased by one, each time a DWO is detected.

Algorithm 3 REVICE-3$(x_i, x_j) : boolean$

```
1: for each a ∈ D(x_i) do
2:     if ∄b ∈ D(x_j) such that c_ij(a, b) then
3:         delete a from D(x_i)
4:     end if
5: end for
6: if D(x_i) = ∅ then
7:     weight[c_ij] + +
8: end if
9: return D(x_i) ≠ ∅
```

Although experimentally it has been shown that these heuristics are extremely effective on a wide range of problems, in theory it seems quite plausible that they may not always assign weights to constraints in an accurate way. This has been noticed by Grimes and Wallace who proposed alternative heuristics that increase the weight of a constraint whenever it causes value deletions. However, the obtained heuristics did not demonstrate any advantage compared to dom/wdeg in practice [43]. To better illustrate our conjecture about the accuracy in assigning weights to constraints, we give the following example.

Example 1 Assume we are using MAC-3 (i.e. MAC with AC-3) to solve a CSP (X, D, C) where X includes, among others, the three variables $\{x_i, x_j, x_k\}$, all having the same domain $\{a, b, c, d, e\}$, and C includes, among others, the two binary constraints c_{ij}, c_{ik}. Also assume that a conflict-driven vari-

able ordering heuristic (e.g. *dom/wdeg*) is used, and that at some point during search AC tries to revise variable x_i. That is, it tries to find supports for the values in $D(x_i)$ in the constraints where x_i participates. Suppose that when x_i is revised against c_{ij}), values $\{a, b, c, d\}$ are removed from $D(x_i)$ (i.e. they do not have a support in $D(x_j)$). Also suppose that when x_i is revised against c_{ik}, value $\{e\}$ is removed from $D(x_i)$ and hence a DWO occurs. Then, the dom/wdeg heuristic will increase the weight of constraint c_{ik} by one but it will not change the weight of c_{ij}.

It is obvious from this example that although constraint c_{ij} removes more values from $D(x_i)$ than c_{ik}, its important indirect contribution to the DWO is ignored by the heuristic. In contrast, note that the *alldel* heuristic of [43] will indeed increase the weight of constraint c_{ij} as soon as this constraint deletes values from $D(x_i)$.

A second point regarding potential inefficiencies of *wdeg* and *dom/wdeg* has to do with the order in which revisions are made by the AC algorithm used. Coarse-grained AC algorithms, like AC-3, use a *revision list* of arcs, variables, or constraints, depending on the implementation, to propagate the effects of variable assignments. It has been shown that the order in which the elements of the list are selected for revision affects the overall cost of search. Hence a number of revision ordering heuristics have been proposed [85, 17]. In general, revision ordering and variable ordering heuristics have different tasks to perform when used in a search algorithm like MAC. Before the appearance of conflict-driven heuristics there was no way to achieve an interaction with each other, i.e. the order in which the revision list was organized during the application of AC could not affect the decision of which variable to select next (and vice versa). The contribution of revision ordering heuristics to the solver's efficiency was limited to the reduction of list operations and constraint checks.

However, when a conflict-driven variable ordering heuristic like *dom/weg* is used, then there are cases where the decision of which arc (or variable) to revise first can affect the variable selection. To better illustrate this interaction we give the following example.

Example 2 Assume that we want to solve a CSP (X, D, C) using a conflict-driven variable ordering heuristic (e.g. dom/wdeg), and that at some

point during search the following AC revision list is formed: Q={(x_1), (x_3), (x_5)}. Suppose that revising x_1 against constraint c_{12} leads to the DWO of $D(x_i)$, i.e. the remaining values of x_1 have no support in $D(x_2)$. Suppose also that the revision of x_5 against constraint c_{56} leads to the DWO of $D(x_5)$, i.e. the remaining values of x_5 have no support in $D(x_6)$. Depending on the order in which revisions are performed, one or the other between the two possible DWOs will be detected. If a revision ordering heuristic R_1 selects x_1 first then the DWO of $D(x_1)$ will be detected and the weight of constraint c_{12} will increased by 1. If some other revision ordering heuristic R_2 selects x_5 first then the DWO of $D(x_5)$ will be detected, but this time the weight of a different constraint (c_{56}) will increased by 1. Although the revision list includes two variables (x_1, x_5) that can cause a DWO, and consequently two constraint weights can be increased (c_{12}, c_{56}), dom/wdeg will increase the weight of only one constraint depending on the choice of the revision heuristic. Since constraint weights affect the choices of the variable ordering heuristic, R_1 and R_2 can lead to different future decisions for variable instantiation. Thus, R_1 and R_2 may guide search to different parts of the search space.

From the above example it becomes clear that known heuristics based on constraint weights are quite sensitive to revision orderings and their performance can be affected by them.

In order to overcome the above described weaknesses that the weighted degree heuristics seem to have, we next describe a number of new variable ordering heuristics which can be seen as variants of *wdeg* and *dom/weg*.

3.4.4 Constraints responsible for value deletions

The first enhancement to *wdeg* and *dom/wdeg* tries to alleviate the problem illustrated in Example 1. To achieve this, we propose to record the constraint which is responsible for each value deletion from any variable in the problem. In this way, once a DWO occurs during search we know which constraints have, not only directly, but also indirectly contributed to the DWO. Based on this idea, when a DWO occurs in a variable x_i, constraint weights can be updated in the following three alternative ways:

- for every constraint that is responsible for any value deletion from $D(x_i)$, we increase its weight by one.
- for every constraint that is responsible for any value deletion from variable $D(x_i)$, we increase its weight by the number of value deletions.
- for every constraint that is responsible for any value deletion from variable $D(x_i)$, we increase its weight by the normalized number of value deletions. That is, by the ratio between the number of value deletions and the size of $D(x_i)$.

The new variable ordering heuristics derived will be referred to as $H1$, $H2$ and $H3$ respectively. Using these alternative ways to increase constraint weights, we can compute the weighted degree of any variable x_i as in [18] using the following equation:

$$\alpha_{wdeg}(x_i) = \sum weight_{H1,2,3}[C] \;\mid\; x_i \in vars(C) \wedge |FutVars(C)| > 1 \quad (3.5)$$

where $FutVars(C)$ denotes the uninstantiated variables in $vars(C)$. The current domain of the variable can also be incorporated to give the heuristics: $dom/wdeg_{H1}$, $dom/wdeg_{H2}$ and $dom/wdeg_{H3}$. The way in which the new heuristics update constraint weights is displayed in the following example.

Example 3 Assume that when solving a CSP (X, D, C), the domain of some variable e.g. x_1 is wiped out. Suppose that $D(x_1)$ initially was $\{a, b, c, d, e\}$ and each of the values was deleted because of constraints: $\{c_{12}, c_{12}, c_{13}, c_{12}, c_{13}\}$ respectively. The proposed heuristics will assign different constraint weights as follows: H1($weight_{H1}[c_{12}]$=$weight_{H1}[c_{13}] = 1$), H2($weight_{H2}[c_{12}] = 3$, $weight_{H2}[c_{13}] = 2$) and H3($weight_{H3}[c_{12}] = 3/5$, $weight_{H3}[c_{13}] = 2/5$)

Heuristics $H1$, $H2$, $H3$ are closely related to the three heuristics proposed by Grimes and Wallace [43]. However, the weights in [43] are increased during constraint propagation in each value deletion for all variables. Our proposed heuristics differ by increasing constraints weights

only when a DWO occurs. As discussed in [43], DWOs seem to be particularly important events in helping identify hard parts of the problem. Hence we focus on information derived from DWOs and not just any value deletion.

Algorithm 4 describes the implementation of the modified revision function for AC-3, depicting the new proposed heuristics. The two dimensional table *responsibleConstraint* in used to record the constraint which is responsible for any value deletion (line 4). In line 8, we show how the three alternative heuristics can increase constraint weights.

Algorithm 4 $newRevice_H(x_i, x_j) : boolean$

```
1:  for each a ∈ D(x_i) do
2:      if ∄b ∈ D(x_j) such that c_ij(a, b) then
3:          delete a from D(x_i)
4:          responsibleConstraint[x_i][a] = c_ij
5:      end if
6:  end for
7:  if D(x_i) = ∅ then
8:      ∀ c_ij ∈ responsibleConstraint[x_i][D(x_i)] weight[c_ij] + + //(H1)
        for each a ∈ D(x_i)
          weight[resposibleConstraint[x_i][a]] + + //(H2)
          weight[responsibleConstraint[x_i][a]]+ = 1/sizeof(D(x_i)) //(H3)
        end for
9:  end if
10: return D(x_i) ≠ ∅
```

3.4.5 Constraint weight aging

Most of the state-of-the-art SAT solvers like BerkMin [41] and Chaff [64], use the strategy of weight "aging". In such solvers, each variable is assigned with a counter that stores the number of clauses responsible for at least one conflict . The value of this counter is updated during search. As soon as a new clause responsible for the current conflict is derived, the counters of the variables, whose literals are in this clause, are incremented

by one. The values of all counters are periodically divided by a small constant greater than 1. This constant is equal to 2 for Chaff and 4 for BerkMin. In this way, the influence of "aged" clauses is decreased and preference is given to recently deduced clauses.

Inspired from SAT solvers, we propose here the use of "aging" to periodically age constraint weights. As in SAT, constraint weights can be "aged" by periodically dividing their current value by a constant greater than 1. The period of divisions can be set according to a specified number of backtracks during search. With such a strategy we give greater importance to recently discovered conflicts. The following example illustrates the improvement that weight "aging" can contribute to the solver's performance.

Example 4 Assume that in a CSP (X, D, C) with D={0,1,2}, we have a ternary constraint $c_{123} \in C$ for variables x_1, x_2, x_3 with disallowed tuples {(0,0,0), (0,0,1), (0,1,1), (0,2,2)}. When variable x_1 is set to a value different from 0 during search, constraint c_{123} is not involved in a conflict and hence its weight will not increase. However, in a branch that includes assignment $x_1 = 0$, constraint c_{123} becomes highly "active" and a possible DWO in variable x_2 or x_3 should increase the importance of constraint c_{123} (more that a simple increment of its weight by one). We need a mechanism to quickly adopt changes in the problem caused by a value assignment. This can be done, by "aging" the weights of the other previously active constraints.

Aging constraint weights can be used in conjunction with any of the newly proposed heuristics and any alternative aging strategy can be followed.

3.4.6 Fully assigned weights

When arc consistency is maintained during search using a coarse grained algorithm like AC-3, a revision list is created after each variable assignment. This list consists of variables, arcs, or constraints, depending on the particular implementation of the AC algorithm. Hereafter we assume a

variable-oriented implementation which is the most efficient alternative [17]. The variables that have been inserted into the list are removed and revised in turn. The revision process stops either if the list becomes empty or if a DWO is detected. When the latter situation occurs for some variable x_i, a weighted-based heuristic like *dom/wdeg* will increase the weight of the constraint that was responsible for the wipeout of $D(x_i)$, and search will continue by backtracking to the most recent choice point. Any variable that remained in the revision list pending revision will be discarded, and a new revision list will be created after the next variable assignment is made.

However, it is possible that some of the remaining variables in the revision list would also cause a DWO if they were selected for revision before x_i i.e., through the use of a different revision ordering heuristic. This leads to a natural presumption that constraints weights are not always fully assigned. That is, each time a DWO occurs when AC is applied during search, only one constraint weight is increased, whereas plausibly, more than one constraint could lead to the DWO. To better illustrate this situation, consider again Example 2 where there are two DWO-revisions in the revision list but only one is detected, and as a result, the weight of only one constraint is incremented.

The question here is how to identify any additional DWO-revisions and consequently increase more than one constraint weight in each call to AC. Is this possible, considering that the variable revisions stop after the first DWO-revision is encountered? We propose here a mechanism that fully assigns weights to all constraints that are potentially responsible for DWOs.

When the first DWO-revision is detected in the revision list, we increase the weight of the responsible constraint by one and then we "freeze" the search procedure and we "undo" the deletions that this revision has made. Then, we continue by revising the remaining variables that are still in the revision list, until the next DWO-revision is identified or the revision list is emptied. If a new DWO-revision is detected, we increase the appropriate constraint weight and "undo" the last value deletions. This process continues until the revision list becomes empty. After that, we "redo" the

deletions of the first DWO-revision detected and we continue search by instantiating the next appropriate variable.

Although this heuristic theoretically seems to be promising, its experimental behavior was not the expected. Experiments on a wide variety of real world problems showed that the variables that remain in the revision list after the detection of the first DWO are very unlikely to cause a new DWO. After a statistical analysis on many real problems we observed that on average, the 96.5% of the revisions are redundant (they achieve no pruning), the 3.3% are fruitful (they delete at least one value) and only the 0.2% are DWO revisions. Thus, in practice we can say that it is almost impossible to identify a second DWO in a revision list.

However, it is also observed that in the same revision list, different revision ordering heuristics can lead to the DWOs of different variables. To better illustrate this, we give the following example.

Example 5 Assume that we use two different revision ordering heuristic R_1, R_2 to solve a CSP (X, D, C), and that at some point during search the following AC revision list is formed for R_1 and R_2. R_1:$\{X_1$,$X_2\}$, R_2:$\{X_2$,$X_1\}$. We also assume the following: *a*) The revision of X_1 deletes some values from the domain of X_1 and it causes the addition of the variable X_3 in the revision list. *b*) The revision of X_2 deletes some values from the domain of X_2 and it causes the addition of the variable X_4 in the revision list. *c*) The revision of X_3 deletes some values from the domain of X_1. *d*) The revision of X_4 deletes some values from the domain of X_2. *e*). A DWO occurs after a sequential revision of X_3 and X_1. *f*) A DWO occurs after a sequential revision of X_4 and X_2. Considering the R_1 list, the revision of X_1 is fruitful and adds X_3 in the list (R_1:$\{X_3$,$X_1\}$). The sequential revision of X_3 and X_1 leads to the DWO of X_1. Considering the R_2 list, the revision of X_2 is fruitful and adds X_4 in the list (R_2:$\{X_4$,$X_2\}$). The sequential revision of X_4 and X_2 leads to the DWO of X_2.

From the above example it is clear that although only one DWO is identified in a revision list, both X_1 and X_2 can be responsible for this. In R_1 where X_1 is the DWO variable, we can say that X_2 is also a "potential" DWO variable i.e. it would be a DWO variable, if the R_2 revision ordering

was used. The question that arises here is: how can we identify the ”potential” DWO variables that exists on a revision list? A first observation that can be helpful in answering this question is that “potential” DWO variables are among variables that participate in fruitful revisions.

Based on this observation, we propose here a new conflict-driven variable ordering heuristic that takes into account the “potential” DWO variables. This heuristic increases the weights of constraints that are responsible for a DWO by one (as $wdeg$ heuristic does) and also, only for revision lists that lead to a DWO, increases by one the weights of constraints that participate in fruitful revisions. Hence, to implement this heuristic we record all variables that delete at least one value during the application of AC. If a DWO is detected, we increase the weight of all these variables.

An interesting direction for future work can be a more selective identification of “potential” DWO variables.

Pleasure in the job puts perfection in the work.

Aristotle

4

Empirical Evaluation of Variable Selection Strategies

In this chapter, we experimentally evaluate the most recent and powerful variable ordering heuristics, and new variants of them, over a wide range of benchmarks. This experimental analysis is divided in two parts. In the first part, we evaluate the new proposed conflict-driven adaptive heuristics which where presented in Chapter 3. In the second part, we experimentally evaluate the performance of the most recent and powerful variable ordering heuristics over a wide range of benchmarks, in order to reveal their strengths and weaknesses. Before presenting the empirical results we give some details about the benchmarks and the CSP solver which was used throughout this study.

4.1 Benchmarks' description

In recent years, the CP research community has collected many series of structures and random instances from different backgrounds. One such collection is represented and stored in XCSP 2.1 format [71] (an XML representation of CSP instances) and allows researchers to control and reproduce experimental results based on them. All these series of instances can be found in C. Lecoutre's web repository (http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html).

Throughout the experimental studies that are presented in this thesis,

we have used many benchmarks taken from this web repository. We have used a wide range of CSP instances taken from different backgrounds. These instances can be divided in the following categories: instances from real world applications, instances following a regular pattern and involving a random generation, academic instances which do not involve any random generation, random instances containing a small structure, pure random instances and, finally, instances which involve only Boolean variables. The selected instances include both binary and non-binary constraints.

In the following sections we give a brief description of all the benchmarks that we have used in our experimental studies.

4.1.1 Real-world Instances

The Radio Link Frequency Assignment Problem

The Radio Link Frequency Assignment Problem (RLFAP) is the task of assigning frequencies to a number of radio links so that a large number of constraints are simultaneously satisfied and as few distinct frequencies as possible are used. A number of modified RLFAP instances have been produced from the original set of problems. These instances have been translated into pure satisfaction problems after removing some frequencies (denoted by f followed by a value)[20]. For example, scen11-f8 corresponds to the instance scen11 for which the 8 highest frequencies have been removed.

The Driver Problem

This set of instances are taken from the Third International Planning Competition [58]. Each problem involves sets of drivers, trucks, locations, and packages. The goal is to deliver packages to different locations, and have the drivers and trucks finish at specified destinations.

The Renault Problem

This is a CSP instance obtained from a Renault Megane configuration problem that has been converted from symbolic domains to numeric ones. Interestingly, this instance, denoted by *renault*, involve large table constraints of high arity. The series *modifiedRenault* contains instances generated from the original configuration one.

4.1.2 Patterned instances

The Graph Coloring Problem

One of the most widely studied combinatorial problems is the Graph Coloring problem. Given a graph, and a set of colors, the problem is to color the nodes such that no edge connects two nodes of the same color.

The Black Hole Problem

The Black Hole problem is the task of moving all cards of 17 fans of 3 cards each to the center pile, the Black hole, which initially only contains the Ace of Spades.

The QCP, QWH and BQWH Problems

The Quasi-group Completion problem (QCP) is the task of determining whether the remaining entries of the partial Latin square can be filled in such a way that we obtain a complete Latin square, ie. a full multiplication table of a quasi-group. The Quasi-group With Holes problem (QWH) is a variant of the QCP as instances are generated in such a way that they are guaranteed to be satisfiable. BQWH instances are satisfiable balanced quasi-group instances with holes.

The Primes Problem

The Primes instances are non-binary intensional instances. All instances are satisfiable. The domains of the variables consist of prime numbers and all constraints are linear equations. The coefficients and constants in the

equations are also prime numbers. These instances are interesting because solving them using Gaussian elimination is polynomial, assuming that the basic arithmetic operations have a time complexity of O(1). In reality this assumption does not hold and the choice of prime numbers in the equations gives rise to large intermediate coefficients in the equations, making the basic operations more time consuming.

The Haystacks Problem

The Haystacks instances are binary unsatisfiable instances. They are parameterized by their size. Instance haystacks-n.xml is the haystack instance of size n. It has $n \times n$ variables and each variable has domain $\{0, ..., n-1\}$. The constraint graph is highly regular, consisting of n clusters: one central cluster and $n-1$ outer clusters. Each cluster is an n-clique. The outer clusters are connected to the central cluster by a single edge (constraint). The instances were designed such that if the variables in the central cluster have singleton domains, only one of the outer clusters contains an inconsistency (this instance is the haystack). The instances were designed such that "learning" to locate the haystack from past experience is difficult. This was done as follows. The assignment to the variables in the central cluster determines which outer cluster is the haystack. Different assignments to the variables in the central cluster lead to different haystacks. The task then consists of finding the haystack and deciding that it is inconsistent, thereby providing a proof that the current assignment to the variables in the central cluster is invalid. If the structure of these instances isn't taken into account then solving them may take quite some time when the size becomes large.

4.1.3 Academic instances

The All-Interval Series Problem

The all-interval series problem is the task of finding a vector s = (s_1, ..., s_n), such that s is a permutation of $\{0, 1, ..., n-1\}$ and the interval vector $v = (|s_2 - s_1|, |s_3 - s_2|, ...|s_n - s_{n-1}|)$ is a permutation of $\{1, 2, ..., n-1\}$.

Each instance is denoted by *series-n*.

The Golomb Ruler Problem

The Golomb Ruler problem is the task of putting n marks on a ruler of length m such that the distance between any two pairs of marks is distinct. Each instance from the model involving ternary (resp. quaternary) constraints is denoted by *ruler-m-n-a3* (resp. *ruler-m-n-a4*).

The Chessboard Coloration Problem

The chessboard coloration problem is the task of coloring all squares of a chessboard composed of r rows and c columns. There are exactly n available colors and the four corners of any rectangle extracted from the chessboard must not be assigned the same color. Each instance is denoted by *cc-r-c-n*.

The Langford Problem

The generalized version of the Langford problem is to arrange k sets of numbers ranging from 1 to n, so that each appearance of the number m is m numbers on from the last. Each instance is denoted by *langford-k-n*.

The Queens Problem

The *n-queens* puzzle is the problem of placing n chess queens on an $n \times n$ chessboard so that none of them can capture any other using the standard chess queen's moves. The queens must be placed in such a way that no two queens attack each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

The Queen Attacking Problem

The Queen Attacking problem is the task of putting a queen and the n^2 numbers $1, ..., n^2$, on a $n \times n$ chessboard so that no two numbers are on the same cell, any number $i + 1$ is reachable by a knight move from the cell containing i and the number of cells containing a prime number that are

not attacked by the queen is 0 (for satisfaction). Each instance is denoted by *queenAttacking-n*.

The Queens-Knights Problem

The Queens-Knights problem is the task of putting on a chessboard of size $n \times n$, q queens and k knights such that no two queens can attack each other and all knights form a cycle (when considering knight moves). In one version of this problem (identified by "add"), a square of the chessboard can be shared by both a queen and a knight and in another one (identified by "mul"), it is not allowed. For the first version, each instance is denoted by *queensKnights-n-k-add* while for the second one, it is denoted by *queensKnights-n-k-mul*.

The Domino Problem

The domino problem, denoted *domino-n-d*, is binary and corresponds to an undirected constraint graph with a cycle. More precisely, n denotes the number of variables, the domains of which are $\{1, ..., d\}$, and there exists $n-1$ equality constraints $X_i = X_{i+1}$ (for all $i \in \{1, ..., n-1\}$) and a trigger constraint $(X_1 = X_{n+1} \wedge X_1 < d) \vee (X_1 = X_n \wedge X_1 = d)$.

4.1.4 Boolean instances

The Dimacs Problem

These are sets of instances taken from the 2nd DIMACS Implementation Challenge [49]:random 3-SAT instances (*aim* and *dubois*), 2-coloring problems (*pret*) and random SAT instances (*jnh*) with variable length clauses (2-14 literals per clause).

4.1.5 Quasi-random instances

The Geometric Problem

The geometric instances are a kind of random instances generated as follows. Instead of a density parameter, a "distance" parameter, dst, is used

such that $dst <= sqrt(2)$. For each variable, two coordinates are chosen at random so the associated point lies in the unit square. Then for each variable pair, (x, y), if the distance between their associated points is less than or equal to dst, the arc (x, y) is added to the constraint graph. Constraint relations are created in the same way as they are for homogeneous random CSP instances. Each instance is prefixed by *geom*.

The Composed Problem

Model B describes a class of CSPs as $< n, k, d, t >$ where n is the number of its variables, k the maximum domain size, d the density, and t the tightness [42]. Consider $< n, k, d, t > s < n', k', d', t' > d''\ t''$, a class of composed problems. Each composed problem has a central component described by $< n, k, d, t >$, s satellites, each described by $< n', k', d', t' >$, and links with density d'' and tightness t'' between its central component and its satellites.

The Ehi Problem

A 3-SAT instance is a SAT instance such that each clause contains exactly 3 literals. Two series of 3-SAT unsatisfiable instances have been converted into CSP instances using the dual method as described in [1]. These series are denoted by *ehi-85* and *ehi-90*.

4.1.6 Random instances

Model B

A class of random CSP instances of model B is denoted as:$< k, n, d, p_1, p_2 >$, where k denotes the arity of each constraint, n the number of variables, d the size of each domain, p_1 the number of constraints and p_2 the number of disallowed tuples of each relation.

Model D

A class of random CSP instances of model D is denoted as:$< k, n, d, p_1, p_2 >$, where k denotes the arity of each constraint, n the number of variables, d

the size of each domain, p_1 the number of constraints and p_2 the the constraint tightness in terms of probability.

Model RB

Given a set V of n variables, first, we select with repetition $m = r \cdot n \cdot ln(n)$ random constraints ($r > 0$ is a constant), each of which is formed by selecting k different variables at random from V, where $k \geq$ is an integer ($k = 2$ for binary CSPs). Next, for each constraint, we uniformly select without repetition $q = p \cdot d \cdot k$ incompatible (unallowed) tuples of values (i.e. nogoods), where $d = n \cdot \alpha$ is the domain size of each variable ($\alpha > 0$ is a constant) [88].

4.2 Experiments with conflict-driven VOHs

In this section we experimentally investigate the behavior of the new proposed conflict variable ordering heuristics, presented in Chapter 3 on several classes of real, academic and random problems.

We compare our proposed heuristics with *dom/wdeg*, which is one the most efficient general purpose heuristics. Regarding the heuristics of Section 3.4.1, we only show results from $dom/wdeg_{H1}$, $dom/wdeg_{H2}$ and $dom/wdeg_{H3}$, denoted as $H1$, $H2$ and $H3$ for simplicity, which are more efficient than the corresponding versions that do not take the domain size into account. We have also include in this experimental comparison results from the *alldel* heuristic, described in Section 3.4.2.

In our tests we have used the following measures of performance: cpu time in seconds (t) and number of visited nodes (n). For these experiments we have used d-way branching, lexicographic value ordering and restarts. Concerning the restart policy, the initial number of allowed backtracks for the first run has been set to 10 and at each new run the number of allowed backtracks increases by a factor of 1.5. Regarding heuristics that employ weight aging, we have selected to periodically decrease all constraint weights by a factor of 2, with the period set to 20 backtracks. All experiments were run on an Intel dual core PC T4200 2GHz with 3GB

RAM.

Our search algorithm is MAC-3, denoting MAC with AC-3, coupled with a variable-oriented propagation scheme. Concerning revision ordering inside AC-3, we have used the standard FIFO queue.

Table 4.1: *Cpu times (t), and nodes (n) from frequency allocation problems. Best cpu time is in bold.*

Instance		$dom/wdeg$	$H1$	$H2$	$H3$	$aging$ $dom/wdeg$	$fully$ Assigned	alldel
scen1-f8	t	2,4	**2,1**	2,2	2,5	2,4	2,4	2,4
(sat)	n	1.141	1.038	1.041	1.128	1.134	1.096	1.129
scen1-f9	t	3,6	3,5	3,6	3,6	**2,8**	3,8	2,9
(unsat)	n	1.055	1.029	1.002	1.128	1.040	777	984
scen2-f25	t	6,3	8,4	9,2	7,3	**5,8**	8,1	7,4
(unsat)	n	2.434	2.682	2.735	2.204	1.912	2.571	2.553
scen3-f10	t	**1,43**	1,49	2	2,2	1,46	2	2
(sat)	n	598	670	787	850	591	773	972
scen3-f11	t	6,7	**3,7**	5,4	7,8	5,5	6,1	4
(sat)	n	1.613	801	1.199	1.929	1.263	1.274	880
graph2-f25	t	8,6	2,2	2,8	6,4	3,1	2,7	**1,9**
(unsat)	n	6.701	1.351	2.003	4.303	2.218	1.611	1.292
graph8-f10	t	18,6	15,6	11,8	21,4	7,7	**6,7**	18,7
(sat)	n	9.705	6.559	5.234	9.905	3.631	2.984	8.634
graph9-f10	t	9,8	6,14	6,7	8,3	**6,11**	8,9	10,6
(unsat)	n	3.120	1.640	1.966	2.513	1.750	2.671	3.292
graph14-f27	t	34	26,6	12,3	**8,7**	92,5	88,7	86,1
(sat)	n	28.087	24.037	10.482	7.242	66.534	82.947	86.375
graph14-f28	t	21,4	1,8	29,3	10,8	27,6	36,5	**0,4**
(unsat)	n	16.006	1.078	22.367	9.051	16.017	29.491	199

Table 4.1 shows results from some real world RLFAP instances. The proposed heuristics display in general a slightly better performance than *dom/wdeg*, which achieves the best cpu time only on instance scen3-f10. The "aging" version of the *dom/wdeg* heuristic achieves the best cpu time in three instances and heuristics *H1* and *alldel* in two instances. The "fully assigned" heuristic has a better performance on graph8-f10 instance.

In Table 4.2 we present results from structured instances belonging to the *driver* benchmark class, while in Table 4.3 we give results from non-binary problems. The results are similar to the ones from RLFAPs shown in Table 4.1. We must notice here that the *dom/wdeg* heuristic does not achieve any win, in all the tested experiments. H1, H2 and H3 heuristics

Table 4.2: *Results from the driver problem. Best cpu time is in bold.*

Instance		$dom/wdeg$	$H1$	$H2$	$H3$	$aging$ $dom/wdeg$	$fully$ Assigned	alldel
driverlogw-02c	t	2	1,5	1,4	2,1	**1,1**	3,4	2,3
(sat)	n	1.859	1.553	1.446	1.910	1.009	2.713	1.771
driverlogw-04c	t	1,9	0,7	0,7	2,1	1,7	0,5	**0,3**
(sat)	n	1.946	663	663	2.007	1.357	456	300
driverlogw-05c	t	1,3	1,8	1,3	2,4	1,1	1,4	**0,6**
(sat)	n	1.026	1.473	978	1.894	823	794	506
driverlogw-08cc	t	13,7	4,9	9,5	17,2	7,2	**1,4**	2,7
(sat)	n	8.759	2.582	5.503	8.657	1.915	674	1.365
driverlogw-08c	t	13,4	3,3	2,9	10,7	3,1	1,4	**1,3**
(sat)	n	8.759	2.582	5.503	8.657	1.915	674	1.365
driverlogw-09	t	124,1	36,3	38,6	46,3	**30,4**	123,1	132,3
(sat)	n	55.166	12.768	15.091	20.246	5.010	27.830	27.247

also do not achieve any win in driver problems but they behave well on non-binary instances.

Table 4.3: *Results from non-binary problems. Best cpu time is in bold.*

Instance		$dom/wdeg$	$H1$	$H2$	$H3$	$aging$ $dom/wdeg$	$fully$ Assigned	alldel
series-11	t	1,6	1,1	5,4	14,9	**1**	12,6	4,2
(sat)	n	2.728	19.689	9.287	25.973	1.280	27.229	9.613
series-12	t	15,9	14,8	12,9	5,6	11,7	48,6	**0,04**
(sat)	n	19.651	22.127	16.200	7.576	11.710	87.036	25
series-13	t	5,8	**1,4**	128,1	31,8	41,7	98,5	169,6
(sat)	n	5.504	1.322	128.793	28.347	37.744	133.911	251.466
series-14	t	316,4	172,3	86,1	78,5	9,9	501,2	0,06
(sat)	n	291.836	133.665	80.207	67.371	6.390	487.564	36
ruler-17-7-a3	t	10,8	**3,72**	4,1	3,9	7,6	3,77	6
(unsat)	n	1.378	685	683	666	1.097	658	1.007
ruler-25-8-a3	t	69	97,7	**53,2**	64,4	130	77,2	76,1
(unsat)	n	3.587	5.957	2.552	3.618	6.413	4.882	3.779
ruler-34-8-a3	t	152,3	633,6	78,5	**43,2**	273,7	93,7	151,9
(sat)	n	4.192	24.813	2.090	1.084	6.813	2.962	3.811
ruler-34-9-a3	t	1.409	1.177	900	744	1.634	1.070	**678**
(unsat)	n	36.662	29.166	21.343	16.926	32.297	32.045	13.364

In Table 4.4 we present results from random problems. One can notice here the bigger cpu time variation among all heuristics. A possible explanation for this diversity is the lack of structure that random instances have. Heuristic *aged-H1* displays the best performance in three cases, while the rest of the heuristics at most in one.

Table 4.4: *Results from random problems. Best cpu time is in bold.*

Instance		$dom/wdeg$	$H1$	$H2$	$H3$	$aging$ $dom/wdeg$	$fully$ Assigned	alldel
frb30-15-1	t	14,9	14,3	51,2	50,4	11,1	38,1	**9,8**
(sat)	n	8.103	7.895	26.796	26.371	5.632	20.832	5.497
frb30-15-2	t	**58,2**	68,9	115	114	170,3	86,9	80,2
(sat)	n	32.625	38.212	63.839	63.753	89.505	49.417	46.993
frb30-15-4	t	105,9	111,9	76,3	79,4	35,2	**26,9**	43,3
(sat)	n	58.185	60.145	42.601	42.417	19.128	14.927	25.200
frb30-15-2-mgd	t	66,9	32,8	60,9	59,6	59,4	31	**20,6**
(sat)	n	36.036	17.679	31.559	30.852	30.511	16.030	11.410
frb30-15-3-mgd	t	20,7	10,7	3,7	55,2	4,8	9,3	**2,8**
(sat)	n	11.082	5.496	1.849	28.140	2.368	4.717	1.427
frb30-15-5-mgd	t	16,1	16,2	17,9	23,7	**15,3**	22,1	24,6
(sat)	n	8.132	7.691	8.956	11.903	7.409	10.801	12.472

Table 4.5: *Averaged values for Cpu times (t), and nodes (n) from 6 different problem classes. Best cpu time is in bold.*

Problem Class		$dom/wdeg$	$H1$	$H2$	$H3$	$aged$ $dom/wdeg$	$fully$ assigned	allDel
RLFAP scensMod	t	1,9	2	2,2	2,3	**1,7**	2,2	2,2
(13 instances)	n	734	768	824	873	646	738	809
RLFAP graphMod	t	9,1	**5,2**	6,1	5,5	12,9	13,4	11,1
(12 instances)	n	6168	3448	4111	3295	8478	11108	9346
Driver	t	22,4	7	7,8	11,6	**6,4**	18,8	20
(11 instances)	n	10866	2986	3604	5829	1654	4746	4568
Interval Series	t	34	19,4	23,4	13,3	**6,5**	66,4	17,4
(10 instances)	n	32091	18751	23644	13334	5860	74310	26127
Golomb Ruler	t	274,9	321,4	173,1	**143,4**	342,1	208,3	154,4
(6 instances)	n	7728	10337	4480	3782	7863	6815	3841
geo50-20-d4-75	t	62,8	174,1	72,1	95	69	**57,6**	76
(10 instances)	n	15087	36949	16970	23562	15031	12508	18094
frb30-15	t	37,3	35,1	45,8	57,2	42,3	32,9	**26,1**
(10 instances)	n	20176	18672	24326	30027	21759	17717	14608

Finally, in Table 4.5 we show averaged results from all the tested problem classes. The first two classes are from the RLFA Problem. For the *scensMod* class we have run 13 instances and in this table we present the averaged values for cpu time and nodes visited. Since these instances are quite easy to solve, all the heuristics have almost the same behavior. The *aged* version of the *dom/wdeg* heuristic has a slightly better performance. For the *graphMod* class we have run 12 instances. Here heuristics *H1, H2, H3* that record the constraint which is responsible for each value deletion display better performance. The third problem class is from another real world problem, which is called *Driver*. In these 11 instances the *aged dom/wdeg* heuristic has on average the best behavior. The next 10 instances are from the non-binary academic problem "All Interval Series" and have maximum constraint arity of 3. We must notice here that the *aged dom/wdeg* heuristic, which has the best performance is five times faster compared to *dom/wdeg*.

This good performance that the *aged dom/wdeg* heuristic has, is not generic within different problem classes. This can be seen in the next academic problem class (the well known Golomb Ruler problem) where the *aged dom/wdeg* heuristic, has the worst performance. The last two classes are from the "*geo*"quasi-random instances (random problems which contain some structure) and from the "*frb*" pure random instances that are forced to be satisfiable. Here, although on average the *fullyAssigned* and *allDel* heuristics have the best performance, within each class we observed a big variation in cpu time among all the tested heuristics. A possible explanation for this diversity is the lack of structure that random instances have.

We must also comment that interestingly the *dom/wdeg* heuristic does not achieve any win, in all the tested experiments. As a general comment we can say that experimentally, all the proposed heuristics are competitive with *dom/wdeg* and in many benchmarks a notable improvement is observed.

4.3 Experimental Evaluation of modern VOHs

The aim of this second part of experiments is to evaluate the performance of the most recent and powerful heuristics over a wide range of benchmarks, in order to reveal their strengths and weaknesses. We compare here conflict-driven variable ordering heuristics (*dom/wdeg*, *alldel*, *random probing*, *fully assigned*) with the *impacts*, *node impacts* and *RSC* heuristics. We have also tried combinations of them. This is the first experimental study in the literature where state-of-the-art heuristics like *impacts* and *dom/wdeg* are compared exhaustively.

Most results were obtained using a lexicographic value ordering, but we also evaluated the impact of random value ordering on the relative performance of the heuristics. We employed a geometric restart policy where the initial number of allowed backtracks for the first run was set to 10 and at each new run the number of allowed backtracks increased by a factor of 1.5. In addition, we evaluated the heuristics under a different restart policy and in the absence of restarts. Since our solver does not yet support global constraints, we have left experiments with problems that include such constraints as future work.

In our experiments the random probing technique is run to a fixed failure-count cutoff C = 40, and for a fixed number of restarts R = 50 (these are the optimal values from [43]). After the random probing phase has finished, search starts with the failure-count cutoff being removed and the dom/wdeg heuristic used based on the accumulated weights for each variable. According to [43], there are two strategies one can pursue during search. The first is to use the weights accumulated through probing as the final weights for the constraints. The second is to continue to increment them during search in the usual way. In our experiments we have used the latter approach. Cpu time and nodes for random probing are averaged values for 50 runs. For heuristics that use probing we have measured the total cpu time and the total number of visited nodes (from both random probing initialization and final search). In the next tables (except Table 4.6) we also show in parenthesis results from the final search only (with the random probing initialization overhead excluded).

Concerning impacts, we have approximated their values at the initialization phase by dividing the domains of the variables into (at maximum) four sub-domains.

As a primary parameter for the measurement of performance of the evaluated strategies, we have used the cpu time in seconds (t). We have also recorded the number of visited nodes (n) as this gives a measure that is not affected by the particular implementation or by the hardware used. In all the experiments, a time out limit has been set to 1 hour.

In Section 4.3.1 we give some additional details on the heuristics which we have selected for the evaluation. In Section 4.3.2 we present results from the radio link frequency assignment problem (RLFAP). In Section 4.3.3 we present results from structured and patterned problems. These instances are taken from some academic (langford), real world (driver) and patterned (graph coloring) problems. In Section 4.3.4 we consider instances from quasi-random and random problems. Experiments with non-binary constraints are presented in Section 4.3.5. The last experiments presented in Section 4.3.6 include Boolean instances. In Section 4.3.7, we study the impact of the selected restart policy on the evaluated heuristics, while in Section 4.3.8 we present experiments with random value ordering. Finally in Section 4.4 we make a general discussion where we summarize our results.

4.3.1 Details on the evaluated heuristics

For the evaluation we have selected heuristics from 5 recent papers mentioned above. These are: i) *dom/wdeg* from Boussemart et al. [18], ii) the random probing technique and the "alldel by #del" heuristic where constraint weights are increased by the size of the domain reduction (Grimes and Wallace [43]), iii) Impacts and Node Impacts from Refalo [70], iv) the "RSC" heuristic from Correia and Barahona [24] and, finally, v) our "fully assigned" heuristic [6].

We have also included in our experiments some combinations of the above heuristics. For example, *dom/wdeg* can be combined with RSC (in this case RSC is used only to break ties). Random probing can be applied

to any conflict-driven heuristic, hence it can be used with the *dom/wdeg* and "fully assigned" heuristics. Moreover, the impact heuristic can be combined with RSC for breaking ties.

The full list of the heuristics that we have tried in our experiments includes 15 variations. These are the following: 1) dom/wdeg, 2) dom/wdeg + RSC (the second heuristic is used only for breaking ties), 3) dom/wdeg with random probing, 4) dom/wdeg with random probing + RSC, 5) Impacts, 6) Node Impacts, 7) Impacts + RSC, 8) alldel by #del, 9) alldel by #del + RSC, 10) alldel by #del with random probing, 11) alldel by #del with random probing + RSC, 12) fully assigned, 13) fully assigned + RSC, 14) fully assigned with random probing, and 15) fully assigned with random probing + RSC. In all these variations the RSC heuristic is used only for breaking ties.

4.3.2 RLFAP instances

Results from Table 4.6 show that conflict-driven heuristics (dom/wdeg, alldel and fully assigned) have the best performance. In the final line of Table 4.6 we give the averaged values for all the instances.

Although the Impact heuristic seems to make a better exploration of the search tree on some easy instances (like s2-f25, g14-f27, s11, s11-f12), it is clearly slower compared to conflict-driven heuristics. This is mainly because the process of impact initialization is time consuming. On hard instances, the Impact heuristic has worse performance and in some cases it cannot solve the problem within the time limit. In general we observed that impact based heuristics cannot handle efficiently problems which include variables with relatively large domains. Some RLFA problems, for example, have 680 variables with up to 44 values in their domains.

Table 4.6: *Cpu times (t) from frequency allocation problems. Best cpu time is in bold. The s and g prefixes stand for scen and graph respectively.*

Instance		$d/wdeg$	$d/wdeg$ $r.probe$	$d/wdeg$ RSC	$d/wdeg$ $r.probe$ RSC	$Impact$	$Node$ $Impact$	$Impact$ RSC	$alldel$	$alldel$ $r.probe$	$alldel$ RSC	$alldel$ $r.probe$ RSC	$fully$	$fully$ $r.probe$	$fully$ RSC	$fully$ $r.probe$ RSC
s2-f25	t	**7,4**	29,2	14,2	23,6	14,2	19,5	19,5	9,3	28,5	9,8	22,7	11,1	29,8	9,1	29,4
(unsat)	n	1195	16317	2651	14548	2088	2091	2091	1689	16231	1579	14321	1744	16672	1388	16211
s3-f10	t	2,2	36,5	10,2	33,7	$>1h$	$>1h$	$>1h$	2,3	38,7	5,5	30	**1,2**	37,2	10,4	37,2
(sat)	n	724	18119	728	17781	–	–	–	900	18312	941	17147	472	927	631	18891
s3-f11	t	9,6	41,4	9,9	36,2	$>1h$	$>1h$	$>1h$	**5,3**	47,5	10,4	39,9	9,6	47,7	11,7	48,7
(unsat)	n	1078	18728	861	17211	–	–	–	641	18862	889	17865	1078	18993	1546	19944
g8-f10	t	15	72,5	45,2	62,4	$>1h$	$>1h$	$>1h$	21,3	76,5	14,4	60,7	**10,9**	77,4	15,8	66,4
(sat)	n	4193	26535	6018	22739	–	–	–	6877	27781	3887	23005	3428	27193	4127	24162
g8-f11	t	7	62,7	10,5	54,2	$>1h$	$>1h$	$>1h$	1,6	60,3	1,7	55,3	**0,8**	61	1,1	58,3
(unsat)	n	1450	24244	940	23348	–	–	–	224	23878	455	23668	107	23979	105	24138
g14-f27	t	**18,8**	48,4	82,5	49,1	53,9	216,7	217,1	28,8	48,3	70,1	60,2	39,8	52,5	89,3	52,7
(sat)	n	12251	28337	13106	27785	6284	6284	6284	18143	28019	40211	38901	20820	29211	47655	31925
g14-f28	t	75,3	43,3	18,2	37,2	$>1h$	$>1h$	$>1h$	**0,4**	60,5	57,3	55,7	46,4	51,5	57,1	53,1
(unsat)	n	33556	22303	1459	19544	–	–	–	99	29928	30239	29327	16397	20389	24356	28874
s11	t	5,5	118,1	141,2	157,5	29,3	210,6	224,8	**4**	120,5	56,2	97,4	4,3	127,8	120,3	181,3
(sat)	n	1024	35097	959	29391	834	833	833	947	35788	1540	29080	853	36611	780	35610
s11-f12	t	6,6	56,2	4,8	51,5	22,4	25,7	25,8	3,7	54,1	3,9	48,3	**3,1**	54,3	3,5	52,6
(unsat)	n	1102	24158	981	22893	421	421	421	566	23661	989	21775	386	23865	977	22798
s11-f11	t	6,8	55,6	4,7	51,1	22,1	25,7	25,8	3,7	53,4	3,8	48,5	**3,1**	53,7	3,6	51,8
(unsat)	n	1102	23555	981	22751	421	421	421	522	23101	989	21557	386	23566	977	22564
s11-f10	t	3,5	56,7	4,8	52,4	$>1h$	$>1h$	$>1h$	4,5	58,2	**3,3**	55,8	4,5	59,7	4,6	59,4
(unsat)	n	490	23131	498	22891	–	–	–	556	23664	376	24077	528	23512	631	24972
s11-f9	t	14,3	71,7	18,1	65,6	$>1h$	$>1h$	$>1h$	16,4	72,4	18,9	65,2	**12,1**	72,8	17,1	71,1
(unsat)	n	1412	24261	1384	23441	–	–	–	1906	24547	1753	23287	1156	24781	1150	24763
s11-f8	t	**21,2**	87,1	44,7	79,6	$>1h$	$>1h$	$>1h$	26,3	89,1	40,1	76,8	26,1	89,5	45	83,5
(unsat)	n	2112	28083	2897	24892	–	–	–	2526	27867	3192	24682	2181	27944	2784	27443
s11-f7	t	133,7	189,9	211,2	201,6	$>1h$	$>1h$	$>1h$	**130,6**	191,4	166,8	221,8	137,7	198,5	160,2	203,5
(unsat)	n	12777	39469	20154	42345	–	–	–	13205	39557	14886	45388	12777	24689	15017	42167
s11-f6	t	391,4	402,9	412,7	416,8	$>1h$	$>1h$	$>1h$	**307,6**	488,4	465,2	479,8	330,4	330,4	301,4	320,6
(unsat)	n	34714	61523	40892	62557	–	–	–	27949	63447	37954	69432	28947	29930	19236	29084
Mean cpu time	t	47,9	91,5	68,8	91,5	–	–	–	**37,7**	99,1	61,8	94,5	42,7	89,5	56,8	91,3

Node Impact and its variation, "Impact RSC", are strongly related, and this similarity is depicted in the results. As mentioned in Chapter 3 (Sections 3.2 and 3.3), Node Impact computes the accurate impacts and the "RSC" heuristic computes the reduction in the search space, after the application of Restricted Singleton Consistency. Since node impact computation also uses Restricted Singleton Consistency (it subsumes it), these heuristics differ only in the measurement function that assigns impacts to variables. Hence, when they are used to break ties on the Impact heuristic, they usually make similar decisions.

When "RSC" is used as a tie breaker for conflict-driven heuristics, results show that it does not offer significant changes in the performance. So we have excluded it from the experiments that follow in the next sections, except for the dom/wdeg + RSC combination.

Concerning "random probing", although experiments in [43] show that it has often better performance when compared to simple *dom/wdeg*, our results show that this is not the case when *dom/wdeg* is combined with a geometric restart strategy. Even on hard instances, where the computation cost of random probes is small compared to the total search cost, results show that *dom/wdeg* and its variations are dominant. Moreover, the combination of "random probing" with any other conflict-driven variation heuristic ("alldel" or "fully assigned") does not result in significant changes in the performance. Thus, for the next experiments we have kept only the "random probing" and *dom/wdeg* combination.

Finally, among the three conflict-driven variations, *"alldel"* seems to display slightly better performance on this set of instances.

4.3.3 Structured and patterned instances

This set of experiments contains instances from academic problems (langford), some real world instances from the "driver" problem and 6 patterned instances from the graph coloring problem. Since some of the variations presented in the previous paragraph (Table 4.6) were shown to be less interesting, we have omitted their results from the next tables.

Results in Table 4.7 show that the behavior of the selected heuristics is

close to the behavior that we observed in RLFA problems. Conflict-driven variations are again dominant here. The *dom/wdeg* heuristic has in most cases the best performance, followed by "alldel" and "fully assigned". Impact based heuristics have by far the worst performance. Random probing again seems to be an overhead as it increases both run times and nodes visits.

Table 4.7: *Cpu times (t), and nodes (n) from structured and patterned problems. Best cpu time is in bold.*

Instance		$d/wdeg$	$d/wdeg$ $r.probe$	$d/wdeg$ RSC	$Impact$	$Node$ $Impact$	$Impact$ RSC	$alldel$ $by\ \#del$	$fully$ $assigned$
langford-	t	**42,8**	48,5 (44,1)	52,2	65,5	70	73,8	46,9	48,2
2-9(unsat)	n	65098	64571 (59038)	68901	73477	52174	53201	62171	60780
langford-	t	**364,5**	380 (374,2)	431,2	406,9	660,6	530,7	402,2	395,2
2-10(unsat)	n	453103	422742 (417227)	481909	458285	494407	479092	435599	428681
langford-	t	**584,8**	673,2 (621)	632,8	1094	1917	1531	726,6	676,8
3-11(unsat)	n	140168	134133 (126991)	140391	174418	200558	187091	141734	138919
langford-	t	**65,9**	238,2 (65,3)	101,3	183,4	289,3	301,1	106,7	70,3
4-10(unsat)	n	5438	14024 (4582)	5099	9257	9910	9910	7362	5031
driver-	t	13,6	43,1 (0,7)	31,2	27,8	31,2	31,1	**1,3**	1,4
8c (sat)	n	4500	9460 (420)	3110	431	429	429	660	632
driver-9	t	262,3	305,2 (219,7)	201,1	$> 1h$	1409	2121	**123,5**	167,9
(sat)	n	58759	58060 (46413)	18581	–	19668	60291	13657	20554
will199-5	t	**1,4**	17 (1,7)	5,2	$> 1h$	$> 1h$	$> 1h$	1,7	2,1
(unsat)	n	577	13060 (726)	650	–	–	–	538	582
will199-6	t	15,8	42,9 (21,9)	30,1	$> 1h$	$> 1h$	$> 1h$	**12,7**	13,4
(unsat)	n	4288	22792 (5763)	4582	–	–	–	2852	2846
ash608-4	t	3,3	20,1 (1,8)	81,3	35,1	136,2	123,3	2,6	**1,2**
(sat)	n	3146	21346 (1823)	2291	3860	2452	2293	2586	1266
ash958-4	t	12,8	36,8 (3)	299,2	111,4	$> 1h$	$> 1h$	11,6	**1,2**
(sat)	n	8369	27322 (1992)	3870	5105	–	–	7399	1266
ash313-5	t	**18,2**	134,7 (18,4)	43,2	172,2	442,1	489,7	19,4	19
(unsat)	n	512	10204 (512)	512	512	512	512	512	512
ash313-7	t	**828,4**	1011 (809,6)	1271	1015	$> 1h$	$> 1h$	995,7	1056
(unsat)	n	20587	35135 (19990)	20139	20539	–	–	20411	20406
Mean cpu time	t	**184,4**	245,8	264,9	–	–	–	204,2	204,3

4.3.4 Random instances

In this set of experiments we have selected some quasi-random instances which contain some structure ("ehi" and "geo" problems) and also some purely random instances, generated following Model RB and Model D.

Results are presented in Table 4.8. All the conflict-driven heuristics (*dom /wdeg*, "alldel" and "fully assigned") have much better cpu times compared to impact based heuristics. In pure random problems the "alldel" heuristic has the best cpu times, while in quasi-random instances the three conflict-driven heuristics share a win. Random probing can slightly improve the performance of *dom/wdeg* on Model D problems but it is an overhead on the rest of the instances.

Table 4.8: *Cpu times (t), and nodes (n) from random problems. Best cpu time is in bold.*

Instance		$d/wdeg$	$d/wdeg$ $r.probe$	$d/wdeg$ RSC	$Impact$	$Node$ $Impact$	$Impact$ RSC	$alldel$ $by\ \#del$	$fully$ $assigned$
ehi-85-0	t	2,1	94,2 (0,15)	2,7	11,7	12,1	12	**0,15**	1,2
(unsat)	n	722	8005 (4)	61	3	3	3	4	149
ehi-85-2	t	**1**	101,6 (0,15)	2,4	11,8	12,4	12,4	5,6	1,1
(unsat)	n	248	7944 (5)	12	4	4	4	650	145
geo50-d4-	t	334,9	526 (490,6)	311,3	$> 1h$	$> 1h$	$> 1h$	280	**129,2**
75-2(sat)	n	50483	88615 (76247)	46772	–	–	–	42946	18545
frb30-15-1	t	**10,5**	42 (15,4)	13,2	66,4	295,6	375,6	20,5	15,6
(sat)	n	3557	15833 (4426)	3275	17866	71052	85017	6044	4493
frb30-15-2	t	63,7	123,6 (97,8)	55,4	273,4	**5,4**	391,3	86,8	91,2
(sat)	n	21330	38765 (27458)	20019	79936	1306	81911	26596	26296
40-8-753-	t	76,5	70,9 (45,9)	60,4	2117	404,5	931,3	**50,5**	486,3
0,1 (sat)	n	21164	21369 (13422)	15239	523831	67979	180281	13823	127686
40-11-414-	t	1192	1261 (1234)	1219	$> 1h$	$> 1h$	$> 1h$	1178	**1162**
0,2 (unsat)	n	336691	354778 (345212)	345886	–	–	–	346368	332844
40-16-250-	t	2919	2928 (2895)	3172	$> 1h$	$> 1h$	$> 1h$	**2893**	3038
0,35 (unsat)	n	741883	755386 (743183)	750910	–	–	–	747757	764989
40-25-180-	t	2481	2689 (2632)	2878	$> 1h$	$> 1h$	$> 1h$	**2340**	2606
0,5 (unsat)	n	373742	402266 (385072)	390292	–	–	–	349685	389603
Mean cpu time	t	786,7	870,7	857,1	–	–	–	**761,6**	836,7

4.3.5 Non-binary instances

In this set of experiments we have included problems with non-binary constraints. The first three instances are from the chessboard coloration problem and have maximum arity of 4. The next two instances are from the academic problem "All Interval Series" which have maximum arity of 3, while the last three instances are from a Renault Megane (maximum arity is 10).

Results are presented in Table 4.9. Here again the conflict-driven heuristics have the best performance in most cases. The Impact based heuristics have the best cpu performance in two instances (cc-15-15-2 and series-16), but on the other hand they cannot solve 4 instances within the time limit.

We must also note here that although the "node impact" and "impact RSC" heuristics are slow on chessboard coloration instances, they visit less nodes. In general, with impact based heuristics there are cases where we can have a consistent reduction in number of visited nodes, albeit at the price of increasing the running time.

Random probing is very expensive for non-binary problems, especially when the arity of the constraints is large and the cost of constraint propagation is high. As a result, adding random probing forced the solver to time out on many instances.

4.3.6 Boolean instances

This set of experiments contains instances involving only Boolean variables and non-binary constraints. We have selected a representative subset from Dimacs problems. To be precise, we have included a subset of the "jnhSat" collection which includes the hardest instances from this collection, 4 randomly selected instances from the "aim" set, where all problems are relatively easy to solve, and the first instance from the "pret" and "dubois" sets, which include very hard instances. All the selected instances have constraint arity of 3, except for the "jnhSat" instances which have maximum arity of 14.

Results from these experiments can be found in Table 4.10. The behavior of the evaluated heuristics in this data set is slightly different from

Table 4.9: *Cpu times (t), and nodes (n) from problems with non-binary constraints. Best cpu time is in bold.*

Instance		$d/wdeg$	$d/wdeg$ $r.probe$	$d/wdeg$ RSC	$Impact$	$Node$ $Impact$	$Impact$ RSC	$alldel$ $by\ \#del$	$fully$ $assigned$
cc-10-10-2	t	31	40,6 (30,3)	47,7	31,3	193,1	219,2	**29,9**	33,1
(unsat)	n	16790	20626 (15800)	16544	16161	10370	10233	15639	15930
cc-12-12-2	t	50,7	67,6 (14,3)	79,3	65	523,6	555,6	**49,1**	54,3
(unsat)	n	16897	19429 (49780)	16596	21532	13935	13564	16292	16135
cc-15-15-2	t	98,6	125 (94,5)	159,7	**91,3**	1037	1134	103,6	102,1
(unsat)	n	16948	20166 (14881)	16674	16437	10374	10012	15741	15945
series-16	t	**147,3**	543,9 (516,5)	177,6	$> 1h$	$> 1h$	$> 1h$	$> 1h$	$> 1h$
(sat)	n	49857	155102 (146942)	51767	–	–	–	–	–
series-18	t	$> 1h$	$> 1h$	$> 1h$	$> 1h$	$> 1h$	$> 1h$	$> 1h$	$> 1h$
(sat)	n	–	–	–	–	–	–	–	–
renault-mod-0	t	1285	$> 1h$	2675	$> 1h$	$> 1h$	$> 1h$	1008	**776,2**
(sat)	n	288	–	251	–	–	–	166	179
renault-mod-1	t	2126	$> 1h$	2283	$> 1h$	$> 1h$	$> 1h$	**431,4**	785,4
(unsat)	n	474	–	469	–	–	–	161	234
renault-mod-3	t	2598	$> 1h$	2977	$> 1h$	$> 1h$	$> 1h$	993,5	**435,7**
(unsat)	n	546	-	475	–	–	–	203	176

the behavior that we observed in previous problems. Although conflict-driven heuristics again display the best overall performance, impact based heuristics are in some cases faster.

The main bottleneck that impact based heuristics have, is the time consuming initialization process. On Boolean instances, where the variables have binary domains, the cost for the initialization of impacts is small. And this can significantly increase the performance of these heuristics.

Among the conflict-driven heuristics, the "alldel" heuristic is always better than its competitors. We recall here that in this heuristic constraint weights are increased by the size of the domain reduction. Hence, on binary instances constraint weights can be increased at minimum by one and at maximum by two (in each DWO).

The same good performance of the "alldel" heuristic was also observed in 30 additional instances from the Dimacs problem class ("aim" instances) not shown here. These extended experiments showed that this way of in-

crementing weights seems to work better on Boolean problems where the deletion of a single value is of greater importance compared to problems with large domains, i.e. it is more likely to lead to a DWO.

Table 4.10: *Cpu times (t), and nodes (n) from boolean problems. Best cpu time is in bold.*

Instance		$d/wdeg$	$d/wdeg$ $r.probe$	$d/wdeg$ RSC	$Impact$	$Node$ $Impact$	$Impact$ RSC	$alldel$ $by\ \#del$	$fully$ $assigned$
jnh01	t	10,2	95,2 (3,5)	14,2	**2,2**	13,2	13	4,2	6,2
(sat)	n	970	5215 (362)	515	100	100	100	481	692
jnh17	t	3,1	57,3 (0,5)	18,8	1,9	10,1	9,9	**1,4**	1,5
(sat)	n	477	4914 (189)	1233	132	131	131	216	204
jnh201	t	3	79,6 (2,1)	3,3	2,6	11	10,7	**1,13**	1,14
(sat)	n	336	5222 (121)	168	177	180	180	179	178
jnh301	t	33,4	121 (14,5)	38,2	**2,2**	5,7	5,5	7	8
(sat)	n	2671	6144 (1488)	1541	110	108	108	608	787
aim-50-1-	t	0,15	0,43 (0,13)	0,21	0,82	0,49	0,5	**0,07**	0,08
6-unsat-2	n	1577	6314 (1404)	1412	6774	474	474	691	474
aim-100-1-	t	0,34	1,47 (1,05)	1,42	91	4,3	6,3	**0,16**	0,2
6-unsat-1	n	3592	17238 (10681)	7932	697503	2338	3890	1609	1229
aim-200-1-	t	0,76	1,28 (0,47)	2,1	1,66	1,9	1,8	**0,24**	0,26
6-sat-1	n	4665	11714 (3236)	1371	4747	213	213	1756	1442
aim-200-1-	t	1,9	3,2 (2,4)	5,3	105,9	4,8	8,5	**0,19**	0,23
6-unsat-1	n	12748	26454 (16159)	28548	436746	1615	3654	1255	1093
pret-60-	t	1255	1385 (1385)	$> 1h$	3589	$> 1h$	$> 1h$	**1027**	1108
25 (unsat)	n	44,6M	44,777M (43,773M)	–	95,4M	–	–	42,5M	43,8M
dubois-20	t	1196	1196 (1196)	$> 1h$	$> 1h$	$> 1h$	$> 1h$	**1004**	1245
(unsat)	n	44,9M	44,461M (44,457M)	–	–	–	–	40,5M	43,8M
Mean cpu time	t	250,3	294	857,1	–	–	–	**204,5**	237

4.3.7 The effect of restarts on the results

In all the experiments reported in the previous sections we followed a geometric restart policy. This policy were introduced in [87] and it has been shown to be very effective. However, different restart policies can be applied within the search algorithm, or we can even discard restarts in favor of a single search run. In order to check how the selected restart policy

affects the performance of the evaluated variable ordering heuristics, we ran some additional experiments.

Apart from the geometric restart policy which we used on the previous experiments, we also tried an arithmetic restart policy. In this policy the initial number of allowed backtracks for the first run has been set to 10 and at each new run the number of allowed backtracks increases also by 10. We have also tested the behavior of the heuristics without the use of any restarts.

Selected results are depicted in Table 4.11. Unsurprisingly, results show that the arithmetic restart policy is clearly inefficient. On instances that can be solved within a small number of restarts (like *scen11, ehi-85-297-0, rb30-15-1* and *ash958GPIA-4*), the differences between the arithmetic and the geometric restart policies are small. But, when some problem (like *scen11-f7, aim-200-1-6, langford-4-10* and *cc-12-12-2*) requires a large number of restarts to be solved, the geometric restart policy clearly outperforms the arithmetic one. Importantly, this behavior is independent of the selected variable ordering heuristic.

Comparing search without restart to the geometric restart policy, we can see that the former is more efficient on some instances. But in general restarts are necessary to solve very hard problems. Importantly, the relative behavior of the conflict-driven heuristics compared to impact based heuristics is not significantly affected by the presence or absence of restarts. That is, the conflict-driven heuristics are always faster than the impact based ones, with or without restarts. Some small differences in the relative performance of the conflict-driven heuristics can be noticed when no restarts are used, but they generally have similar cpu times. *Random probing* seems to work better with no restarts, in accordance with the results and conjectures in [86], but this small improvement is not enough for it to become more efficient than the *dom/wdeg*, "*alldel*" and "*fully assigned*" heuristics.

Table 4.11: *Cpu times for the three selected restart policies: without restarts, arithmetic restarts and geometric restarts. Best cpu time is in bold.*

Instance	restart policy	$d/wdeg$	$d/wdeg$ $r.probe$	$d/wdeg$ RSC	$Impact$	$Node$ $Impact$	$Impact$ RSC	$alldel$ $by\ \#del$	$fully$ $assigned$
scen11	no restart	42,5	102,2	148,3	$> 1h$	$> 1h$	$> 1h$	112,4	**41,3**
(sat)	arithmetic	8	109,5	142,7	29	211,3	218,3	**4**	4,5
	geometric	5,5	118,1	141,2	29,3	210,6	224,8	**4**	4,3
scen11-f7	no restart	$> 1h$	**109**	$> 1h$	$> 1h$	$> 1h$	$> 1h$	$> 1h$	$> 1h$
(unsat)	arithmetic	1848	**1464**	1991	$> 1h$	$> 1h$	$> 1h$	3207	2164
	geometric	133,7	189,9	211,2	$> 1h$	$> 1h$	$> 1h$	**130,6**	137,7
aim-200-1-6	no restart	4,8	1,5	4,7	2,3	2,8	3,1	**0,28**	0,31
(unsat)	arithmetic	81,4	150,3	212,7	124,8	9,4	9,2	0,39	**0,27**
	geometric	1,9	3,2	5,3	105,9	4,8	8,5	**0,19**	0,23
ehi-85-297-0	no restart	17,1	90,4	7,1	11,8	12,2	12,4	**0,16**	1,9
(unsat)	arithmetic	2	102,2	2,8	12,8	12,4	12,3	**0,15**	1,18
	geometric	2,1	94,2	2,7	11,7	12,1	12	**0,15**	1,2
frb30-15-1	no restart	**3,2**	30,1	3,6	152,6	215,1	201,5	7,1	3,8
(sat)	arithmetic	15,9	149,2	**15,1**	303,9	626,1	532,5	184,3	201,2
	geometric	**10,5**	42	13,2	66,4	295,6	375,6	20,5	15,6
langford-4-10	no restart	**16,2**	193,7	24,6	59,1	74,6	79,3	24,1	20,8
(unsat)	arithmetic	**521,1**	749,7	557,2	2579	904,1	1293	1011	744,9
	geometric	**65,9**	238,2	101,2	183,4	289,3	301,1	106,7	70,3
cc-12-12-2	no restart	17	27,6	25,4	**16,4**	115,9	98,2	17,9	17,2
(unsat)	arithmetic	2939	**1976**	$> 1h$	$> 1h$	$> 1h$	$> 1h$	2501	2589
	geometric	50,7	67,6	79,3	65	523,6	555,6	**49,1**	54,3
ash958GPIA-4	no restart	10,4	35,6	162,2	383,7	$> 1h$	$> 1h$	**6,3**	6,4
(sat)	arithmetic	13	36,2	310,2	118,3	$> 1h$	$> 1h$	14	**10,7**
	geometric	12,8	36,8	299,2	111,4	$> 1h$	$> 1h$	11,6	**1,2**

4.3.8 Using random value ordering

As noted at the beginning of Section 4.3, all the experiments were ran with a lexicographic value ordering. In order to check if this affects the performance of the evaluated variable ordering heuristics, we have ran some additional experiments. In these experiments we study the performance of the heuristics when random value ordering is used.

Selected results are depicted in Table 4.12 where we show cpu times for both random and lexicographic value ordering. Concerning the random value ordering, all the results presented here are averaged values for 50 runs. Looking at the results and comparing the performance of the heuristics under the different value orderings, we can see some differences in cpu time. However, the relative behavior of the conflict-driven heuristics compared to impact based heuristics is not significantly affected by the use of lexicographic or random value ordering.

Table 4.12: *Cpu times for the two different value orderings: lexicographic and random. Best cpu time for each ordering is in bold.*

Instance	value ordering	$d/wdeg$	$d/wdeg$ $r.probe$	$d/wdeg$ RSC	$Impact$	$Node$ $Impact$	$Impact$ RSC	$alldel$ $by\ \#del$	$fully$ $assigned$
scen11-f7	random	161	232,5	191,3	$> 1h$	$> 1h$	$> 1h$	**157,2**	178,9
(unsat)	lexico	133,7	189,9	211,2	$> 1h$	$> 1h$	$> 1h$	**130,6**	137,7
aim-200-1-6	random	2,3	2,3	6,2	11,9	6,4	6,1	**0,18**	0,23
(unsat)	lexico	1,9	3,2	5,3	105,9	4,8	8,5	**0,19**	0,23
ehi-85-297-0	random	1,3	3,5	5,1	11,4	11,8	11,9	**0,16**	0,8
(unsat)	lexico	2,1	94,2	2,7	11,7	12,1	12	**0,15**	1,2
frb30-15-1	random	39,7	52,8	27,5	120,9	132,4	123,6	32,3	**28,2**
(sat)	lexico	**10,5**	42	13,2	66,4	295,6	375,6	20,5	15,6
langford-4-10	random	**61,2**	229,8	75,4	155,7	255,7	249,6	280,5	83,8
(unsat)	lexico	**65,9**	238,2	101,2	183,4	289,3	301,1	106,7	70,3
cc-12-12-2	random	55,6	74,5	82,4	**51,2**	423,9	437,2	55,8	54,8
(unsat)	lexico	50,7	67,6	79,3	65	523,6	555,6	**49,1**	54,3
ash958GPIA-4	random	5,3	35,6	242,2	106,6	515,4	450,1	**3,8**	3,9
(sat)	lexico	12,8	36,8	299,2	111,4	$> 1h$	$> 1h$	11,6	**1,2**

4.4 A general summary of the results

In order to get a summarized view of the evaluated heuristics, we present six figures. In these figures we have included cpu time and number of visited nodes for the three major conflict-driven variants (*dom/wdeg*, "*alldell*" and "*fully assigned*") and we have compared them graphically to the Impact heuristic (which has the best performance among the impact based heuristics).

Results are collected in Figure 4.1. The left plots in these figures correspond to cpu times and the right plots to visited nodes. Each point in these plots, shows the cpu time (or nodes visited) for one instance from all the presented benchmarks. The y-axes represent the solving time (or nodes visited) for the Impact heuristic and the x-axes the corresponding values for the $dom/wdeg$ heuristic (Figures (a) and (b)), "*alldell*" heuristic (Figures (c) and (d)) and "*fully assigned*" heuristic (Figures (e) and (f)). Therefore, a point above line $y = x$ represents an instance which is solved faster (or with less node visits) using one of the conflict-driven heuristics. Both axes are logarithmic.

As we can clearly see from Figure 4.1 (left plots), conflict-driven heuristics are almost always faster. Concerning the numbers of visited nodes, the right plots do not reflect an identical performance. Although it seems that in most cases conflict-driven heuristics are making a better exploration in the search tree, there is a considerable set of instances where the Impact heuristic visit less nodes.

The main reason for this variation in performance (cpu time versus nodes visited) that the impact heuristic has, is the time consuming process of initialization. The idea of detecting choices which are responsible for the strongest domain reduction is quite good. This is verified by the right plots of Figure 4.1. But the additional computational overhead of computing the "best" choices, really affect the overall performance of the impact heuristic (Figure 4.1, left plots). As our experiments showed the impact heuristic cannot handle efficiently problems which include variables with relatively large domains. For example in the RLFA problems where we have 680 variables with at most 44 values in their domains re-

sults in Table 4.6 verified our hypothesis. On the other hand in problems where variables have only a few values in their domains (as in the Boolean instances of Section 4.3.6) results showed that the impact heuristic is quite competitive.

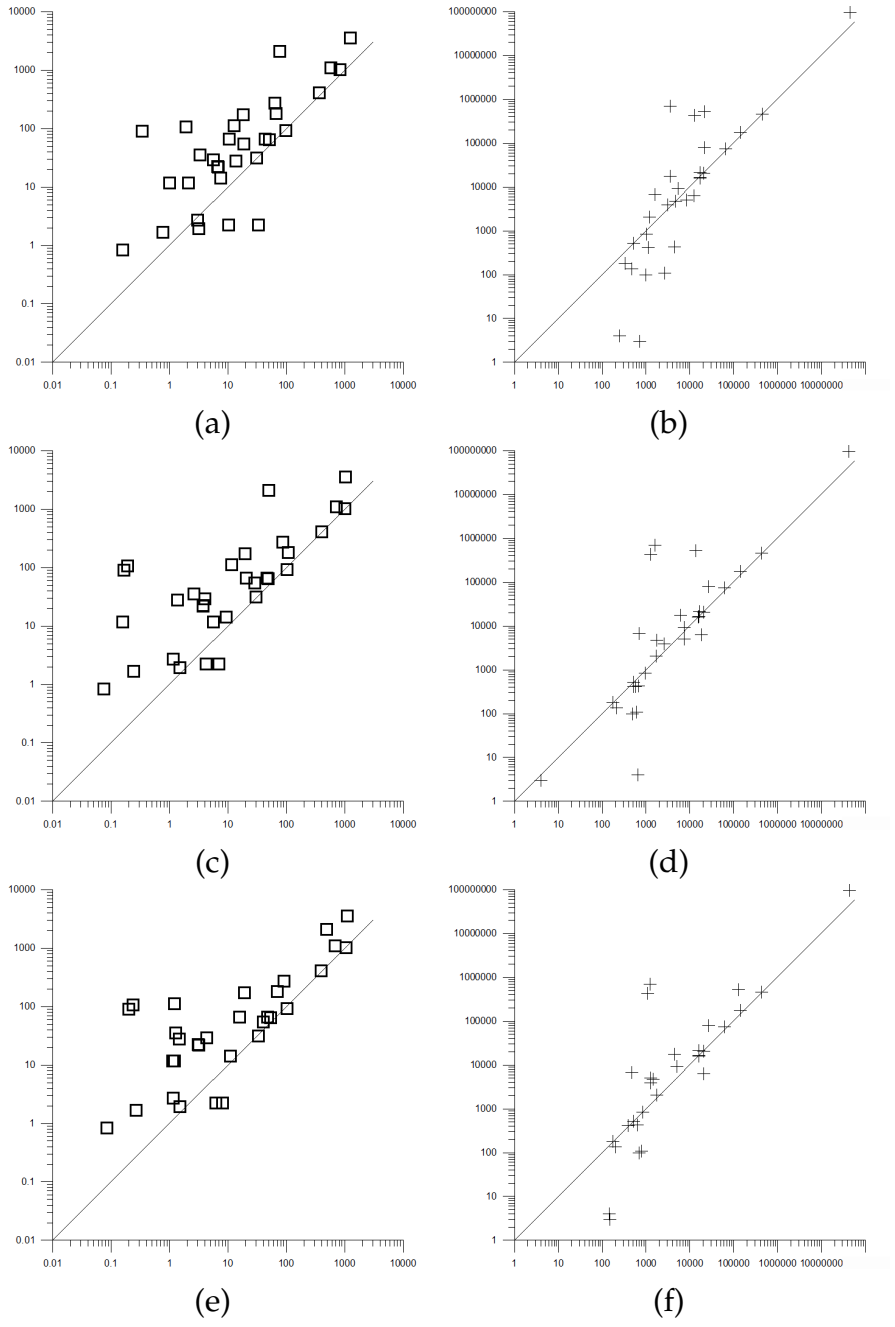

Figure 4.1: *A summary view of run times (left figures) and nodes visited (right figures), for* $dom/wdeg$ *and* $impact$ *heuristics (figures (a),(b)), "*alldell*" and* $impact$ *heuristics (figures (c),(d)), "*fully assigned*" and* $impact$ *heuristics (figures (e),(f)).*

Abundance of knowledge does not teach men to be wise.

Heraclitus

5

Adaptive Revision Ordering in Propagation Algorithms

Any CP solver performs constraint propagation by maintaining a propagation list. The elements of this list may be variables, constraints or arcs. In any case, the order in which the elements of the list are processed plays an important role in the efficiency of the solver.

In arc consistency algorithms, like AC-3, the propagation list records the revisions that are still to be performed. It is well known that the performance of such algorithms is affected by the order in which revisions are carried out. As a result, several heuristics for ordering the elements of the revision list have been proposed. These heuristics exploit information about the original and the current state of the problem, such as domain sizes, variable degrees, and allowed combinations of values, to reduce the number of constraint checks and list operations aiming at speeding up arc consistency computation.

In this chapter, we show how information about constraint weights can be exploited to efficiently order the revision list when AC is applied during search. We propose a number of simple revision ordering heuristics based on constraint weights for arc, variable, and constraint oriented implementations of coarse grained arc consistency algorithms, and compare them to the most efficient existing revision ordering heuristic. Importantly, the new heuristics can not only reduce the numbers of constraints checks and list operations, but also cut down the size of the explored search tree. Results from various structured and random problems demonstrate that

some of the proposed heuristics can offer significant speed-ups.

5.1 Introduction

It is well known that the order in which the elements of the revision list are processed affects the overall cost of search [85, 17, 75]. This is true for solvers that implement variable or constraint based propagation as well as for propagator oriented solvers like Ilog Solver and Gecode. In general, revision ordering and variable ordering heuristics have different tasks to perform when used by a search algorithm like MAC. Prior to the emergence of conflict-driven heuristics there was no way to achieve an interaction with each other, i.e. the order in which the revision list was organized during the application of AC could not affect the decision of which variable to select next (and vice versa). The contribution of revision ordering heuristics to the solver's efficiency was limited to the reduction of list operations and constraint checks.

In this chapter, we first show that the ordering of the revision list can affect the decisions taken by a conflict-driven DVO heuristic. That is, different orderings can lead to different parts of the search space being explored. Based on this observation, we then present a set of new revision ordering heuristics that use constraint weights, which can not only reduce the numbers of constraints checks and list operations, but also cut down the size of the explored search tree. Finally, we demonstrate that some conflict-driven DVO heuristics, e.g. "*alldel*" and "*fully assigned*", are less amenable to changes in the revision list ordering than others (e.g. *dom/wdeg*).

First of all, to illustrate the interaction between a conflict-driven variable ordering heuristic and revision list orderings, we give the following example.

Example 6 *Assume that we want to solve a CSP* (X, D, C)*, where* X *contains* n *variables* $\{x_1, x_2, ..., x_n\}$*, using a conflict-driven variable ordering heuristic (e.g. dom/wdeg), and that at some point during search and propagation the variables pending for revision are* x_1 *and* x_5*. Also assume that two of the constraints in the problem are* $x_1 > x_2$ *and* $x_5 > x_6$*, and that the domains of* x_1, x_2, x_5, x_6

are as follows: $D(x_1) = D(x_5) = \{0, 1\}$, $D(x_2) = D(x_6) = \{2, 3\}$. *Given these constraints and domains, the revision of* x_1 *against* x_2 *would result in the DWO of* x_1*, and the revision of* x_5 *against* x_6 *would result in the DWO of* x_5*. Independent of which variable is selected to be revised first (i.e. either* x_1 *or* x_5*), a DWO will be detected and the solver will reject the current variable assignment. However, depending on the order of revisions, the* dom/wdeg *heuristic will increase the weight of a different constraint. To be precise, if a revision ordering heuristic* R_1 *selects to revise* x_1 *first then the DWO of* $D(x_1)$ *will be detected and the weight of constraint* c_{12} *will be increased by 1. If some other revision ordering heuristic* R_2 *selects* x_5 *first then the DWO of* $D(x_5)$ *will be detected, but this time the weight of constraint* c_{56} *will be increased by 1. Since increases in constraint weights affect the subsequent choices of the variable ordering heuristic,* R_1 *and* R_2 *can lead to different future decisions for variable instantiation. Thus,* R_1 *and* R_2 *may guide search to different parts of the search space.*

From the above example it becomes clear that the revision ordering can have an important impact on the performance of conflict-driven heuristics like *dom/wdeg*. One might argue that a way to overcome this is to continue propagation after the first DWO is detected, try to identify all possible DWOs and increase the weights of all constraints involved in failures. The problem with this approach is threefold: First, it may increase the cost of constraint propagation significantly, second it requires modifications in the way all solvers implement constraint propagation (i.e. stopping after a failure is detected), and third, experiments we have run showed that the possibility of more than one DWO occurring is typically very low. As we will discuss in Section 5.5, some variants of *dom/wdeg* are less amenable to different revision orderings, i.e. their performance do not depend on the ordering as much, without having to implement this potentially complex approach.

5.2 Background

In this section we first review three standard implementations of revision lists for AC, i.e. the arc-oriented, variable-oriented, and constraint-

oriented variants. Then, we summarize the major revision ordering heuristics that have been proposed so far in the literature, before describing the new adaptive revision ordering heuristics we propose.

5.2.1 AC-3 variants

The numerous AC algorithms that have been proposed can be classified into *coarse grained* and *fine grained*. Typically, coarse grained algorithms like AC-3 [59] and its extensions (e.g. AC2001/3.1 [15] and AC-3_d [32]) apply successive revisions of arcs, variables, or constraints. On the other hand, fine grained algorithms like AC-4 [62] and AC-7 [13] use various data structures to apply successive revisions of variable-value-constraint triplets. Here we are concerned with coarse grained algorithms, and specifically AC-3. There are two reasons for this. First, although AC-3 does not have an optimal worst-case time complexity, as the fine grained algorithms do, it is competitive and often better in practice and has the additional advantage of being easy to implement. Second, many constraint solvers that can handle constraints of any arity follow the philosophy of coarse grained AC algorithms in their implementation of constraint propagation. That is, they apply successive revisions of variables or constraints. Hence, the revision ordering heuristics we describe below can be easily incorporated into most of the existing solvers.

As mentioned, the AC-3 algorithm can be implemented using a variety of propagation schemes. We recall here the three variants, following the presentation of [17], which respectively correspond to algorithms with an arc-oriented, variable-oriented or constraint-oriented propagation scheme.

The first one (arc-oriented propagation) is the most commonly presented and used because of its simple and natural structure. Algorithm 5 depicts the main procedure. As explained, an arc is a pair (c_{ij}, x_j) which corresponds to a directed constraint. Hence, for each binary constraint c_{ij} involving variables x_i and x_j there are two arcs, (c_{ij}, x_j) and (c_{ij}, x_i). Initially, the algorithm inserts all arcs in the revision list Q. Then, each arc (c_{ij}, x_j) is removed from the list and revised in turn. If any value in $D(x_j)$

is removed when revising (c_{ij}, x_j), all arcs pointing to x_j (i.e. having x_i as second element in the pair), except (c_{ij}, x_i), will be inserted in Q (if not already there) to be revised. Algorithm 6 depicts function *REVISE*(c_{ij}, x_j) which seeks supports for the values of x_j in $D(x_i)$. It removes those values in $D(x_j)$ that do not have any support in $D(x_i)$. The algorithm terminates when the list Q becomes empty.

Algorithm 5 ARC-ORIENTED AC3

```
1: Q ←{(c_ij, x_j) | c_ij ∈ C and x_j ∈ vars(c_ij)}
2: while Q ≠ ∅ do
3:     select and delete an arc (c_ij, x_j) from Q
4:     if REVISE(c_ij, x_j) then
5:         Q ← Q ∪ {(c_kj, x_k) | c_kj ∈ C, k ≠ i}
6:     end if
7: end while
```

Algorithm 6 REVISE(c_{ij}, x_i)

```
1: DELETE ← false
2: for each a ∈ D(x_i) do
3:     if ∄ b ∈ D(x_j) such that (a, b) satisfies c_ij then
4:         delete a from D(x_i)
5:         DELETE ← true
6:     end if
7: end for
8: return DELETE
```

The variable-oriented propagation scheme was proposed by McGregor [61] and later studied in [22]. Instead of keeping arcs in the revision list, this variant of AC-3 keeps variables. The main procedure is depicted in Algorithm 7. Initially, all variables are inserted in the revision list Q. Then each variable x_i is removed from the list and each constraint involving x_i is processed. For each such constraint c_{ij} we revise the arc (x_j,x_i). If the revision removes some values from the domain of x_j, then variable x_j is inserted in Q (if not already there).

Algorithm 7 VARIABLE-ORIENTED AC3

```
1:  Q ← {x_i | x_i ∈ X}
2:  ∀ c_ij ∈ C, ∀x_i ∈ vars(c_ij), ctr(c_ij, x_i) ← 1
3:  while Q ≠ ∅ do
4:      get x_i from Q
5:      for each c_ij | x_i ∈ vars(c_ij) do
6:          if ctr(c_ij, x_i) = 0 then continue
7:          for each x_j ∈ vars(c_ij) do
8:              if NEEDS-NOT-BE-REVISED(c_ij, x_j) then continue
9:              nbRemovals ← REVISE(c_ij, x_j)
10:             if nbRemovals > 0 then
11:                 if dom(x_j) = ∅ then return false
12:                 Q ← Q ∪ {x_j}
13:                 for each c_jk | c_jk ≠ c_ij ∧ x_j ∈ vars(c_jk) do
14:                     ctr(c_jk, x_j) ← ctr(c_jk, x_j) + nbRemovals
15:                 end for
16:             end if
17:         end for
18:         for each x_j ∈ vars(c_ij) do ctr(c_ij, x_j) ← 0
19:     end for
20: end while
21: return true
```

Algorithm 8 NEEDS-NOT-BE-REVISED(c_{ij}, x_i)

```
1: return (ctr(c_ij, x_i) > 0 and ∄x_j ∈ vars(c_ij) | x_j ≠ x_i ∧ ctr(c_ij, x_j) > 0)
```

Function *NEEDS-NOT-BE-REVISED* [17] given in Algorithm 8, is used to determine relevant revisions. This is done by associating a counter $ctr(c_{ij},x_i)$ with any arc (x_i,x_j). The value of the counter denotes the number of removed values in the domain of variable x_i since the last revision involving constraint c_{ij}. If x_i is the only variable in $vars(c_{ij})$ that has a counter value greater than zero, then we only need to revise arc (x_j,x_i). Otherwise, both arcs are revised.

The constraint-oriented propagation scheme is depicted in Algorithm 9.

Algorithm 9 CONSTRAINT-ORIENTED AC3

```
1:  Q ← {c_ij | c_ij ∈ C}
2:  ∀ c_ij ∈ C, ∀x_i ∈ vars(c_ij), ctr(c_ij, x_i) ← 1
3:  while Q ≠ ∅ do
4:      get c_ij from Q
5:      for each x_j ∈ vars(c_ij) do
6:          if NEEDS-NOT-BE-REVISED(c_ij, x_j) then continue
7:          nbRemovals ← REVISE(c_ij, x_j)
8:          if nbRemovals > 0 then
9:              if dom(x_j) = ∅ then return false
10:             for each c_jk | c_jk ≠ c_ij ∧ x_j ∈ vars(c_jk) do
11:                 Q ← Q ∪ {x_j}
12:                 ctr(c_jk, x_j) ← ctr(c_jk, x_j) + nbRemovals
13:             end for
14:         end if
15:     end for
16:     for each x_j ∈ vars(c_ij) do ctr(c_ij, x_j) ← 0
17: end while
18: return true
```

This algorithm is similar to Algorithm 7. Initially, all constraints are inserted in the revision list Q. Then each constraint c_{ij} is removed from the list and each variable $x_j \in vars(c_{ij})$ is selected and revised. If the revision of the selected constraint c_{ij} is fruitful, then the reinsertion of the constraint c_{ij} in the list is needed. As in the variable-oriented scheme, the same counters are also used here to avoid useless revisions.

5.2.2 Overview of revision ordering heuristics

The standard implementation to process the revision list is a FIFO queue. Alternative implementations can be done with revision ordering heuristics, which is a topic that has received considerable attention in the literature. The first systematic study on this topic was carried out by Wallace and Freuder, who proposed a number of different heuristics that can be

used with the arc-oriented variant of AC-3 [85]. These heuristics, which are defined for binary constraints, are based on three major features of CSPs: (i) the number of acceptable pairs in each constraint (the constraint size or satisfiability), (ii) the number of values in each domain and (iii) the number of binary constraints that each variable participates in (the degree of the variable). Based on these features, they proposed three revision ordering heuristics: (i) ordering the list of arcs by increasing relative satisfiability (*sat up*), (ii) ordering by increasing size of the domain of the variables (*dom j up*) and (iii) ordering by descending degree of each variable (*deg down*).

The heuristic *sat up* counts the number of acceptable pairs of values in each constraint (i.e the number of tuples in the Cartesian product built from the current domains of the variables involved in the constraint) and puts constraints in the list in ascending order of this count. Although this heuristic reduces the list additions and constraint checks, it does not speed up the search process. When a value is deleted from the domain of a variable, the counter that keeps the number of acceptable arcs has to be updated. This process is usually time consuming because the algorithm has to identify the constraints in which the specific variable participates and to recalculate the counters with acceptable value pairs. Also an additional overhead is needed to reorder the list.

The heuristic *dom j up* counts the number of remaining values in each variable's current domain during search. Variables are inserted in the list by increasing size of their domains. This heuristic reduces significantly list additions and constraint checks and is the most efficient heuristic among those proposed in [85].

The *deg down* heuristic counts the current degree of each variable. The initial *degree* of a variable x_i is the number of variables that share a constraint with x_i. During search, the *current degree* of x_i is the number of unassigned variables that share a constraint with x_i. The *deg down* heuristic sorts variables in the list by decreasing size of their current degree. As noticed in [85] and confirmed in [17], the (*deg down*) heuristic does not offer any improvement.

Gent et al. [39] proposed another heuristic called k_{ac}. This heuristic

is based on the number of acceptable pairs of values in each constraint and tries to minimize the constrainedness of the resulting subproblem. Experiments have shown that k_{ac} is time expensive but it performs less constraint checks when compared to *sat up* and *dom j up*.

Boussemart et al. [17] performed an empirical investigation of the heuristics of [85] with respect to the different variants (arc, variable and constraint) of AC-3. In addition, they introduced some new heuristics. Concerning the arc-oriented AC-3 variant, they have examined the *dom j up* as a stand alone heuristic (called dom^v) or together with *deg down* which is used in order to break ties (called $ddeg \circ dom^v$). Moreover, they proposed the ratio *sat up/dom j up* (called dom^c/dom^v) as a new heuristic. Regarding the variable-oriented variant, they adopted the dom^v and $ddeg$ heuristics from [85] and proposed a new one called rem^v. This heuristic corresponds to the greatest proportion of removed values in a variable's domain. For the constraint-oriented variant they used dom^c (the smallest current domain size) and rem^c (the greatest proportion of removed values in a variable's domain). Experimental results showed that the variable-oriented AC-3 implementation with the dom^v revision ordering heuristic (simply denoted dom hereafter) is the most efficient alternative.

5.3 Revision ordering heuristics based on constraint weights

The heuristics described in the previous subsection, and especially dom, improve the performance of AC-3 (and MAC) when compared to the classical queue or stack implementation of the revision list. This improvement in performance is due to the reduction in list additions and constraint checks. A key principle that can have a positive effect on the performance of the AC algorithms is the "ASAP principle" by Wallace and Freuder [85] which urges to "remove domain values as soon as possible". Considering revision ordering heuristics this principle can be translated as follows: When AC is applied during search (within an algorithm such as MAC), to reach as early as possible a failure (*DWO*), order the revision list by putting

first the arc or variable which will guide you to early value deletions and thus, most likely, earlier to a *DWO*.

To apply the "ASAP principle" in revision ordering heuristics, we must use some metric to compute which arc (or variable) in the AC revision list is the most likely to cause failure. Until now, constraint weights have only been used for variable selection. In the next paragraphs we describe a number of new revision ordering heuristics for all three AC-3 variants. These heuristics use information about constraint weights as a metric to order the AC revision list and they can be used efficiently in conjunction with conflict-driven variable ordering heuristics to boost search.

The main idea behind these new heuristics is to handle as early as possible potential *DWO-revisions* by appropriately ordering the arcs, variables, or constraints in the revision list. In this way the revision process of AC will be terminated earlier and thus constraint checks can be reduced significantly. Moreover, with such a design we may be able to avoid many *redundant revisions*. As will become clear, all of the proposed heuristics are lightweight (i.e. cheap to compute) assuming that the weights of constraints are updated during search.

Arc-oriented heuristics are tailored for the arc-oriented variant where the list of revisions Q stores arcs of the form (c_{ij},x_i). Since an arc consists of a constraint c_{ij} and a variable x_i, we can use information about the weight of the constraint, or the weight of the variable, or both, to guide the heuristic selection. These ideas are the basis of the proposed heuristics described below. For each heuristic we specify the arc that it selects. The names of the heuristics are preceded by an "a" to denote that they are tailored for arc-oriented propagation.

- a_wcon: selects the arc (c_{ij},x_i) such that c_{ij} has the highest weight *wcon* among all constraints appearing in an arc in Q.
- a_wdeg: selects the arc (c_{ij},x_i) such that x_i has the highest weighted degree *wdeg* among all variables appearing in an arc in Q.
- $a\_dom/wdeg$: selects the arc (c_{ij},x_i) such that x_i has the smallest ratio between current domain size and weighted degree among all variables appearing in an arc in Q.

- $a\_dom/wcon$: selects the arc (c_{ij},x_i) having the smallest ratio between the current domain size of x_i and the weight of c_{ij} among all arcs in Q.

The call to one of the proposed arc-oriented heuristics can be attached to line 3 of Algorithm 5. Note that heuristics $a\_dom/wdeg$ and $a\_dom/wcon$ favor variables with small domain size hoping that the deletion of their few remaining values will lead to a DWO. To strictly follow the "ASAP principle" which calls for early value deletions we intend to evaluate the following heuristics in the future:

- $a\_dom/wdeg\_inverse$: selects the arc (c_{ij},x_i) such that x_j has the smallest ratio between current domain size and weighted degree among all variables appearing in an arc in Q.

- $a\_dom/wcon\_inverse$: selects the arc (c_{ij},x_i) having the smallest ratio between the current domain size of x_j and the weight of c_{ij} among all arcs in Q.

Heuristics $a\_dom/wdeg\_inverse$ and $a\_dom/wcon\_inverse$ favor revising arcs (c_{ij},x_i) such that x_j, i.e. the other variable in constraint c_{ij}, has small domain size. This is because in such cases it is more likely that some values in $D(x_i)$ will not be supported in $D(x_j)$, and hence will be deleted.

Variable-oriented heuristics are tailored for the variable-oriented variant of AC-3 where the list of revisions Q stores variables. For each of the heuristics given below we specify the variable that it selects. The names of the heuristics are preceded by an "v" to denote that they are tailored for variable-oriented propagation.

- v_wdeg: selects the variable having the highest weighted degree *wdeg* among all variables in Q.

- $v\_dom/wdeg$: selects the variable having the smallest ratio between current domain size and *wdeg* among all variables in Q.

The call to one of the proposed variable-oriented heuristics can be attached to line 4 of Algorithm 7. After selecting a variable, the algorithm

revises, in some order, the constraints in which the selected variable participates (line 5). Our heuristics process these constraints in descending order according to their corresponding weight.

Finally, the constraint-oriented heuristic c_wcon selects a constraint c_{ij} from the AC revision list having the highest weight among all constraints in Q. The call to this heuristic can be attached to line 4 of Algorithm 9. One can devise more complex constraint-oriented heuristics by aggregating the weighted degrees of the variables involved in a constraint. However, we have not yet implemented such heuristics.

5.4 Experiments with revision ordering heuristics

In this section we experimentally investigate the behavior of the new revision ordering heuristics proposed above on several classes of real world, academic and random problems. We only include results for the two most significant arc consistency variants: arc and variable oriented. We have excluded the constraint-oriented variant since this is not as competitive as the other two [17].

We compare our heuristics with dom, the most efficient previously proposed revision ordering heuristic. We also include results from the standard FIFO implementation of the revision list which always selects the oldest element in the list (i.e. the list is implemented as a queue). The selected variable ordering heuristic in these experiments is the *dom/wdeg* heuristic. In our tests we have used the following measures of performance: cpu time in seconds (t), number of visited nodes (n), number of constraint checks (c) and the number of times (r) a revision ordering heuristic has to select an element in the propagation list Q.

Tables 5.1 and 5.2 show results from some real-world RLFAP instances. In the arc-oriented implementation of AC-3 (Table 5.1), heuristics a_wcon, mainly, and $a\_dom/wcon$, to a less extent, decrease the number of constraint checks and list revisions compared to dom. However, the decrease is not substantial and is rarely leads into a decrease in cpu times. The no-

table speed-up observed for problem s11-f6 is mainly due to the reduction in the number of visited nodes offered by the two new heuristics. a_wdeg and $a\_dom/wdeg$ are less competitive, indicating that information about the variables involved in arcs is less important compared to information about constraints.

The variable-oriented implementation (Table 5.2) is clearly more efficient than the arc-oriented one. This confirms the results of [17]. Concerning this implementation, heuristic $v\_dom/wdeg$ in most cases is better than dom and $queue$ in all the measured quantities (number of visited nodes, constraint checks and list revisions). Importantly, these savings are reflected on notable cpu time gains making the variable-oriented heuristic $v\_dom/wdeg$ the overall winner. Results also show that as the instances becomes harder, the efficiency of $v\_dom/wdeg$ compared to dom increases. The variable-oriented v_wdeg heuristic in most cases is better than dom but is clearly less efficient than $v\_dom/wdeg$.

In Table 5.3 we present results from structured instances belonging to benchmark classes *langford* and *driver*. As the variable-oriented AC-3 variant is more efficient than the arc-oriented one, we only present results from the former. Results show that on easy problems all heuristics except $queue$ are quite competitive. But as the difficulty of the problem increases, the improvement offered by the $v\_dom/wdeg$ revision heuristic becomes clear. On instance driverlogw-09 we can see the effect that weight based revision ordering heuristics can have on search. $v\_dom/wdeg$ cuts down the number of node visits by more than 5 times resulting in a similar speed-up. It is interesting that $v\_dom/wdeg$ is considerably more efficient than v_wdeg and dom, indicating that information about domain size or weighted degree alone is not sufficient to efficiently order the revision list.

Finally, in Table 5.4 we present results from random and quasi-random problems. In the geo50-20-d4-75-2 , which is a quasi-random instance we can see that the proposed heuristics (v_wdeg and $v\_dom/wdeg$) are one order of magnitude faster than dom. This suggest that the small presence of structure is this problem results in behavior similar to the behavior observed in the structured instances of Table 5.3.

On the rest of the instances, which are purely random, there is a large

Table 5.1: *Cpu times (t), constraint checks (c), number of list revisions (r) and nodes (n) from frequency allocation problems (hard instances) using arc oriented propagation. The s prefix stands for scen instances. Best cpu time is in bold.*

		ARC ORIENTED					
Inst.		*queue*	*dom*	*a_wcon*	*a_wdeg*	*a_dom/wdeg*	*a_dom/wcon*
s11-f9	t	18,8	**12,8**	14,6	14,8	19	14,2
	c	25,03M	19,3M	13,2M	20,8M	21M	16,8M
	r	1,1M	910060	529228	1,04M	1,01M	737803
	n	1202	1153	1155	1145	1148	1159
s11-f8	t	37,5	**20,3**	22,5	21,9	28,5	23,5
	c	46,5M	29,3M	19,1M	30,1M	32,9M	27,5M
	r	1,95M	1,3M	748050	1,52M	1,43M	1,11M
	n	1982	1830	1843	1876	1832	1928
s11-f7	t	257,5	**146,5**	170	265,2	205,8	326,2
	c	268,4M	159,4M	128,5M	281,4M	205,1M	300M
	r	13,3M	10,2M	6,1M	17,7M	12,1M	15M
	n	17643	14734	15938	20617	15318	29845
s11-f6	t	568,5	465,2	**309,4**	540,4	834,9	396,4
	c	482,3M	468,2M	230,8M	517,2M	745,4M	362,7M
	r	27,5M	29,7M	10,4M	34,9M	49,5M	16,6M
	n	46671	50021	29057	49201	68217	35860
s11-f5	t	2821	2307	3064	3234	2898	**2291**
	c	2,492G	2,139G	2,097G	2,928G	2,596G	1,965G
	r	137,8M	157M	116,5M	215,7M	172,2M	103,3M
	n	212012	217407	287017	258261	185991	187363
s11-f4	t	11216	**7774**	8256	10386	12520	10473
	c	9,938G	7,054G	5,298G	9,020G	10,711G	8,598G
	r	533,4M	523,1M	311,7M	681,2M	738,1M	464,7M
	n	753592	709196	762477	832892	850446	786924

variance in the results. All heuristics seems to lack robustness and there is no clear winner. The constraint weight based heuristics can be faster than dom (instance frb30-15-1), but they can also be significantly slower (frb30-15-2). In all cases, the large run time differences in favor of one or another heuristic are caused by corresponding differences in the size of the explored search tree, as node visits clearly demonstrate.

A plausible explanation for the diversity in the performance of the heuristics on pure random problems as opposed to structured ones is the following. When dealing with structured problems, and assuming we use the variable-oriented variant of AC-3, a weighted based heuristic like $v\_dom/wdeg$ will give priority for revision to variables that are involved in hard subproblems and hence will carry out DWO-revisions faster. This

Table 5.2: *Cpu times (t), constraint checks (c), number of list revisions (r) and nodes (n) from frequency allocation problems (hard instances) using variable oriented propagation. The s prefix stands for scen instances. Best cpu time is in bold.*

		VARIABLE ORIENTED			
Inst.		*queue*	*dom*	*v_wdeg*	*v_dom/wdeg*
s11-f9	t	14,3	10,2	10,9	**9,9**
	c	22,6M	11,4M	12,9M	11M
	r	28978	17177	20161	17048
	n	1413	1117	1145	1137
s11-f8	t	21,2	17,3	18,5	**16,7**
	c	42,1M	17,2M	20,4M	16,8M
	r	48568	24885	28807	24819
	n	2112	1842	1830	1841
s11-f7	t	133,7	158,1	154,5	**108,2**
	c	193,3M	116,9M	157,6M	82,7M
	r	313568	223094	263306	156160
	n	12777	18773	14570	13181
s11-f6	t	391,4	391	434,4	**269,5**
	c	306,2M	263,2M	413,6M	192,6M
	r	426469	509474	673935	340583
	n	34714	46713	41609	31538
s11-f5	t	2473	3255	2019	**1733**
	c	2,073G	2,115G	1,502G	1,157G
	r	3,63M	4,52M	2,97M	2,2M
	n	223965	397590	190496	199854
s11-f4	t	13969	11859	9490	**6669**
	c	12,059G	7,512G	6,915G	4,322G
	r	20,3M	15,9M	14M	8,9M
	n	1,148M	1,354M	939094	716427

will in turn increase the weights of constraints that are involved in such hard subproblems and thus search will focus on the most important parts of the search space. Pure random instances that lack structure do not in general consist of hard local subproblems. Thus, different decisions on which variables to revise first can lead to different DWO-revisions being discovered, which in turn can guide search tree to different parts of the search space with unpredictable results. Note that for structured problems only very few possible DWO-revisions are present in the revision list at each point in time, while for random ones there can be a large number of such revisions.

Table 5.3: *Cpu times (t), constraint checks (c), number of list revisions (r) and nodes (n) from structured problems using variable oriented propagation. Best cpu time is in bold.*

Instance		$queue$	dom	v_wdeg	$v\_dom/wdeg$
langford-2-9	t	56,5	46,9	60,3	**46,2**
	c	99,6M	81,7M	99,9M	81,5M
	r	633113	533656	741596	533261
	n	48533	40228	49114	40363
langford-2-10	t	489,8	430,6	418,9	**340,1**
	c	336,1M	283,7M	275,2M	197,9M
	r	5,3M	4,5M	4M	2,9M
	n	337772	280600	260343	208651
langford-3-11	t	695,8	648,5	843,5	**513,5**
	c	408,6M	352,7M	468,8M	256,7M
	r	2,3M	1,9M	2,9M	1,6M
	n	99508	68042	103863	65958
langford-4-10	t	81,4	57,7	99,4	**41,2**
	c	52,3M	33,2M	59,6M	21,7M
	r	150493	99646	194952	75889
	n	3852	2973	5759	2661
driverlogw-08c	t	19,4	14,7	**14,4**	14,6
	c	20,8M	8,6M	10,9M	9M
	r	86809	39063	40256	38748
	n	3151	3040	1960	2660
driverlogw-09	t	174,6	411	346,3	**70,1**
	c	151,5M	251,5M	203,6M	39,5M
	r	521358	1,05M	583686	139962
	n	21220	41039	31548	7457

5.5 Dependency of conflict-driven heuristics on the revision ordering

As we showed in the previous section, *dom/wdeg* is strongly dependent on the order in which the revision list is constructed and updated during constraint propagation. Looking at the results in Tables 5.1 – 5.4, we can see that there are cases where the differences in cpu performance between dom and $v\_dom/wdeg$ can be up to 5 times. Hence, when *dom/wdeg* is used as DVO heuristic, we must carefully select a good revision ordering using for example one of the heuristics we have proposed in Section 5.3. In contrast, the conflict-driven DVO heuristics "*alldel*" and "*fully assigned*" are not as amenable to the selection of revision ordering. To better illustrate this statement, let us consider the following example.

Table 5.4: *Cpu times (t), constraint checks (c), number of list revisions (r) and nodes (n) from random problems using variable oriented propagation. Best cpu time is in bold.*

Instance		$queue$	dom	v_wdeg	$v\_dom/wdeg$
frb30-15-1	t	22,3	20,9	29,3	**14,1**
	c	16,5M	11,1M	16,4M	7,5M
	r	105626	70924	102724	46727
	n	3863	3858	4138	2499
frb30-15-2	t	84,9	**29,7**	118,9	95
	c	45,7M	21,8M	90M	68,9M
	r	311040	149119	624360	472124
	n	15457	7935	25148	24467
frb35-17-1	t	125,8	193,7	**118**	250,9
	c	93,9M	144M	89,7M	180,9M
	r	533694	836462	514258	1,03M
	n	18587	40698	19167	50611
rand-2-30-15	t	1240	**74,4**	98	108,1
	c	114,5M	53M	72,5M	78,1M
	r	922251	443792	602582	642665
	n	28725	19846	20192	28766
geo50-20-d4-75-2	t	226,1	401,8	**34,8**	39,5
	c	191,8M	310,3M	28,2M	28,8M
	r	778758	1,3M	117241	124163
	n	20069	60182	3735	5484

Example 7 *Assume that we want to solve a CSP (X, D, C) with X: $\{x_1, x_2, x_3, x_4\}$, by using two different revision ordering heuristics R_1 (lexicographic ordering) and R_2 (reverse lexicographic ordering). For the revision of each $x_i \in X$, we assume that the following hypotheses are true:* a*) The revision of x_1 is fruitful and it causes the addition of the variable x_3 in the revision list.* b*) The revision of x_2 is fruitful and it causes the addition of the variable x_4 in the revision list.* c*) The revision of x_4 is fruitful and it causes the addition of the variable x_3. We also assume the a DWO can only occur either* d*) in x_4 after a sequential revision of x_2 and x_3 or* e*) in x_3 after a sequential revision of x_4 and x_1. Finally, assume that at some point during search only the variables x_1 and x_2 have remained in the AC revision list Q, but with different orderings for R_1 and R_2. That is, Q_{R_1}:$\{x_1,x_2\}$, Q_{R_2}:$\{x_2,x_1\}$. Following all these assumptions (which can exist commonly in any real world CSP), lets now trace the behavior of both R_1 and R_2 during problem solving. Considering the Q_{R_1} list, the revision of x_1 is fruitful and adds x_3 in the list (due to hypothesis a). Now the revision list changes to Q_{R_1}:$\{x_2,x_3\}$. The sequential revision of x_2 and x_3 leads to the DWO of x_4 (due to hypotheses b and d).*

Considering the Q_{R_2} list, the revision of x_2 is fruitful and adds x_4 in the list (due to hypothesis b). Now the revision list changes to Q_{R_2}:{x_4,x_1}. The sequential revision of x_4 and x_1 leads to the DWO of x_3. (due to hypotheses c and e).

From the above example it is clear that although only one DWO is identified in the revision list, both x_1 and x_2 can be responsible for this. In R_1 where x_1 is the DWO variable, we can say that x_2 is also a "potential" DWO variable i.e. it would be a DWO variable, if the R_2 revision ordering was used. Although the *dom/wdeg* heuristic ignores all the "potential" DWO variables, the other two DVO heuristics,"*alldel*" and "*fully assigned*", take into account their contribution. The former heuristic increases the weights for every constraint that causes a value deletion, and thus succeeds to increase the weights of the constraints related to the "potential" DWO variables. The latter heuristic increases the weights of constraints that participate in fruitful revisions (only for revision lists that lead to a DWO), and thus is able to frequently identify "potential" DWO variables.

To experimentally verify the strong dependance of $dom/wdeg$ heuristic on the revision ordering and the ability of the "*alldel*" and "*fully assigned*" heuristics to be less dependent, we have computed the variance in the number of node visits for the three conflict-driven heuristics on some selected instances.

The variance is a measure of how spread out a distribution of a variable's values is. A variable's spread is the degree to which the values of the variable differ from each other. If all values of the variable were about equal, the variable would have very little spread. In other words, it is a measure of variability. In our case the measured variable x is the number of visited nodes for the conflict-driven heuristics. For each conflict-driven heuristic the x variable can take N=3 values. That is, the number of visited nodes when any one of the 3 main revision ordering heuristics ($queue$, dom, $v\_dom/wdeg$) is used.

The variance is calculated by taking the arithmetic mean of the squared differences between each value and the mean value, using the following equation:

$$VARIANCE = \frac{\sum(x - \bar{x})^2}{N} \tag{5.1}$$

where x is the number of node visits when a specific revision ordering heuristic is used and $\bar{x}$ is the mean number of visited nodes of the N=3 main revision ordering heuristics ($queue$, dom, $v\_dom/wdeg$).

The smaller the variance of a conflict-driven heuristic, the less the dependance from the selected revision ordering heuristic. Results from these experiments are depicted in Table 5.5. As we can see, in almost all cases the $dom/wdeg$ heuristic displays the highest variance, while the other two conflict-driven heuristics in most cases have smaller values. This suggests that indeed the "*alldel*" and "*fully assigned*" heuristics are less amenable to changes in the revision ordering than *dom/wdeg* and therefore can be more robust.

Table 5.5: *The computed variances for the three conflict-driven heuristics. Best values is in bold.*

Instance	$dom/wdeg$	$alldel$	$fully\ assigned$
scen-11	96732	7432	**67**
scen-11-f8	6893	2127	**701**
scen-11-f7	3974589	6384509	**1454538**
jnh01	6123	**80**	41280
jnh17	1316	**52**	91
jnh201	4238	12	**7**
jnh301	66738	19783	**91**
langford-2-10	7564932	**4547893**	10923451
driverlogw-08c	291287	8465	**912**
driverlogw-09	71643951	19821345	**13189345**
will199GPIA-5	1139	**0**	3717
will199GPIA-6	5313746	860138	**614930**

Finally, it would be interesting to apply similar ideas as the ones presented in this chapter to propagator-oriented solvers. Constraint propagation in such solvers is not handled by a revision list of variables or constraints, but they do use heuristics to choose the order in which propagators will be applied [75]. Hence exploiting information such as constraint weights might be beneficial.

We are more ready to try the untried when what we do is inconsequential. Hence the fact that many inventions had their birth as toys.

E. Hoffer

6

Adaptive Branching for CSPs

In this chapter we are interested in developing an adaptive branching scheme for CSPs. In order to devise such a scheme, we first have to explore and analyze the practical differences of the standard branching schemes. The most widely used standard branching schemes for CSPs are d-way and 2-way branching. Although it has been shown that in theory the latter can be exponentially more effective than the former, there is a lack of empirical evidence showing such differences. To investigate this, we initially make an experimental comparison of the two branching schemes over a wide range of benchmarks and under a variety of variable ordering heuristics. Experimental results verify the theoretical gap between d-way and 2-way branching as we move from a simple variable ordering heuristic like *smallest domain* to more sophisticated ones like *dom/ddeg*. However, exponential differences are rarely observed when state-of-the-art variable ordering heuristics like *dom/wdeg* and *impact* are used. Perhaps surprisingly, experiments also demonstrate that under such heuristics d-way branching can be clearly more efficient than 2-way in many cases. Exploiting this observation, we propose two generic heuristics that can be applied at certain points during search to decide whether 2-way branching or a restricted version of 2-way branching, which is close to d-way branching, will be followed. The application of these heuristics results in an adaptive branching scheme. Experiments with instantiations of the two generic heuristics confirm that search with adaptive branching outperforms search with a fixed branching scheme on a wide range of problems.

6.1 Introduction

Branching decisions repeatedly split the search tree into two or more subtrees. Classical branching schemes for making such decisions are 2-way (or binary) branching and d-way branching (or enumeration). In d-way branching, after a variable x with domain $\{a_1, \ldots, a_d\}$ is chosen, d branches are built, each one corresponding to one of the d possible value assignments of x. In 2-way branching, after a variable x is chosen, its values are assigned through a sequence of binary choices. The first choice point creates two branches, corresponding to the assignment of a_1 to x (left branch) and the removal of a_1 from the domain of x (right branch). The full version of 2-way branching allows for any variable to be chosen after a value removal, while a commonly used restricted version requires branching on variable x again. 2-way branching was described by Freuder and Sabin within the MAC algorithm [73] and in theory it can achieve exponential savings in search effort compared to d-way branching [47]. Indeed, 2-way is the standard branching scheme of most constraint solvers.

Since we are interested in developing adaptive branching schemes, we first need to identify and analyze the practical differences of the standard branching schemes described above. Identifying cases where each standard branching scheme, like 2-way or d-way, work efficiently, can give us indications on how we can adapt them.

In order to achieve this analysis, we first make a detailed experimental comparison between 2-way branching, in both its restricted and full versions, and d-way branching, under a variety of different variable ordering heuristics (VOHs). Results show that, unsurprisingly, the d-way and restricted 2-way branching schemes are closely matched across the different VOHs, with d-way being slightly more cost effective. Also, confirming the theoretical results, exponential differences in favor of full 2-way branching are observed as soon as we move from a simple heuristic like smallest domain (*dom*) to more sophisticated ones like domain over dynamic degree (*dom/ddeg*). But perhaps surprisingly, when state-of-the-art heuristics like *dom/wdeg* and *impact* are used, significant differences in favor of d-way (and restricted 2-way) are also observed. We conjecture that this is because

in some cases the VOH mistakenly chooses to branch on a variable other than the current one after the successful propagation of a value removal. This can divert search away from a hard part of the search space, resulting in increased search effort.

Exploiting and analyzing this observation, we next propose two generic heuristics that can be applied at successful right branches once the VOH chooses to branch on a variable other than the current one. At this point the heuristics are used to decide whether the advice of the VOH will be followed or not. The application of these heuristics results in an adaptive branching scheme that dynamically switches between 2-way branching and its restricted version (which is close to d-way branching). Both of our heuristics can be used in tandem with any backtracking search algorithm and VOH. The first heuristic is based on measuring the difference between the scores that the VOH assigns to its selected variable and the current variable. The VOH is followed only if the difference is sufficiently large. As a downside, this heuristic requires some tuning to optimize its performance. The second heuristic is based on the use of a secondary advisor to decide if the VOH will be followed, and it does not require any tuning.

Experiments with instantiations of the two generic heuristics confirm that search with adaptive branching outperforms search with a fixed branching scheme on a wide range of problems. This is more profound when a conflict-directed VOH like *dom/wdeg* is used, but it also notable under the *impact* VOH. Interestingly, in many cases where full 2-way branching significantly outperforms d-way and restricted 2-way, the adaptive branching methods match its performance with only very few decisions following the VOH when it suggests to move away from the current variable at successful right branches.

6.2 Branching schemes

From the early days of CSP research, search algorithms were usually implemented using either a d-way or a 2-way branching scheme. The former works as follows. After a variable x with domain $D(x) = \{a_1, a_2, ..., a_d\}$

is selected, d branches are created, each one corresponding to a value assignment of x. In the first branch, value a_1 is assigned to x and constraint propagation is triggered. If this branch fails, a_1 is removed from $D(x)$. Then the assignment of a_2 to x is made (second branch), and so on. If all d branches fail then the algorithm backtracks. An example of a search tree explored with d-way branching is shown in Figure 6.1a. This type of branching was typically used in the past to describe older search algorithms like chronological backtracking, forward checking, and various versions of backjumping [44, 68, 51], but it is still used in some solvers, mainly ones developed in academia.

In 2-way or *binary* branching, after a variable x and a value $a_i \in D(x)$ are selected, two branches are created. In the left branch a_i is assigned to x, or in other words the constraint x=a_i is added to the problem and is propagated. In the right branch the constraint $x \neq a_i$ is added to the problem and is propagated. If both branches fail then the algorithm backtracks. Figure 6.1b shows a search tree explored with 2-way branching. This type of branching is typical in constraint programming systems and is the most widely used branching scheme in commercial and other well-developed solvers (e.g. [48, 81]). Since the description of the MAC algorithm with 2-way branching [73], it has gradually taken over from d-way in academia as well.

There are two differences between these branching schemes:

- In 2-way branching, if the branch assigning a value a_i to a variable x fails then the removal of a_i from $D(x)$ is immediately propagated. Instead, d-way branching tries the next available value a_j of $D(x)$. Note that the propagation of a_j subsumes the propagation of a_i's removal.

- In 2-way branching, after a failed branch corresponding to an assignment x=a_i, and assuming the removal of a_i from $D(x)$ is then propagated successfully, the algorithm can choose to branch on any variable (not necessarily x), according to the VOH (e.g. Figure 6.1b). In d-way branching the algorithm has to choose the next available value for variable x after x=a_i fails.

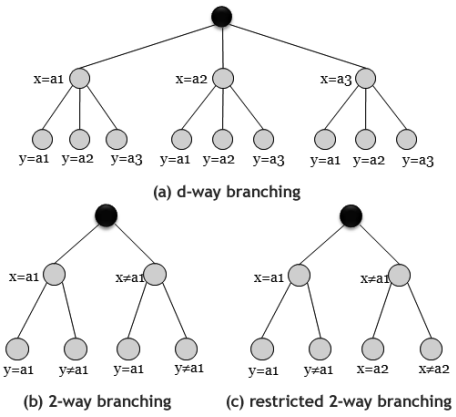

Figure 6.1: *Examples of search trees for the three branching schemes.*

In between these two schemes is the version of 2-way branching used in [78] where the algorithm is forced to assign x with its next value after the successful propagation of a_i's removal from $D(x)$. In the following we call this *restricted 2-way* branching. Figure 6.1c shows a search tree explored with restricted 2-way branching.

We should note that alternative branching schemes have been proposed but are by far less popular than 2-way and d-way. The most notable among them is dichotomic domain splitting [31]. This method originates from numerical CSPs and proceeds by splitting the current domain of the selected variable into two sets, usually based on the lexicographical ordering of the values.

All the above branching schemes essentially post unary constraints at each decision point (e.g. $x = a_i$, $x \neq a_i$, $x > a_i$). A survey of these and other less widely used branching schemes that post unary constraints is given in [2]. Alternatively, there are branching schemes that post non-unary constraints, but these are rather problem-specific and although they are successful on certain problems (e.g. in scheduling), they typically need to be hand-crafted into a solver as the default method offered is 2-way branching.

6.3 Comparing 2-way to d-way Branching

Although there is an implicit assumption in the community that 2-way branching is more efficient in practice than d-way (as noted above, 2-way is the dominant branching scheme in CP solvers), there are only a few experimental studies on this topic in the literature. Despite this assumption and the theoretical result of [47], the few experimental studies comparing 2-way and d-way branching have not displayed significant differences between them. Park showed that 2-way and d-way display very similar performance when the *dom* VOH is used [65], while Smith and Sturdy showed that 2-way outperforms d-way when searching for all solutions, albeit not considerably (an average speed-up of 30% was reported) [78]. However, it is important to note that both research works used restricted 2-way branching in their experimental study.

The lack of extensive experimental evidence on the relative performance of (full) 2-way branching compared to d-way motivated us to empirically evaluate them on a wide range of random and structured benchmarks. The aim of this study was threefold:

1. To compare the three branching schemes (2-way, restricted 2-way, d-way) on a wide range of problems using a standard search algorithm and variable ordering heuristic.

2. To investigate if and to what extent the choice of variable ordering heuristic affects the relative performance of the branching schemes.

3. To investigate if and to what extent the level of local consistency applied during search affects the relative performance of the branching schemes.

To address these three goals we compared three implementations of the MAC algorithm, one for each branching scheme, using a variety of VOHs: *dom*, *dom/ddeg*, *dom/wdeg*, *dom/wdeg + aging*, *impact*. We also compared three corresponding implementations of the MmaxRPC algorithm using the *dom/wdeg* heuristic.

We have experimented with 1600 instances taken from C. Lecoutre's web repository. We have tried to include a wide range of CSP instances from different backgrounds but our focus was mainly on binary problems. Hence, we have experimented with instances from real world applications (like the Radio Link Frequency Assignment problem and the Driver problem), instances following a regular pattern and involving a random generation (Quasigroup Completion and Quasigroup With Holes problems, the Black Hole problem, the Graph Coloring problem and Haystacks), academic instances which do not involve any random generation (All-Interval series, chessboard coloration, Golomb Ruler, Langford, Queens, Queen Attacking, Queens-Knights and Domino), random instances containing a small structure (the Geometric problem) and, finally, pure random instances (generated following models D and RB). Although the majority of the selected problems are binary, we have also experimented with the following non-binary academic problem classes: All-Interval series, Chessboard Coloration and Golomb Ruler.

As a primary parameter for the measurement of the performance of the evaluated branching schemes, we have used the cpu time in seconds (t). We also report the number of visited nodes (n) as this gives a measure that is not affected by the particular implementation or by the hardware used. A node in 2-way branching can correspond to a value assignment or to a value removal, while in d-way branching it can only correspond to a value assignment. Hence, they cannot be compared directly. However, the number of nodes gives an accurate measure of the difference in search effort between 2-way and restricted 2-way.

Constraint propagation in our solver is variable-oriented and the revision list is implemented as a *fifo* list. That is, the oldest variable inserted in the list is always selected (i.e. the list is implemented as a queue).

The ordering of values for the selected variable is done lexicographically. Experiments with an informed value ordering heuristic, namely Geelen's promise [37], showed that the relative behavior of the tested branching schemes was qualitatively similar to that obtained under lexicographic value ordering. A time limit of 2 hours was set for every single experiment, unless otherwise stated. All the experiments were run on an Intel

Table 6.1: *CPU times(t) in seconds and nodes(n) for the three branching schemes using the VOH* dom.

Instance		$2-way$	$restricted$ $2-way$	$d-way$
cc-10-10-2	t	246.2	248.6	**241.9**
$(unsat)$	n	0.47M	0.47M	0.37M
cc-12-12-2	t	2,201	2,205	**2,152**
$(unsat)$	n	2.75M	2.75M	2.11M
queens-100	t	45.8	47	**40.6**
(sat)	n	16,018	16,111	13,140
queenA-5	t	124.1	124.4	**114.7**
(sat)	n	0.48M	0.48M	0.35M
queensK-10-5	t	**114.2**	127.9	131.5
$(unsat)$	n	85,207	85,401	81,348
bqwh18-141-0	t	45.1	43.7	**42.7**
(sat)	n	0.18M	0.18M	0.11M
langford3-11	t	161.6	156.2	**147.4**
$(unsat)$	n	96,775	88,479	41,042
langford4-10	t	18.1	17.7	**16.2**
$(unsat)$	n	3,544	3,207	1,481
domino300-300	t	9.4	**9.2**	9.4
$(unsat)$	n	300	300	300

dual core PC T4200 2GHz with 3GB RAM.

6.3.1 Using dom and dom/ddeg as VOHs

In a first set of experiments, we compared the performance of the three branching schemes when *dom* or *dom/ddeg* were used for variable ordering. The search algorithm for these experiments was MAC. These two VOHs are quite ineffective compared to more advanced ones like *dom/wdeg* and *impact*. Thus, only a fraction of the instances that terminated within the time limit using a state-of-the-art VOH, managed to terminate using *dom* or *dom/ddeg* (excluding very easy instances). Despite this, results from the instances that did terminate demonstrate a clear pattern with respect to the performance of the different branching schemes.

In Tables 6.1 and 6.2 we give indicative results from the use of the *dom* and *dom/ddeg* heuristics respectively. Under the *dom* VOH, 2-way branching essentially emulates d-way branching and the three branch-

Table 6.2: *CPU times(t) in seconds and nodes(n) for the three branching schemes using the VOH* dom/ddeg.

Instance		$2-way$	$restricted$ $2-way$	$d-way$
cc-10-10-2	t	**29.3**	144.8	143.6
$(unsat)$	n	30,706	0.14M	71,358
cc-12-12-2	t	**58.3**	978	959
$(unsat)$	n	36,994	0.56M	0.27M
scen-11	t	**7.1**	508	372
(sat)	n	1,379	68,597	45,862
scen-11-f11	t	**8.1**	$> 2h$	$> 2h$
$(unsat)$	n	2,716	-	-
driver-09	t	**205**	1197	1207
(sat)	n	75,033	0.42M	0.16M
ehi-85-297-0	t	**62.3**	2509	2481
$(unsat)$	n	37,246	2.5M	0.9M
ehi-85-297-2	t	**41.8**	$> 2h$	$> 2h$
$(unsat)$	n	20,733	-	-
ash958-4	t	**5.1**	$> 2h$	$> 2h$
(sat)	n	2712	-	-
ash313-7	t	**2,405**	$> 2h$	$> 2h$
(sat)	n	0.14M	-	-

ing schemes are closely matched in terms of run times (with d-way being slightly better). This is in accordance with previous results [65]. Concerning the *dom/ddeg* VOH, our results confirm the theoretical results of [47] as we can observe huge differences in favor of 2-way branching. Restricted 2-way and d-way display similar performance with d-way usually being slightly faster.

As an example that clearly depicts the differences in relative performance of the three branching schemes when the dom as opposed to the $dom/ddeg$ VOH is used, we can notice their behavior on the non-binary chessboard coloration instances cc-10-10-2 and cc-12-12-2. While in Table 6.1 the three branching schemes are very close, in Table 6.2, 2-way is over a magnitude faster than d-way and restricted 2-way.

6.3.2 Using conflict-driven VOHs

The dynamic VOHs of the previous subsection are nowadays considered relatively poor general purpose heuristics. Conflict-driven VOHs, like *dom/wdeg*, can learn from failures encountered during search and thus make more informed choices [18]. Such VOHs are considered to be state-of-the-art.

In this subsection we present results from the use of *dom/wdeg* in tandem with the three branching schemes. In addition, we also give results from a variant of *dom/wdeg* which applies periodic weight aging [8]. Regarding this variant, we have selected to periodically decrease all constraint weights by a factor of 2, with the period set to 20 backtracks.

In Table 6.3 we give indicative results from various structured instances. This table is divided in two parts. In the first part we include instances where d-way and restricted 2-way are the best choices, while in the second part we include instances where 2-way is better. The selected instances are among the ones that demonstrate the most notable differences in favor of either 2-way or d-way branching.

These results show that none of the branching schemes is always the best choice, even for instances within the same problem class (see for example the *ruler25* and *bqwh-18-141* problems in Table 6.3). Perhaps sur-

Table 6.3: *Cpu times (t), and nodes (n) from indicative instances when MAC is used with* dom/wdeg *and* dom/wdeg + aging*. Best cpu time is in bold.*

		dom/wdeg			dom/wdeg + aging		
Series of Instances		$2-way$	$restricted$ $2-way$	$d-way$	$2-way$	$restricted$ $2-way$	$d-way$
geo50-20-d4-75-62 (sat)	t	2,398	964	**883**	4,763	1,020	**904.9**
	n	0.43M	0.15M	0.12M	1.04M	0.25M	0.23M
geo50-20-d4-75-57 (unsat)	t	1,632	1,232	**1,083**	3,485	1,238	**1,126**
	n	0.18M	0.14	0.13	0.59M	0.23M	0.24M
haystacks-05 (unsat)	t	41.8	4	**3.7**	13	3.4	**3**
	n	1.27M	0.13M	0.11M	0.44M	0.10M	92,900
ruler-25-7-a3 (sat)	t	12.4	4.1	**2.1**	3.1	3.9	**2.1**
	n	1,444	225	129	258	228	129
qwh-15-106-1 (sat)	t	34.3	14.8	**14.3**	31	**1.4**	1.5
	n	44,150	19,964	12,788	35,542	2,104	1,742
bqwh-18-141-84 (sat)	t	97.1	59.1	**58.4**	86.5	51.8	**48.8**
	n	0.33M	0.18M	0.11M	0.22M	0.13M	0.11M
series13 (sat)	t	**148.4**	1,132	1,057	**839.9**	1,638	1,371
	n	0.15M	1.6M	1.33M	0.77M	1.94M	1.55M
scen2-f25 (unsat)	t	**42.3**	183.8	137.9	**38.4**	146.8	118.2
	n	7,819	31,516	42,699	8,879	34,704	42,459
scen1-f9 (unsat)	t	**5.7**	13.6	12.2	**4.3**	12.4	11.3
	n	1,300	3,582	2,808	1,341	4,156	3,538
ruler-25-8-a3 (unsat)	t	**50.3**	180.8	167.6	**39.7**	162.6	154.4
	n	1,829	6,407	7,026	1,579	6,300	6,963
qcp-15-120-6 (sat)	t	**62.8**	80.5	76.2	**96.9**	110.4	108.5
	n	70,384	92,058	60,314	0.11M	0.12M	0.10M
bqwh-18-141-95 (sat)	t	**1.6**	8.5	8.4	**2.7**	14.5	14
	n	4,417	25,957	15,984	6,075	38,509	30,797

prisingly, there are many instances where d-way (and restricted 2-way) are clearly better than 2-way branching. Moreover, in cases where 2-way branching dominates, exponential differences are rare. Actually, only in the case of the "All-Interval series" problem class did we observe persistent exponential differences in most of the instances included in the class. As in the case of *dom/ddeg*, restricted 2-way branching displays a behavior very close to that of d-way branching. Results from Table 6.3 also show that the relative performance of the three branching schemes does not change when we move from *dom/wdeg* to *dom/wdeg*+aging. Although the two heuristic display differences in performance (sometimes *dom/wdeg* is better and sometimes it is worse), the dominating branching scheme on each instance remains the same.

Comparing these results with the results of Table 6.2, we can notice two differences: First, there are many instances where 2-way branching is less efficient, sometimes considerably, compared to restricted 2-way and d-way. Second, in instances where 2-way branching dominates, the differences are not as striking as in Table 6.2, albeit they can still be considerable.

Aggregate results from all the binary structured problem classes that we experimented with are presented in Table 6.4. Similar results from non-binary problems are presented separately in Table 6.5. For each problem class we report the mean cpu time in seconds and the mean visited nodes. We have excluded instances that terminated in less than one second, and also instances which did not terminate within the time limit. In the last line of these tables we also give the total mean. This mean value has been computed over all the instances whose aggregate results are given in Table 6.4 and in Table 6.5.

Results showed that in almost half of the tried instances all methods were very close. In around 25% 2-way was faster than d-way, while in the remaining 25% it was slower. The last two cases included most of the hardest instances.

On binary problems, d-way branching is on average more efficient than 2-way. This superiority is observed in most of the problem classes. RLFAPs is the only case where 2-way dominates. The relative performance of the three branching schemes is not affected by the selection of conflict-

Table 6.4: *Mean cpu times (t), and nodes (n) from binary structured and patterned problems using MAC with* dom/wdeg *and* dom/wdeg + aging*. Best cpu time is in bold.*

		dom/wdeg			dom/wdeg + aging		
Series of Instances		$2-way$	$restricted$ $2-way$	$d-way$	$2-way$	$restricted$ $2-way$	$d-way$
geometric	t	537.6	440.7	**401.8**	1,221	478.7	**417.1**
	n	0.11M	96,624	86,986	0.26M	0.11M	0.10M
Driver	t	35.4	34.7	**28**	**9.4**	10.1	9.8
	n	16,176	17,393	8,111	2,292	2,532	1,736
rlfapScensMod	t	**8.5**	36.7	29	**7.4**	32.2	26.2
	n	3,505	15,932	18,470	2,623	11,634	12,835
rlfapGraphsMod	t	**31.7**	210.6	120.4	**13.4**	47.5	113.9
	n	10,396	0.12M	41,310	6,597	25,318	56,900
Black Hole-4-4	t	15	14.5	**13.7**	18.6	13	**12.1**
	n	9,892	9,762	8,661	15,483	13,384	12,283
Haystacks	t	20.9	2	**1.9**	6.5	1.7	**1.6**
	n	1.27M	0.13M	0.11M	0.22M	53,139	46,565
qwh-15-106	t	17	15.2	**15.1**	29.4	21.5	**21.1**
	n	21,585	20,959	12,689	31,054	23,825	18,984
qcp-10-67	t	135.3	131	**126.8**	**22.1**	25.2	24.5
	n	0.43M	0.42M	0.32M	68,511	84,729	69,075
qcp-15-120	t	380.4	382.3	**373.5**	**368.8**	393.6	385.3
	n	0.43M	0.45M	0.28M	0.38M	0.52M	0.40M
Langford2	t	55.7	51.4	**47.8**	117.6	56	**51.5**
	n	0.15M	0.14M	0.11M	0.20M	0.13M	0.11M
bqwh-15-106	t	2.05	1.63	**1.61**	2.6	2.3	**2.2**
	n	11,993	9,365	5,780	8,517	7,988	6,363
bqwh-18-141	t	13.5	11.9	**11.7**	15.5	14.1	**13.7**
	n	49,123	43,321	26,452	39,931	38,570	30,718
TOTAL MEAN	t	143.5	130.7	**117.6**	278.5	130.7	**119.8**

Table 6.5: *Mean cpu times (t), and nodes (n) from non-binary structured problems using MAC with* dom/wdeg *and* dom/wdeg + aging. *Best cpu time is in bold.*

		dom/wdeg			dom/wdeg + aging		
Series of Instances		$2-way$	$restricted$ $2-way$	$d-way$	$2-way$	$restricted$ $2-way$	$d-way$
allIntervalSeries	t	**574.7**	6,002	6,214	**2,849**	7,414	7,090
	n	0.42M	6.4M	6.2M	1.72M	6.3M	5.3M
golombRuler	t	319.7	**185.4**	315	**151.6**	719.9	348
	n	6,541	3,662	5,736	3,032	6,952	5,120
Chessboard Coloration	t	16.1	15.4	**15.3**	16.6	14.9	**14.8**
	n	9,501	9,342	6,069	12,904	12,119	9,304
TOTAL MEAN	t	**353.2**	2,540	2,673	**1,320**	3,538	2,270

driven VOH. Both *dom/wdeg* and the *aging* variation, gives qualitatively similar results.

Results for non-binary problems are quite different. In the case of the "All-Interval series" problem class we observe exponential differences in favor of 2-way. On the contrary, on the "chessboard coloration" problem d-way is slightly better than 2-way. Golomb ruler is the only problem class where the selection of the conflict-driven VOH affects the relative performance of the branching schemes. When *dom/wdeg* is used, 2-way is the worst choice, while when *dom/wdeg*+aging is used, 2-way is the best choice. Also, and uncharacteristically, there is considerable variance between the performance of restricted 2-way and d-way. The computed total mean is clearly affected by the huge differences in favor of 2-way in the "All-Interval series" problem class.

In general we can say that our empirical observations outlined in this subsection show that with conflict-driven variable ordering heuristics exponential differences in favor of 2-way branching are rare. Also, perhaps surprisingly, in a wide range of binary problems d-way branching significantly outperforms 2-way. This is the first study that gives detailed empirical evidence in favor of d-way.

6.3.3 Using the impact VOH

In order to investigate if the choice of an efficient general purpose VOH that is based on different principles than conflict-driven VOHs affects the relative performance of the branching schemes, we ran experiments with the *impact* VOH.

In these experiments we have approximated the values at the initialization phase of impacts by dividing the domains of the variables into (at maximum) four sub-domains. Since the initialization cost of impacts is rather significant (especially on instances with large domains), the fraction of problems that terminated within the time limit was smaller that that of the previous subsection.

In Table 6.6 we give results from indicative instances. In the first part of this table we again include instances where d-way branching is the dominant branching scheme, while in the second part we include instances where 2-way is better. As with the conflict driven VOHs, neither 2-way nor d-way is always the best choice, even for instances within the same problem class (see for example *qwh-15* and *bqwh-15*).

Although there are many instances where d-way is faster than 2-way, mean results for binary problems in Table 6.7 and for non-binary problems in Table 6.8 show that in general 2-way is a better choice.

Comparing the results obtained with *impact* and those obtained with conflict-driven VOHs, we may notice a difference in the relative performance of the branching schemes on binary problems. While with the conflict-driven VOHs d-way was faster than 2-way, with *impact* 2-way was better. But again the experimental behavior of the different branching schemes using any of these state-of-the-art heuristics does not reveal exponential differences between 2-way and d-way branching, such as the ones displayed using *dom/ddeg*.

6.3.4 Maintaining a Stronger Level of Consistency

We now investigate the behavior of the three branching schemes when a stronger level of consistency than AC is maintained during search, namely when maxRPC is maintained. To achieve maxRPC we use the recently

Table 6.6: *Cpu times (t), and nodes (n) from indicative instances when MAC is used with* impact*. Best cpu time is in bold.*

Series of Instances		$2-way$	$restricted$ $2-way$	$d-way$
ruler-34-9-a3	t	1,665	874.2	**746.6**
(unsat)	n	37,224	16,972	17,708
langford-4-10	t	82.1	54.6	**47.3**
(unsat)	n	5,906	4,282	3,997
frb30-15-5-mgd	t	66.4	4.4	**3.4**
(sat)	n	32,627	2,243	1,773
qwh-15-106-4	t	19.1	2.8	**1.4**
(sat)	n	59,478	1,904	754
bqwh-15-106-53	t	23.1	2.37	**2.35**
(sat)	n	0.15M	5,904	5,016
bqwh-15-106-93	t	9.03	1.06	**1.03**
(sat)	n	35,676	1,778	1,068
series13	t	**211.2**	2,710	3,004
(sat)	n	0.30M	4.28M	3.95M
qwh-15-106-1	t	**191.8**	982	508.5
(sat)	n	0.55M	4.75M	1.53M
qwh-15-106-5	t	**54.5**	467.2	141
(sat)	n	0.18M	2.04M	0.36M
bqwh-15-106-3	t	**110.57**	404.3	640.4
(sat)	n	0.67M	2.76M	3.34M
ruler-34-8-a3	t	**68.9**	87.1	238.3
(sat)	n	1,342	1,943	10,174
frb30-15-1	t	**71.5**	385.6	387
(sat)	n	36,042	0.20M	0.25M

Table 6.7: *Mean cpu times (t), and nodes (n) from binary structured and random problems using MAC with* impact*. Best cpu time is in bold.*

Series of Instances		$2-way$	$restricted$ $2-way$	$d-way$
Driver	t	**22.9**	24.2	23.5
	n	882	1,462	1,154
Haystacks	t	54.7	56.9	**47.9**
	n	0.96M	1.09M	0.89M
qwh-15-106	t	**51.3**	183.6	107.2
	n	0.16M	0.82M	0.30M
Langford	t	**148.2**	185.9	157.2
	n	0.10M	0.15M	94,546
bqwh-15-106	t	**9.2**	22.2	18.4
	n	49,449	0.13M	82,418
frb30-15	t	**77.1**	133	100.4
	n	34,591	69,368	66,336
TOTAL MEAN	t	**28.4**	57.5	42.4

Table 6.8: *Mean cpu times (t), and nodes (n) from non-binary structured problems using MAC with* impact *as VOH. Best cpu time is in bold.*

Series of Instances		$2-way$	$restricted$ $2-way$	$d-way$
allIntervalSeries	t	**79.6**	1,040	1,072
	n	0.11M	1.66M	1.46M
golombRuler	t	**863.3**	898.7	994.6
	n	14,894	12,575	18,370
Chessboard Coloration	t	73.5	65.2	**62.4**
	n	95,383	86,788	65,078
TOTAL MEAN	t	**500.1**	738.9	797.5

proposed algorithm maxRPC3^{rm} which is the most efficient maxRPC algorithm to date [3]. The success of this algorithm is based on the exploitation of residual supports, a concept first introduced in the efficient AC algorithms AC^r and AC^{rm} [55, 57]. To place the presented results in context, each instance that was solved with MmaxRPC was also solved with MAC, but this time the AC algorithm used was AC^{rm} (so that a direct comparison with maxRPC3^{rm} can be made). As *dom/wdeg* is perhaps the most successful general purpose VOH, it has been used in all experiments presented here.

In the first set of experiments we have experimented with seven series of random problems, each one consisting of 100 binary instances at the phase transition and generated following Model D. For each class $< n, d, e, t >$, the number of variables n has been set to 40, the domain size d varied between 8 and 180, the number of constraints e between 84 and 753 (meaning that the constraint graph density varied between 0.1 and 0.96) and the tightness t, which denotes the probability that a pair of values is allowed by a relation, varied between 0.1 and 0.9. The first class $< 40, 8, 753, 0.1 >$ corresponds to dense instances involving constraints of low tightness whereas the seventh one $< 40, 180, 84, 0.9 >$ corresponds to sparse instances involving constraints of high tightness.

Figure 6.2 shows the effort (cpu time) required for solving these seven series of random instances with respect to the different tightness values. We compare here the 2-way and d-way branching schemes. When MAC is used, Figure 6.2(a), d-way branching is clearly more cost effective than 2-way branching. When MmaxRPC is used, Figure 6.2(b), again d-way is better but the differences in its favor are smaller, and in instances with high tightness 2-way branching is slightly faster.

In all seven series the d-way and restricted 2-way branching schemes displayed similar performance, with d-way being slightly more cost effective. In Figure 6.3(a) we depict the search effort for tightness=0.65 when MAC was used. The 100 random instances from that set were shorted in ascending order according to the cpu time. In Figure 6.3 (b) we give similar plots for tightness values 0.1, 0.2 and 0.35 when MmaxRPC is used.

In a second set of experiments we evaluated the efficiency of the three

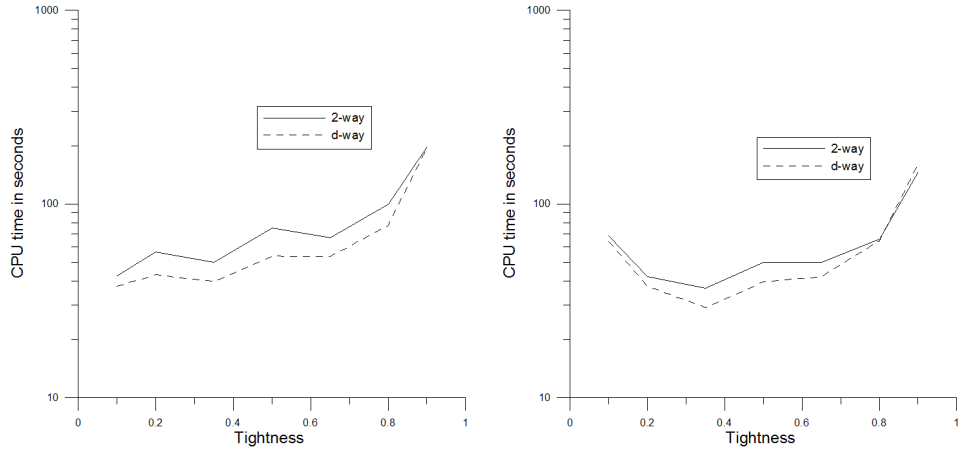

Figure 6.2: *Mean search cost of solving the random instances as tightness is increased. 2-way and d-way branching are comparatively depicted. In (a) with MAC and in (b) with MmaxRPC.*

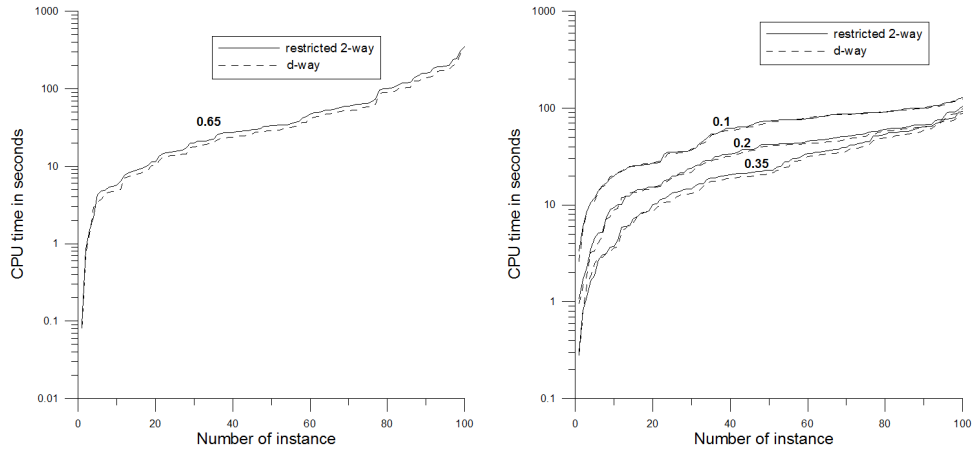

Figure 6.3: *Comparison of restricted 2-way and d-way branching. (a) Solving time for the 100 random instance with tightness = 0.65, shorted in ascending order when MAC is used. (b) Solving time for the 100 random instance with tightness = 0.1, 0.2 and 0.35 when MmaxRPC is used.*

Table 6.9: *Cpu times (t), and nodes (n) from indicative instances using MAC and MmaxRPC with dom/wdeg as VOH. Best cpu times are in bold.*

		MAC			MmaxRPC		
Series of Instances		$2-way$	$restricted$ $2-way$	$d-way$	$2-way$	$restricted$ $2-way$	$d-way$
geo50-20-d4-75-1 (sat)	t	278.8	160.1	**150.2**	306.3	181.6	**173.1**
	n	0.34M	0.20M	0.18M	0.12M	80,581	79,691
geo50-20-d4-75-2 (sat)	t	24.8	**4.1**	4.2	23	4	**3.7**
	n	41,133	6,610	6,239	13,436	2,181	1,826
bqwh-18-141-3 (sat)	t	89	62.4	**52.8**	11.2	9.7	**9**
	n	0.59M	0.40M	0.24M	87,386	74,743	48,549
haystacks-05 (unsat)	t	25.7	2.5	**2.2**	4	**0.3**	0.5
	n	1.27M	0.13M	0.11M	0.16M	10,363	20,278
qcp-15-120-10 (sat)	t	342.4	252	**223**	764.8	771.5	**688.1**
	n	1.01M	0.78M	0.48M	1.98M	2.03M	1.26M
qwh-15-106-1 (sat)	t	22.8	11.1	**10.1**	4.4	4.6	**4.1**
	n	44,150	19,964	12,788	11,991	12,727	8,507
geo50-20-d4-75-20 (sat)	t	**72.4**	188.1	175.8	**65.3**	237.6	200.6
	n	83.708	0.22M	0.19M	22,680	85,958	81,851
scen2-f25 (unsat)	t	**9.8**	41.4	40.6	**14**	54.5	40.8
	n	7,819	31,516	42,699	1,719	6,704	8,310
scen11-f8 (unsat)	t	**105.9**	805.1	722	**113.3**	965.4	748.4
	n	43,677	0.35M	0.34M	12,681	93,429	0.11M
scen11-f9 (unsat)	t	**66.9**	458.5	434.1	**99.1**	658.1	510.6
	n	27,920	0.20M	0.19M	10,265	62,465	69,400
graph9-f9 (sat)	t	**85.8**	323.6	430.3	**68.9**	265.3	101.2
	n	49,173	0.18M	0.28M	16,839	62,058	26,276
qwh-20-166-1 (sat)	t	**74.4**	182.5	151.9	**35.7**	53.7	40
	n	99,171	0.22M	0.13M	54,286	87,337	44,934

branching schemes over a wide range of structured problems. The same set of binary problems has been used for both the search algorithms.

Indicative results for both MAC and MmaxRPC are given in Table 6.9. In the first part of this table we have grouped instances where d-way is faster, while in the second part we have grouped instances where 2-way dominates. For both search algorithms the branching schemes have an analogue behavior. That is, in instances where d-way (resp. 2-way) is the best choice when MAC is used, d-way (resp. 2-way) is also the best choice when MmaxRPC is used.

Mean results per problem class for both MAC and MmaxRPC are col-

lected in Table 6.10. For the computation of the mean values per problem class we have excluded instances that terminated in less than one second and also instances which did not terminate within the time limit of 2 hours. In this set of experiments we have used only binary instances, as maxRPC is only defined for binary constraints, and we have experimented with some additional problem classes that were not used in the previous experiments. Although there is an analogue in performance between the search algorithms, we can notice that the difference among the branching schemes per problem class are smaller when MmaxRPC is used. The total mean computed over all the experimented instances shows that in the case of MAC d-way is slightly better than 2-way, while for MmaxRPC the opposite occurs.

Although it is not the focus of this study, it is very interesting to note that, as mean cpu times show, maxRPC is clearly a better choice of propagation method for binary CSPs regardless of the branching method chosen.

As a general conclusion from this set of experiments we can say that when we increase the level of local consistency maintained during search, 2-way branching becomes stronger. But there is a non negligible set of problem classes where d-way is still a better choice.

6.3.5 General discussion

Hwang and Mitchell have shown that in theory these exist instances which require exponential search trees for backtracking with d-way branching, but have polynomial search trees for backtracking with 2-way branching [47].

Our experimental study has shown that the appearance of such exponential differences in practice strongly depends on the VOH used. The choice of VOH can drastically affect the relative performance of the branching schemes if we consider the whole range of proposed heuristics. A less effective VOH like *dom/ddeg* typically results in exponential differences in favor of 2-way branching. But on the other hand modern VOHs like *dom/wdeg* and *impact* make exponential differences a rarity. State-of-the-

Table 6.10: *Mean cpu times (t), and nodes (n) from binary structured and patterned problems using MAC and MmaxRPC with dom/wdeg as VOH. Best cpu times are in bold.*

		MAC			MmaxRPC		
Series of Instances		$2-way$	$restricted$ $2-way$	$d-way$	$2-way$	$restricted$ $2-way$	$d-way$
geometric	t	104.5	84.5	**75.3**	99.5	92.5	**84.4**
	n	0.11M	96,624	86,986	35,974	34,590	34,260
bqwh-15-106	t	1.46	1.12	**1.03**	0.47	0.43	**0.29**
	n	11,993	9,365	5,780	4,987	4,623	2,110
bqwh-18-141	t	8.16	7.2	**6.1**	3.03	2.95	**2.55**
	n	49,123	43,321	26,452	21,456	21,957	13,417
rlfapScensMod	t	**3.7**	15.6	14.6	**7.3**	18.3	14.3
	n	3,505	15,932	18,470	1,067	2,527	2,708
rlfapGraphsMod	t	**17.1**	115.1	63.2	**14.9**	45.8	19.3
	n	10,396	0.12M	41,310	4,006	11,647	5,287
rlfapScens11	t	**184.1**	1,375	1,283	**93.7**	778.4	552.4
	n	13,174	97,151	95,213	14,664	0.11M	0.1M
qcp-10-67	t	73.8	77.1	**66**	11.4	10.6	**7**
	n	0.43M	0.42M	0.32M	78,989	76,569	42,847
qcp-15-120	t	175.4	189.7	**164.5**	62.9	**54.5**	56.6
	n	0.43M	0.45M	0.28M	0.18M	0.16M	0.12M
qwh-15-106	t	10.8	10.2	**9**	3.7	3.5	**2.8**
	n	21,585	20,959	12,689	10,686	10,063	5,928
qwh-20-166	t	1,701	1,157	**863.8**	219	200.1	**193.9**
	n	1.97M	1.35M	0.77M	0.31M	0.29M	0.2M
driverlogw	t	19.5	25	**15.8**	**13.2**	18.2	20.8
	n	16,176	17,393	8,111	11,255	14,530	9,477
blackHole-4-4	t	2.9	2.8	**2.6**	3.5	3.5	**3.4**
	n	9,892	9,762	8,661	4,741	4,407	4,371
haystacks	t	25.7	2.5	**2.2**	4	**0.3**	0.5
	n	1.27M	0.13M	0.11M	0.16M	10,363	20,278
TOTAL MEAN	t	132	149.2	**124.4**	**53.5**	84.3	69.5

art VOHs like *dom/wdeg* and *impact* are adaptive heuristics, which means that they have the ability to learn during search and subsequently make better decisions. This learning ability seems to alleviate the differences in efficiency between the branching schemes. Perhaps surprisingly, our experiments have also shown that on binary CSPs when the *dom/wdeg* VOH is used, d-way branching is a better choice in general than 2-way.

With respect to the level of local consistency applied, our experiments with maxRPC have shown that increasing the level of consistency does have an effect on the relative performance of the branching schemes, albeit not an overwhelming one. Specifically, 2-way branching becomes stronger when moving from AC to maxRPC. But there still exists a non negligible number of problems where d-way remains a better choice.

A general likely explanation for the failure of 2-way branching on some instances, compared to its restricted version and d-way, is the following. At some right branches during search, the VOH mistakenly chooses to branch on a variable other than the current one. In the case of conflict-driven VOHs this may result in the search process moving away from a hard subproblem to another area of the search space resulting in increased search effort. To test this conjecture we have developed heuristics, presented below, that can be applied at successful right branches to decide whether the advice of the VOH will be followed or not. The use of such heuristics results in an adaptive branching scheme that dynamically switches between 2-way branching and its restricted version.

6.4 Heuristics for Adaptive Branching

It is well known that variable ordering heuristics sometimes make decision errors when, based on their score, they try to select a variable for instantiation. And the probability of making a wrong decision is bigger in the case of ties, or when a set of "best variables" have scores with very small differences.

In general, when 2-way branching is used for splitting the search tree, the choice points where a VOH has to take a decision are significantly more in number compared to the choice points when d-way branching is used.

This is because in 2-way at successful right branches (where the current variable has not yet been successfully instantiated) the VOH has to take an additional decision on whether to continue with the same variable or to branch on another one. And since the number of choice points for a VOH with 2-way branching is bigger compared to d-way, the probability of making a decision error can also be bigger.

In many cases 2-way branching benefits from the extra liberty to branch on different variables at successful right branches, since the VOH used may be better informed and thus take a better decision. But since our experimental study showed that d-way branching is frequently a better choice, we conjecture that this is because sometimes erroneous decisions at successful right branches are quite costly.

The heuristics we propose next try to minimize the decision errors that can occur with 2-way branching during the variable selection process at successful right branches. The intuition behind these heuristics is twofold. First, to avoid branching on a different variable than the current one if the VOH is not "confident enough" about the correctness of this decision. And second, to identify ways to assist this decision by the use of secondary advisors. That is, VOHs that can complementarily be consulted to help in the decision making.

The two generic heuristics proposed can be used to dynamically adapt the search algorithm's branching scheme. We consider the case where dynamic adaptation involves switching between 2-way branching and restricted 2-way branching. As detailed above, the performance of d-way branching is very close to that of restricted 2-way branching. These heuristics can be applied at successful right branches. That is, when the VOH suggests to branch on another variable rather that trying the next value of the current one. Following the above intuitions we propose the following heuristics:

$\mathbf{H_{sdiff}(e)}$ **:- VOH score difference** If the current variable is x and the VOH suggests to branch on a different variable y, we will follow this suggestion only when $|score(y) - score(x)| > e$, where $score(x)$ and $score(y)$ are the values assigned by the VOH to variables x and y, while e is a threshold value difference.

$\mathbf{H_{cadv}(VOH_2)}$: **- complementary advisor** If the current variable is x and the VOH used by the algorithm (VOH_1) suggests to branch on a different variable y, we will follow this suggestion only when a secondary VOH (VOH_2) also prefers y to x. That is, when $score_{VOH_2}(y) > score_{VOH_2}(x)$, where $score_{VOH_2}(x)$ and $score_{VOH_2}(y)$ are the heuristic values assigned by VOH_2 to variables x and y[1].

Both proposed heuristics are generic, in the sense that they can be used in tandem with any VOH and any backtracking search algorithm. However, $H_{sdiff}(e)$ requires some tuning to set the value of e appropriately. In contrast, $H_{cadv}(VOH_2)$ does not require any such tuning, and can use any VOH as a secondary heuristic. The two heuristics can also be combined either conjunctively or disjunctively. In the former (resp. latter) case the suggestion to branch on a variable different than x is followed when both (resp. at least one) of the criterions for H_{sdiff} and H_{cadv} are satisfied. Importantly, the two proposed heuristics are lightweight, assuming that VOH_2 is not too expensive to compute.

6.5 Experiments with Adaptive Branching

In this section we present an experimental evaluation of the proposed adaptive branching strategies. Apart from the cpu time and the number of visited nodes, we also measure the number of *variable changes* (vc). A vc occurs when, after a failed assignment $x = a_i$ and the successful propagation of $x \neq a_i$, the VOH chooses to branch on a variable y other than x. As will be explained, this measure gives a good indication of how much an adaptive branching method approximates 2-way branching.

We have experimented with the same set of 1600 benchmarks taken from the problem classes detailed in Section 6.3. Before presenting the results, we discuss the tuning of heuristic $H_{sdiff}(e)$.

[1]We assume that a greater score is better according to VOH_2.

6.5.1 Tuning Heuristic $H_{sdiff}(e)$

Although heuristic $H_{sdiff}(e)$ is generic and can be used together with any VOH, the optimum threshold value e obviously varies among different VOHs as they may consider different metrics, such as domain sizes, constraint degrees, constraint weights, etc. Even with a fixed VOH and within a specific problem class, the optimum threshold value e may differ from instance to instance, and therefore locating it is not particularly interesting from a practical point of view. However, our experiments have demonstrated that a "good enough" value for e that carries across different problem classes can be located for a given VOH with only a few experiments.

To find a good value for e we proceeded as follows. Taking a single instance from some problem class we repeatedly solved it using the $H_{sdiff}(e)$ heuristic for branching, starting with e set to 0 and gradually increasing e in every repetition. Setting $e = 0$ forces $H_{sdiff}(e)$ to emulate 2-way branching, while as e increases, $H_{sdiff}(e)$ moves closer to restricted 2-way branching. For each run we measured the number of *variable changes* (vc). By definition, with restricted 2-way branching vc is always 0.

Specifically for $dom/wdeg$, the value of e was increased in steps of 0.01. Figures 6.4(a), (b) and (c) show the number of nodes (y-axis) as a relation of e (x-axis) for instances scen11, series12 and haystacks-05 respectively. The first data point in these plots (where $e = 0$) essentially gives the number of nodes for 2-way branching. The experiment was stopped when for some value of e the observed value of vc was 0. The last data point in each plot, corresponding to this situation, essentially gives the number of nodes for restricted 2-way branching.

In Figures 6.4(a) and (b) (scen11 and series12), where 2-way branching is better than its restricted version, we can notice that as e increases there is a point where a sharp decline in the performance of $H_{sdiff}(e)$ occurs. Respectively, in Figure 6.4(c) (haystacks-05) where restricted 2-way is better, we can notice that as e increases there is point where a sharp improvement in the performance occurs. After running similar experiments with benchmark instances from other problem classes, we observed that setting e to values around 0.1, when using $dom/wdeg$ for variable ordering, gives good results across many different instances and problem classes.

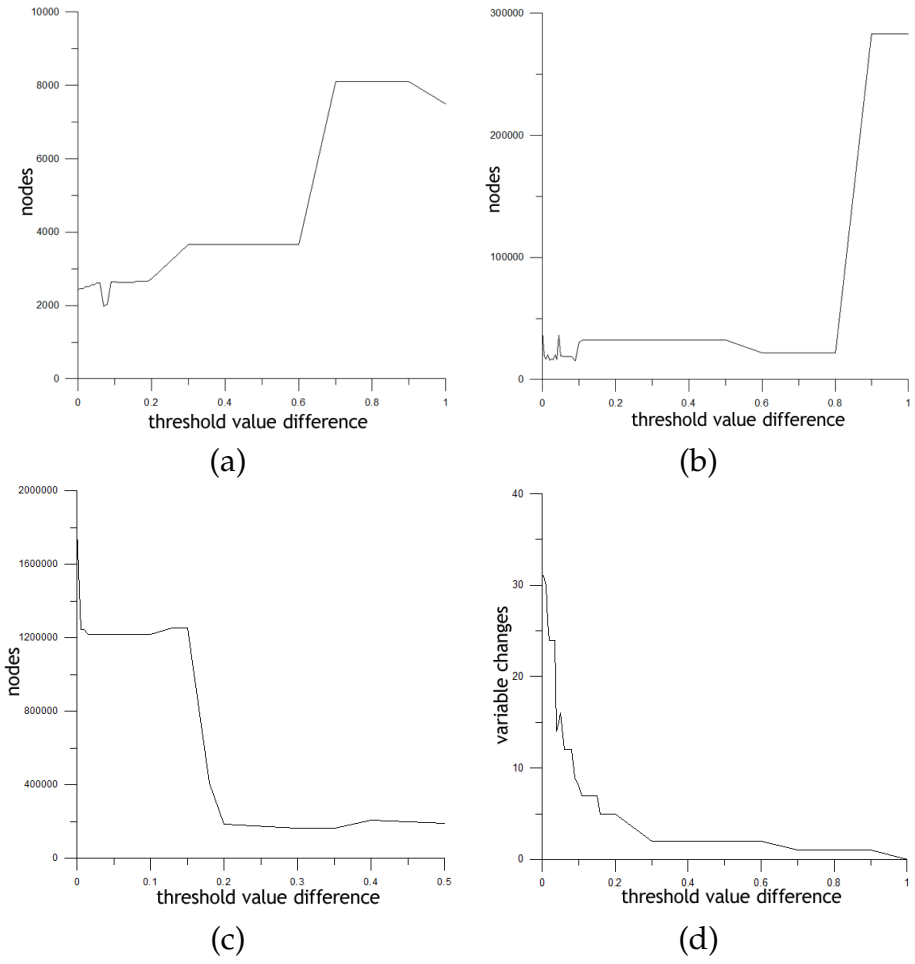

Figure 6.4: *Visited nodes over increasing values of ϵ for (a) scen11 RLFAP, (b) the series 12 instance and (c) the haystacks-05. (d) the decline in the number of variable changes over increasing values of ϵ for the scen11 RLFAP.*

Finally, Figure 6.4(d) gives the recorded value of vc (y-axis) as a relation of e (x-axis) for the RLFAP instance scen11. Not surprisingly, as the value of e increases, the value of vc decreases. Interestingly, there is a sharp decline in the value of vc roughly around the point where $e = 0.1$. A similar phenomenon was observed in all the tested instances.

6.5.2 MAC with dom/wdeg

In a first set of experiments we compare the fixed branching schemes 2-way and restricted 2-way with the following adaptive branching schemes: $H_{sdiff}(0.1)$, $H_{cadv}(wdeg)$, $H_{cadv}(impact)$. We used MAC as the search algorithm and *dom/wdeg* for variable ordering. Regarding the latter two heuristics, we intentionally chose to experiment with *wdeg* and *impact* as secondary advisors in order to observe the effects of using a secondary advisor that is similar to the primary heuristic (*wdeg*) as opposed to one that is quite diverse (*impact*).

In Table 6.11 we give indicative results from various instances. These instances were among the ones where the most notable differences between 2-way and restricted 2-way were observed. The table is divided in two parts. In the first part we include instances where restricted 2-way is better than 2-way, while in the second part instances where 2-way is better than restricted 2-way.

Results from Table 6.11 show that the adaptive branching schemes $H_{sdiff}(0.1)$ and $H_{cadv}(wdeg)$ are, in most cases, close to or even slightly superior to restricted 2-way branching in the fist part of the table, while a similar observation can be made with respect to 2-way branching in the second part of the table. Respectively, the adaptive branching schemes are clearly superior to restricted 2-way in the second part of Table 6.11, and to 2-way in the first part of the table.

The $H_{cadv}(impact)$ heuristic does not always follow the same pattern and as a result it is not as successful. The diversity between the primary heuristic and the secondary advisor sometimes pays off remarkably (haystacks-05) but on many instances it does not (e.g. geo50-20-d4-75-2). Importantly, there are instances where $H_{cadv}(impact)$ does cut down

Table 6.11: *CPU times (t) in seconds, nodes (n), and variable changes (vc) for 2-way, restricted 2-way, and the adaptive branching schemes with the* dom/wdeg *VOH.*

Instance		$2-way$	$restricted$ $2-way$	H_{sdiff} (0.1)	H_{cadv} $(wdeg)$	H_{cadv} $(impact)$
geo50-20-d4-75-1	t	2,298	1,242	**1,233**	2,311	2,331
(sat)	n	0.33M	0.2M	0.2M	0.33M	0.33M
	vc	1428	0	2	689	1289
geo50-20-d4-75-2	t	122	**28.5**	37	55	61.3
(sat)	n	41,133	6,610	8,892	13,644	34,541
	vc	642	0	2	58	354
haystacks-05	t	41.8	4	28.1	41.7	**1.6**
$(unsat)$	n	1.27M	0.13M	0.86M	1.27M	88,760
	vc	20	0	7	20	14
ruler-25-7-a3	t	12.4	4.1	5.8	**3**	15.2
(sat)	n	1,444	225	291	190	1,032
	vc	18	0	2	7	12
qwh-15-106-1	t	34.3	**14.8**	**14.8**	27.3	40.6
(sat)	n	44,150	19,964	19,964	39,158	7,862
	vc	324	0	0	73	241
bqwh-18-141-84	t	97.1	**59.1**	71.8	62.2	85.6
(sat)	n	0.33M	0.18M	0.21M	0.19M	0.22M
	vc	1,457	0	3	551	1,014
series13	t	148.4	1,132	**98.3**	150.6	205.7
(sat)	n	0.15M	1.6M	96,248	0.15M	0.16M
	vc	2,492	0	9	1,163	1,355
scen2-f25	t	42.3	183.8	45	**41.4**	45.7
$(unsat)$	n	7,819	31,516	8,074	7,794	8,118
	vc	722	0	69	484	624
scen1-f9	t	5.74	13.6	7.5	**5.7**	8.3
$(unsat)$	n	1,300	3,582	1,765	1,302	1,304
	vc	21	0	6	12	21
ruler-25-8-a3	t	50.3	180.8	139.5	54.6	55.8
$(unsat)$	n	1,829	6,407	5,022	1,905	1,899
	vc	281	0	1	159	166
qcp-15-120-6	t	62.8	80.5	96.4	81.8	94.1
(sat)	n	70,384	92,058	0.1M	90,321	0.11M
	vc	619	0	2	267	894
bqwh-18-141-98	t	6.7	9.2	**2.7**	16.3	10.5
(sat)	n	21,051	30,293	8,934	47,198	25,154
	vc	117	0	2	124	99

the explored portion of the search space (e.g. qwh-15-106-1) but the extra cpu overhead of having to compute the impacts of the variables results in slower run times. Hence, making sure that the secondary advisor is cheap to compute is important for the success of the H_{cadv} heuristic.

Generally, we can notice that although the adaptive branching schemes do not always achieve the best performance, they obtain a good trade-off between the performance of 2-way and restricted 2-way. Additionally, in many instances the adaptive branching schemes are superior to both 2-way and restricted 2-way (e.g. series-13 for $H_{sdiff}(0.1)$ and ruler-25-7-a3 for $H_{cadv}(wdeg)$). However, we should note there are some instances where one or both of the adaptive methods performed substantially worse than the winner among the standard branching schemes (e.g. bqwh-18-141-84 and ruler-25-8-a3).

Taking a closer look at the results presented in Table 6.11 it is interesting to notice the behavior of the branching schemes on instance series13. Here restricted 2-way is clearly inefficient compared to 2-way. The latter branches on a variable different than the current one after a right branch 2,492 times throughout search (i.e. $vc = 2,492$). But it is notable that H_{sdiff} manages to outperform 2-way branching by only branching on 9 different variables after right branches. Restricted 2-way branching is outperformed by a factor of 12 by only making 9 decisions against the VOH. Similar behavior can be observed in other instances of Table 6.11.

These results suggest that heuristic H_{sdiff} in particular is able to "block" variable changes that have a degrading effect on the search effort. Heuristic H_{cadv} also achieves this, but to a lesser extent, as is evident by the vc numbers.

To verify this conjecture, we rerun all the experiments and each time a different variable y than the current one x was selected at a right branch, we ordered all the (unassigned) variables according to their *dom/wdeg* value. Then we measured the distance (dis) between x and y in this ordering (obviously y was always first). Results show that the average value of dis for H_{sdiff} was significantly larger than the average dis for 2-way branching. For example in the geo50-20-d4-75-1 instance, the average dis for H_{sdiff} was 14 while for 2-way branching it was 1.6. This demonstrates that H_{sdiff}

Table 6.12: *Mean cpu times (t), and nodes (n) from binary structured and patterned problems using MAC with* dom/wdeg. *The best cpu time is given in bold.*

Series of Instances		$2-way$	$restricted\ 2-way$	H_{sdiff} (0.1)	H_{cadv} $(wdeg)$	H_{cadv} $(impacts)$
geometric	t	537.6	**440.7**	468.6	556.1	560.2
	n	0.1M	87,866	90,335	0.1M	91,547
Driver	t	35.4	34.7	35.1	**30.8**	41.3
	n	23,993	25,724	25.724	21,869	30,498
rlfapScensMod	t	8.5	36.7	9.2	**8.4**	41.5
	n	2,886	12,289	3,003	2,828	3,233
rlfapGraphsMod	t	31.7	210.6	**24.5**	40.9	230.6
	n	13,635	0.15M	11,963	15,540	10,882
Black Hole-4-4	t	15	**14.5**	14.7	15	16.1
	n	13,579	13,384	13,402	13,579	13,579
Haystacks	t	20.9	2	14.1	21	**1.6**
	n	0.89M	94,452	0.6M	0.89M	80,436
qwh-15-106	t	17	15.2	15.6	16.8	**14.9**
	n	31,479	30,629	30,629	33,531	28,755
qcp-10-67	t	135.3	**131**	140.9	147	150
	n	0.46M	0.45M	0.47M	0.48M	0.45M
qcp-15-120	t	380.4	382.3	**376.8**	395.5	401.4
	n	0.62M	0.65M	0.64M	0.65M	0.62M
Langford2	t	55.7	**51.4**	52.3	55.2	55.3
	n	0.15M	0.14M	0.14M	0.14M	0.14M
bqwh-15-106	t	2.05	**1.63**	1.88	1.92	3.8
	n	11,499	9,520	10,499	10,535	9,245
bqwh-18-141	t	13.5	**11.9**	12.2	12.6	14.7
	n	63,674	56,872	56,779	58,460	55,747
TOTAL MEAN	t	143.5	130.7	**127.9**	148.2	155.1

allows variable changes only when the selected variable is considerably superior to the current variable according to the VOH.

In Table 6.12 we give mean results from all the binary structured problem classes that we have experimented with. Corresponding results for the case of non-binary CSPs are shown in Table 6.13. In the last line of these tables we have computed the total mean.

By comparing the total mean values in both tables, we can see that the H_{sdiff} branching heuristic is clearly the best choice. As also shown in Section 6.3, restricted 2-way is better than 2-way on binary problems, while the opposite occurs on non-binary problems. The H_{sdiff} branching

Table 6.13: *Mean cpu times (t), and nodes (n) from non-binary structured problems using MAC with* dom/wdeg *as VOH. Best cpu time is in bold.*

Series of Instances		$2-way$	$restricted\ 2-way$	H_{sdiff} (0.1)	H_{cadv} $(wdeg)$	H_{cadv} $(impacts)$
allIntervalSeries	t	574.7	6,002	**385.2**	394.1	447.4
	n	0.42M	6.4M	0.31M	0.34M	0.31M
golombRuler	t	319.7	185.4	**140.4**	313.4	346.2
	n	6,541	3,662	2,948	6,639	3,139
Chessboard Coloration	t	16.1	**15.4**	**15.4**	16.5	16.6
	n	9,501	9,342	9,342	9,516	9,468
TOTAL MEAN	t	353.2	2,540	**211,7**	276.7	346.6

heuristic displays better mean performance than all the other branching methods in both binary and non-binary problems.

It is interesting to note that while in Table 6.12 H_{sdiff} achieves the best performance over all methods in only two problem classes (rlfap-GraphsMod and qcp-15-120) it succeeds in having a better mean performance overall. This is because it is able to obtains a good trade-off between the performance of 2-way and restricted 2-way branching.

On the other hand, the $H_{cadv}(wdeg)$ branching heuristic is successful in terms of mean overall performance only in the case of non-binary problems. $H_{cadv}(impact)$ is clearly the worst among the adaptive branching schemes, but still performs slightly better than 2-way in the case of non-binary problems.

6.5.3 MAC with dom/wdeg + aging

In order to examine if the success of adaptive branching is affected by the selected VOH, we also ran some experiments with the *aging* variant of *dom/wdeg*.

As in the previous section, we will fist present results from indicative instances. These are collected in Table 6.14. As we can see, the H_{sdiff} heuristic reacts adaptively but to a lesser extent compared to its behavior with *dom/wdeg*. This can be explained by the effect of the aging mechanism which periodically divides the constraint weights. These changes in the constraints weights result in larger VOH score differences among vari-

ables compared to *dom/wdeg* without aging. As a result the threshold value selected (0.1) is not high enough to block unnecessary variable changes. For example, on instances series-13 and haystacks-05 the vc numbers for H_{sdiff} in Table 6.14 are 18,946 and 7,098 respectively, while in Table 6.11 they are only 9 and 7. We believe that a higher threshold value for H_{sdiff} will result in considerable performance improvement.

Concerning the behavior of the $H_{cadv}(wdeg)$ heuristic we can notice a significant improvement compared to the results presented in Table 6.11. This improvement becomes clearer when looking at the mean results given in Tables 6.15 and 6.16. These tables give results from binary and non-binary CSPs respectively, and in both cases $H_{cadv}(wdeg)$ outperforms all the other branching methods. On the other hand, the H_{sdiff} heuristic is better that the fixed branching schemes only on non-binary problems.

6.5.4 MAC with impact

We now give results from the evaluation of the adaptive branching schemes with the *impact* VOH. This heuristic uses a different metric to compute its score. Thus, the threshold values for H_{sdiff} that are successful when *d*om/wdeg is used may not be suitable for *impact*. Indeed, after a few experiments following the procedure described in Section 6.5.1, we decided to set the threshold value for the *impact* VOH to 0.5, which is quite higher than the chosen value for *d*om/wdeg. We also include results from the $H_{cadv}(wdeg)$ adaptive branching heuristic.

In Table 6.17 we give indicative results from specific instances. As in the previous sections, this table is divided in two parts. In the upper part we present results where restricted 2-way is better than 2-way, while the lower part includes results where 2-way is better than restricted 2-way.

The adaptive heuristics mostly succeed to follow the performance of the best fixed branching scheme on each instance. And on some instances they outperform both 2-way and restricted 2-way. This is true especially for H_{sdiff}(0.5).

In Tables 6.18 and 6.19 we give mean results for all the tried binary and non-binary CSPs. Although the total number of instances that terminated

Table 6.14: *CPU times (t) in seconds, nodes (n) and variable changes (vc) for 2-way and the adaptive branching schemes using the* dom/wdeg + aging *VOH.*

Instance		$2-way$	$restricted$ $2-way$	H_{sdiff} (0.1)	H_{cadv} $(wdeg)$
geo50-20-d4-75-96	t	82	39.2	**28.9**	37.1
(sat)	n	23,838	12,126	8,420	11,714
	vc	609	0	201	22
haystacks-05	t	13	**3.4**	9.7	43.6
$(unsat)$	n	0.44M	0.10M	0.32M	1.78M
	vc	11,286	0	7,098	20
qcp-15-120-9	t	1,728	**1,301**	1,643	1,169
(sat)	n	1.78M	1.40M	0.67M	1.83M
	vc	57,332	0	46,463	2,725
qwh-15-106-5	t	128.3	104.9	95.1	**64**
(sat)	n	0.13M	0.11M	98,270	0.12M
	vc	4,054	0	2,606	191
bqwh-15-106-1	t	2.5	**1.2**	1.5	1.5
(sat)	n	7,271	3,779	4,605	7,386
	vc	192	0	110	17
bqwh-18-141-98	t	15.1	**3.3**	3.7	16.5
(sat)	n	37,001	8,317	9,031	68,321
	vc	1,166	0	259	124
series-13	t	839	1,638	660	**146.3**
(sat)	n	0.76M	1.94M	0.59M	0.15M
	vc	31,469	0	18,946	799
scen1-f9	t	**4,3**	12.4	4.5	5.4
$(unsat)$	n	1,341	4,156	1,325	1,795
	vc	36	0	33	12
scen2-f25	t	38.4	146.8	**38**	39.4
$(unsat)$	n	8,879	34,704	8,463	9,561
	vc	1,013	0	493	484
graph8-f10	t	57.1	237.6	**18.5**	43.2
(sat)	n	25,249	0.11M	6,105	21,149
	vc	1,569	0	273	245
qcp-15-120-3	t	21.5	70.2	**11.2**	31.3
(sat)	n	23,155	73,194	11,118	48,054
	vc	719	0	307	84
bqwh-18-141-38	t	**44.4**	58.8	47.3	38.2
(sat)	n	0.10M	0.15M	0.11M	0.17M
	vc	3,483	0	3,267	303

Table 6.15: *Mean cpu times (t), and nodes (n) from binary structured and patterned problems using MAC with* dom/wdeg + aging *as VOH. Best cpu time is in bold.*

Series of Instances		$2-way$	$restricted$ $2-way$	H_{sdiff} (0.1)	H_{cadv} $(wdeg)$
geometric	t	1,221	478.7	1,244	**467.1**
	n	0.26M	0.11M	0.26M	0.11M
Driver	t	**9.4**	10.1	24.8	32.9
	n	2,292	2,532	3,876	15,195
rlfapScensMod	t	**7.4**	32.2	7.8	8.1
	n	2,623	11,634	2,646	2,828
rlfapGraphsMod	t	13.4	47.5	**10.5**	42.7
	n	6,597	25,318	5,729	22,872
Black Hole-4-4	t	18.6	**13**	18.8	13.5
	n	15,483	13,384	15,309	13,579
Haystacks	t	6.5	**1.7**	4.9	21.8
	n	0.22M	53,139	0.16M	0.89M
qwh-15-106	t	29.4	21.5	21.6	**18.6**
	n	31,054	23,825	23,180	37,055
qcp-10-67	t	**22.1**	25.2	29.9	23.2
	n	68,511	84,729	87,345	78,258
qcp-15-120	t	368.8	393.6	444.1	329.5
	n	0.38M	0.52M	0.54M	0.54M
Langford2	t	117.6	56	135.3	**53.8**
	n	0.2M	0.13M	0.23M	0.14M
bqwh-15-106	t	2.64	2.33	2.78	**2.05**
	n	8,517	7,988	9,036	10,921
bqwh-18-141	t	15.5	14.1	15.8	**12.3**
	n	39,931	38,570	40,766	55,675
TOTAL MEAN	t	278.5	130.7	287.8	**123.2**

Table 6.16: *Mean cpu times (t), and nodes (n) from non-binary structured problems using MAC with* dom/wdeg + aging *as VOH. Best cpu time is in bold.*

Series of Instances		$2-way$	$restricted$ $2-way$	H_{sdiff} (0.1)	H_{cadv} $(wdeg)$
allIntervalSeries	t	2,849	7,414	2,016	**342.5**
	n	1.72M	6.38M	1.04M	0.30M
golombRuler	t	151.6	719.9	**132.3**	292,3
	n	3,032	6,952	2,726	6,639
Chessboard Coloration	t	16.6	**14.9**	16.5	17
	n	12,904	12,119	12,515	13,788
TOTAL MEAN	t	1,320	3,538	943	**253**

Table 6.17: *CPU times (t) in seconds, nodes (n) and variable changes (vc) for 2-way and the adaptive branching schemes using the* impact *VOH.*

Instance		$2-way$	$restricted$ $2-way$	H_{sdiff} (0.1)	H_{cadv} $(wdeg)$
ruler-34-9-a3	t	1,665	**874**	990	969
$(unsat)$	n	37,224	16,972	20,796	18,542
	vc	2,141	0	587	493
langford-4-10	t	82.1	**54.6**	58.8	56.9
$(unsat)$	n	5,906	4,282	4,315	4,276
	vc	47	0	1	2
frb30-15-5-mgd	t	66.4	4.4	**4.1**	6
(sat)	n	32,627	2,243	2,001	2,722
	vc	1,306	0	12	47
qwh-15-106-4	t	19.1	**2.9**	4.2	5.6
(sat)	n	59,478	1,904	5,627	8,545
	vc	168	0	38	44
bqwh-15-106-93	t	9	**1**	1.3	3.6
(sat)	n	35,676	1,778	2,294	7,697
	vc	380	0	18	85
bqwh-15-106-56	t	50.8	26.8	**24.5**	45
(sat)	n	0.29M	0.17M	0.16M	0.24M
	vc	2,073	0	285	966
series-13	t	211.2	2,710	**114.5**	380.5
(sat)	n	0.29M	4.28M	0.14M	0.56M
	vc	9,583	0	2,901	3,254
qwh-15-106-1	t	191.8	982	160.2	**117.4**
(sat)	n	0.55M	4.75M	0.60M	0.37M
	vc	2,785	0	896	532
qwh-15-106-8	t	44	90.6	**34.1**	39.2
(sat)	n	0.11M	0.29M	86,731	0.10M
	vc	604	0	308	120
bqwh-15-106-83	t	**12.5**	133.1	42	13.9
(sat)	n	60,306	0.73M	0.20M	63,852
	vc	496	0	1,606	268
bqwh-15-106-85	t	**12.6**	103.9	15.1	33.7
(sat)	n	61,147	0.54M	67,647	0.18M
	vc	499	0	450	706
frb30-15-2	t	254.9	645	**85.5**	515.5
(sat)	n	0.13M	0.35M	43,001	0.27M
	vc	3,369	0	734	1,651

Table 6.18: *Mean cpu times (t), and nodes (n) from binary structured and random problems using MAC with* impact*. Best cpu time is in bold.*

Series of Instances		$2-way$	$restricted$ $2-way$	H_{sdiff} (0.5)	H_{cadv} $(wdeg)$
Driver	t	22.9	24.2	**22.5**	24.2
	n	882	1,462	874	1,549
Haystacks	t	54.7	56.9	**44.7**	56.5
	n	0.96M	1.09M	0.77M	0.97M
qwh-15-106	t	51.3	183.6	45.2	**40**
	n	0.16M	0.82M	0.15M	0.13M
Langford	t	**148.2**	185.9	201.1	187.2
	n	0.10M	0.15M	0.12M	0.13M
bqwh-15-106	t	**9.2**	22.2	11.6	11.4
	n	49,449	0.13M	59,898	57,380
frb30-15	t	77.1	133	**56.5**	97.5
	n	34,591	69,368	27,289	50,869
TOTAL MEAN	t	**28.4**	57.5	29.1	32.7

within the time limit is considerably smaller compared to the conflict-driven heuristics, it is clear that the adaptive branching schemes are still competitive. In case of non-binary CSPs, both $H_{sdiff}(0.5)$ and $H_{cadv}(wdeg)$ are much better than the fixed branching schemes. On binary CSPs, 2-way is still a better choice, with $H_{sdiff}(0.5)$ following closely.

Comparing the results obtained with the *impact* VOH to those obtained with the conflict-driven heuristics, we can notice that on average the adaptive branching heuristics are more effective when used in tandem with a conflict-driven VOH. This can be explained if we consider that conflict-driven VOHs try to direct search to hard subproblems by identifying and selecting variables that are involved in many conflicts. The adaptive branching heuristics try to complement this by blocking variable changes that may divert search away from a currently explored hard subproblem. However, it is important that they still exhibit good performance when used in tandem with *impact*.

Table 6.19: *Mean cpu times (t), and nodes (n) from non-binary structured problems using MAC with* impact*. Best cpu time is in bold.*

Series of Instances		$2-way$	$restricted$ $2-way$	H_{sdiff} (0.5)	H_{cadv} $(wdeg)$
allIntervalSeries	t	79.6	1,040	**77.4**	175.2
	n	0.11M	1.66M	0.10M	0.27M
golombRuler	t	863.3	898.7	740.9	**730.5**
	n	14,894	12,575	9,763	9,724
Chessboard	t	73.5	**65.2**	67.6	71.8
Colloration	n	95,383	86,788	84,302	93,910
TOTAL MEAN	t	500.1	738.9	**432.4**	450.3

6.5.5 MmaxRPC with dom/wdeg

In order to investigate if and to what extent the level of local consistency applied during search affects the performance of the proposed adaptive branching schemes, we have run experiments with the MmaxRPC search algorithm. In this case only the *dom/wdeg* VOH has been tried.

In the first set of experiments we experimented with the same seven series of random problems that we used in Section 6.3. Figure 6.5(a) shows the effort (cpu time) required for solving these seven series of random instances with respect to the different tightness values. We compare here the 2-way and the $H_{sdiff}(0.1)$ branching schemes. Although on random problems the observed differences among the fixed branching schemes are not very significant, we can still notice a small improvement on instances with low tightness in favor of $H_{sdiff}(0.1)$. On instances with hight tightness 2-way branching is slightly better.

In Figure 6.5(b) we compare the adaptive branching schemes $H_{sdiff}(0.1)$ and $H_{cadv}(wdeg)$ and we depict the search effort for tightness=0.5. The 100 random instances from that set are shorted in ascending order according to cpu times. As can be seen, the performance of the adaptive branching schemes is almost identical. This was the norm with random problems.

In Table 6.20 we present results from indicative structured instances. From these results it becomes clear that the proposed heuristics still react adaptively when the level of consistency is increased. In the first part of this table, where we have selected instances in which restricted 2-way

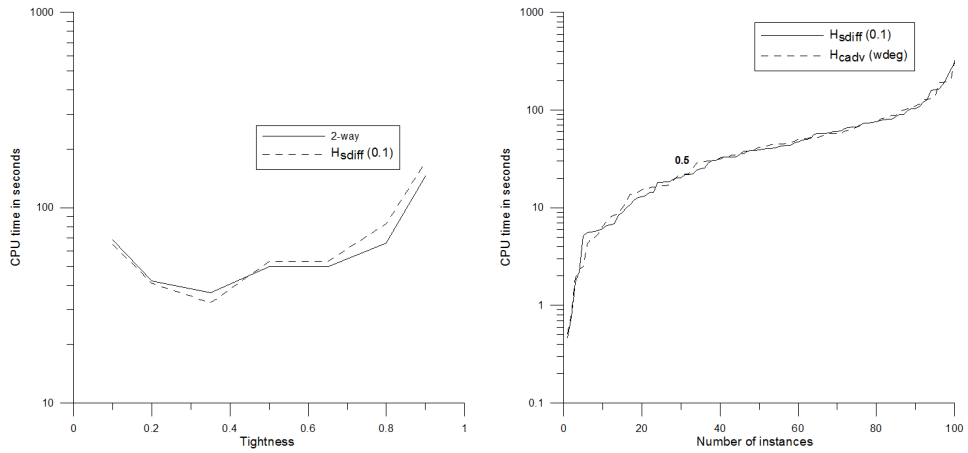

Figure 6.5: *(a) Mean search cost of solving random instances as tightness is increased. 2-way and $H_{sdiff}(0.1)$ are comparatively depicted.(b) Comparison of the $H_{sdiff}(0.1)$ and $H_{cadv}(wdeg)$ adaptive branching schemes. Solving time for the 100 random instance with tightness = 0.5, shorted in ascending order.*

is better than 2-way, both the adaptive branching heuristics are in most cases close to or even better that restricted 2-way. Respectively, in the second part of Table 6.20, where we have collected instances in which 2-way is better than restricted 2-way both the adaptive branching heuristics are much closer to 2-way.

Mean results for all the tried binary CSPs are collected in Table 6.21. Both adaptive branching schemes are on average slightly better than 2-way branching. Concerning the $H_{cadv}(wdeg)$ heuristic, the results obtained here are in accordance with the results of Section 6.3.2 where MAC with *dom/wdeg* was examined. But the mean performance of the H_{sdiff} heuristic seems to fall when the MmaxRPC search algorithm is used instead of MAC. This can be explained as follows. By increasing the level of consistency, the vc number is significantly reduced, as comparing the results of Tables 6.11 and 6.20 shows. This means that the adaptive heuristics have fewer opportunities to block potentially erroneous vc decisions, and therefore a smaller margin for improving the performance of 2-way branching.

Finally, we must keep in mind that the adaptive heuristics aim at achiev-

Table 6.20: *CPU times (t) in seconds, nodes (n) and variable changes (vc) for 2-way and the adaptive branching schemes using MmaxRPC with* dom/wdeg *as VOH.*

Instance		$2-way$	$restricted$ $2-way$	H_{sdiff} (0.1)	H_{cadv} $(wdeg)$
geo50-20-d4-75-87	t	412	**375.8**	379.4	411.7
(sat)	n	0.15M	0.14M	0.14M	0.15M
	vc	704	0	1	309
bqwh-18-141-82	t	3.2	2.6	**1.9**	2.1
(sat)	n	23,497	19,991	14,860	16,003
	vc	174	0	4	72
qcp-15-120-10	t	342.4	**252**	260.2	270.5
$(unsat)$	n	1.01M	0.78M	0.79M	0.82M
	vc	8,600	0	1	3,291
qwh-15-106-5	t	13	11.2	bf 10.1	10.7
(sat)	n	37,308	31,357	28,075	30,096
	vc	224	0	1	112
qwh-20-166-2	t	736.9	335.4	**330.8**	554.2
(sat)	n	1.04M	0.48M	0.48M	0.79M
	vc	10,364	0	0	3,369
haystacks-05	t	4	**0.3**	3.4	8.7
(sat)	n	0.16M	10,363	0.11M	0.33M
	vc	287	0	6	14
geo50-20-d4-75-20	t	65.3	237.6	**50.4**	75.7
(sat)	n	22,680	85,958	16,054	25,948
	vc	107	0	3	61
geo50-20-d4-75-74	t	45.5	53.4	**13.5**	48.4
(sat)	n	20,078	29,241	5,950	20,279
	vc	134	0	2	60
bqwh-18-141-2	t	4.2	6.9	5.6	**2.8**
(sat)	n	0.55M	4.75M	0.60M	0.37M
	vc	2,785	0	896	532
scen11-f8	t	**113.3**	965.4	163.2	114
$(unsat)$	n	12,681	93,429	16,118	12,407
	vc	261	0	73	268
scen2-f25	t	**14**	54.5	18	15
$(unsat)$	n	1,719	6,704	2,125	1,728
	vc	37	0	33	31
graph9-f9	t	**68.9**	265.3	72	78.9
(sat)	n	16,839	62,058	16,813	18,613
	vc	213	0	123	246

Table 6.21: *Mean cpu times (t), and nodes (n) from binary structured and patterned problems using MmaxRPC with dom/wdeg as VOH. Best cpu times are in bold.*

Series of Instances		$2-way$	$restricted$ $2-way$	H_{sdiff} (0.1)	H_{cadv} $(wdeg)$
geometric	t	99.5	**92.5**	92.9	101.3
	n	35,974	34,590	33,433	36,121
Driver	t	**13.2**	18.2	21.4	21.4
	n	11,255	14,530	14,530	18,219
rlfapScensMod	t	**7.3**	18.3	8.2	7.5
	n	1,067	2,527	1,178	1,056
rlfapGraphsMod	t	**14.9**	45.8	15.1	15.7
	n	4,006	11,647	3,938	4,120
rlfapScen11	t	**93.7**	778.4	112.5	95.2
	n	14,664	0.11M	16,351	14,579
Black Hole-4-4	t	3.5	3.5	3.5	3.6
	n	4,741	4,407	4,514	4,741
Haystacks	t	4	**0.3**	3.4	8.7
	n	0.89M	94,452	0.6M	0.89M
qcp-10-67	t	11.4	**10.6**	11.4	11.1
	n	78,989	76,569	80,889	77,811
qcp-15-120	t	62.9	**54.5**	55.7	57.6
	n	0.18M	0.16M	0.16M	0.16M
qwh-15-106	t	3.7	3.5	**3.3**	3.4
	n	10,686	10,063	9,516	9,571
qwh-20-166	t	219	200	196.5	**189.3**
	n	0.31M	0.29M	0.29M	0.27M
bqwh-18-141	t	3	**2.9**	3.1	3.1
	n	21,456	21,957	22,312	22,552
TOTAL MEAN	t	52.7	85.9	**50.1**	50.9

ing a trade-off between the mean performance of 2-way and that of restricted 2-way. Since 2-way is constantly better (or least very close) to restricted 2-way per problem class when maxRPC is maintained, the best we can hope for the adaptive schemes is to match the performance of 2-way. This is certainly achieved.

6.5.6 Statistical analysis

Results from the previous subsections have shown that adaptive branching can be beneficial in a wide range of benchmarks. Indeed, on average it gives better results than 2-way branching. In order to evaluate the statistical significance of our experimental results, a statistical analysis through a set of *paired t-tests* was performed.

The dependent *t-test* (also called the *paired t-test* or *paired-samples t-test*) compares the means of two related groups to detect whether there are any statistically significant differences between these means. Here, we analyze the cpu performance of the adaptive branching schemes compared to 2-way, over all the structured instances with which we have experimented (binary and non-binary). To be more precise, for each VOH and for each search algorithm, we have compared the corresponding 2-way branching scheme with the relative adaptive branching schemes. For example we have compared 2-way branching with the $H_{sdiff}(0.1)$ and the $H_{cadv}(wdeg)$ branching heuristics, when MAC is used in conjunction with the *dom/wdeg* VOH. In case of the *impact* VOH, we have compared the corresponding 2-way branching (where the impact VOH is used) with the relative adaptive branching schemes.

We have excluded from our statistical analysis random instances and instances that can be solved in less than a second. We have measured the mean difference in seconds, standard deviation, t-value and the 95% confidence interval. The risk level (called alpha level) has been set to 0.05. Results are collected in Table 6.22.

In Table 6.22, the mean cpu reduction in all cases is always greater than zero. However, the negative values at the confidence interval indicate that this reduction was not observed in all the tried instances. According

Table 6.22: *Paired t-test measurements for evaluation of the significance of the experimental results. The first group corresponds always to 2-way branching, while the second group is the H_{sdiff} or the H_{cadv}. Statistically significant t-values are in bold.*

	MAC						MmaxRPC	
	dom/wdeg		dom/wdeg + aging		impacts		dom/wdeg	
	H_{sdiff}	H_{cadv}	H_{sdiff}	H_{cadv}	H_{sdiff}	H_{cadv}	H_{sdiff}	H_{cadv}
Mean	25.1	1.22	21.1	230.3	8.1	2.5	2.6	1.8
SD	195.3	99.3	332.6	1,180	81	85.6	41.8	22.9
t-value	**1.95**	0.18	0.95	**2.9**	0.99	0.28	0.7	0.88
95% C.I.	(-0.2, 50.4)	(-11.6, 14.1)	(-22.4, 64.6)	(76, 384)	(-8.13, 24.3)	(-14.6, 19.6)	(-4.4, 9.7)	(-2.16, 5.6)

to standard tables of significance (available as an appendix in the back of most statistics texts) the critical t-value for characterizing a result as significant was in all cases 1.65. Thus, we can confirm that t-values for $H_{sdiff}(0.1)$ when MAC is used with $dom/wdeg$ as VOH and for $H_{cadv}(wdeg)$ when MAC is used with $dom/wdeg + aging$ as VOH, are large enough to be significant.

On the other hand, when MAC is used with the $dom/wdeg$ VOH, the statistical significant improvement of $H_{sdiff}(0.1)$ over 2-way, is an important result. Since the combination of 2-way branching with $dom/wdeg$ on MAC, is widely used in the research community as the most effective way to solve CSPs.

6.6 Conclusions

2-way and d-way are the standard branching schemes employed by the vast majority of constraint solvers. Although in theory the former can be exponentially more efficient than the latter, there is little empirical evidence concerning a practical comparison of their performance. In this Chapter, we empirically evaluate the two branching schemes as well as a widely used restricted version of 2-way branching. We consider a number of different variable ordering heuristics as well as different levels of local consistency. Results show that, unsurprisingly, the d-way and restricted 2-way branching schemes are closely matched across the different VOHs,

with d-way being slightly more cost effective. Also, confirming the theoretical results, exponential differences in favor of full 2-way branching are observed as soon as we move from a simple heuristic like smallest domain (*dom*) to more sophisticated ones like domain over dynamic degree (*dom/ddeg*). But perhaps surprisingly, when state-of-the-art heuristics like *dom/wdeg* and *impact* are used, significant differences in favor of d-way (and restricted 2-way) are also observed.

Based on these results we introduce generic heuristics that can be used to dynamically decide whether the variable ordering heuristic will be followed or not at certain points during search. The application of such heuristics results in an adaptive branching scheme that switches between 2-way branching and its restricted version, which is close to d-way branching. Experiments with instantiations of the generic heuristics confirm that the adaptive heuristics achieve a trade-off between 2-way and restricted 2-way. As a result, search with adaptive branching outperforms search with a fixed branching scheme on a wide range of problems.

The work presented here is, to the best of our knowledge, the first attempt towards designing heuristics for adaptive branching and contributes to the design and implementation of adaptive constraint solvers.

I want to be what I was when I wanted to be what I am now.

Prince, Ray

7 Set Branching

In this chapter, we propose and evaluate a generic approach to set branching where the partition of a domain into sets is created using the scores assigned to values by a value ordering heuristic, and a clustering algorithm from machine learning. Experimental results demonstrate that although exponential differences between branching schemes, as predicted in theory between 2-way d-way branching, are not very common, still the choice of branching scheme can make quite a difference on certain classes of problems. Set branching methods are very competitive with 2-way branching and outperform it on some problem classes. A statistical analysis of the results reveals that our generic clustering-based set branching method is the best among the methods compared.

7.1 Introduction

Although 2-way and d-way are the most widely used branching schemes, another technique that can also be used is dichotomic domain splitting [31]. This method originates from numerical CSPs and proceeds by splitting the current domain of the selected variable into two sets, usually based on the lexicographical ordering of the values. In this way branching is performed on the two created sets and the branching factor is reduced to two. Although domain splitting drastically reduces the branching factor, it can result in a much deeper search tree since the effects of propagation after a branching decision may be diminished.

In addition to these standard schemes, techniques that group together the values of the selected variable, and branch on these created groups instead of individual values, have been proposed [45, 53, 77, 10, 84, 50]. The criteria used for the grouping of values and the methods used to perform the grouping can be different, but all these techniques aim at reducing the size of the search tree. In this chapter, following [50], we call any such method a *set branching* method.

Our first goal in this chapter is to experimentally study the effect of different branching schemes for finite domain CSPs on search performance. Although some existing branching methods have been compared to one another (e.g. [78]), to our knowledge this is the first systematic evaluation of several existing alternatives.

In addition, we propose and study a generic set branching method where the partition of a domain into sets is created using the scores assigned to values by a value ordering heuristic, and a clustering algorithm. Before employing such a method, two fundamental questions need to be addressed: What is the measure of similarity between values, and how do we partition domains using such a measure? Most of the approaches to set branching that have been proposed in the past have either used very strict measures of similarity or are problem specific. Our method offers a generic solution to both the problem of similarity evaluation and the partitioning of domains. For the former we exploit the information acquired from the value ordering heuristic, while for the latter we use a clustering algorithm from machine learning.

Experimental results from a wide range of benchmarks demonstrate that exponential differences between branching schemes, as predicted in theory between 2-way d-way, are not very common. But although the choice of branching scheme does not have as a profound effect as the choice of variable ordering heuristic, it can still make a difference. The generic set branching methods we evaluate outperform the standard 2-way branching scheme in many problem classes resulting in better average performance. It is notable that our clustering-based set branching method displays very promising results without any tuning of the clustering algorithm applied. Importantly, a statistical analysis of the exper-

imental results reveals that this method is the best among the methods compared.

The rest of this Chapter is organized as follows. In Section 7.2, we briefly recall the most well known fixed branching schemes by also noting their differences. In Section **??** we discuss past work on set branching for CSPs. In Section 7.3 we propose a new generic method for set branching which is based on a machine learning clustering algorithm. In Section 7.4 we report results from an experimental evaluation of the various branching schemes including a statistical analysis. Finally, in Section 7.5 we conclude.

7.2 Alternative branching schemes

Dichotomic *domain splitting* [31] is a branching scheme that originates from numerical CSPs. This method proceeds by splitting the current domain of the selected variable into two sets, usually based on the lexicographical ordering of the values. Once the domain has been split, the second set of values is removed from the domain and this removal is propagated. In this way branching is performed on the two created sets and the branching factor is reduced to two. However, domain splitting tends to achieve weaker propagation compared to d-way and 2-way branching. So, although it drastically reduces the branching factor, it can result in a much deeper search tree. Domain splitting is mostly used on optimization problems and especially when the domains of the variables are very large. An example of a search tree explored with domain splitting is shown in Figure 7.1c.

Very recently, Kitching and Bacchus explored the applicability of *set branching* for constraint optimization problems [50]. The basic idea is to group together values that offer similar improvement to the currently computed bounds. In this way entire groups of values that offer no improvement to the bounds can be refuted, resulting in smaller tree sizes.

In this chapter we use the term *set branching* to refer to any branching technique that, using some similarity criterion, identifies values that can be grouped together and branched on as a set. Dichotomic domain splitting and 2-way branching can be seen as manifestations of this generic

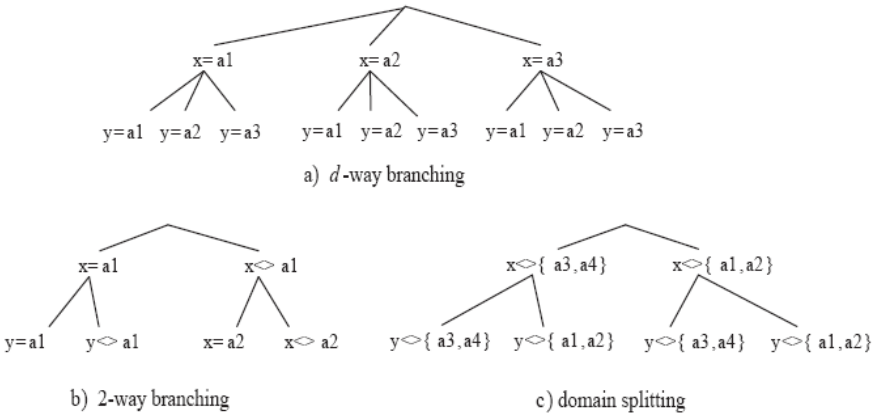

Figure 7.1: *Examples of search trees for the three branching schemes.*

method that use simple grouping criteria. Domain splitting creates two sets of values based on their lexicographical ordering. 2-way branching splits the domain into two sets where the first includes a single value and the second the rest of the values. In general, in order to define a set branching technique, two questions need to be addressed: *What is the measure of similarity between values, and how are domains partitioned using such a measure*?

The idea of set branching for CSPs has been explored in the past. Freuder introduced the notion of interchangeability, substitutability, and their weaker, but tractable, neighborhood versions as means to identify values with similar behavior [33]. Two values of a variable are neighborhhood interchangeable iff they have exactly the same supports in all constraints. One value a is neighborhood substitutable for another value b if the set of values inconsistent wth a is a subset of the values inconsistent with b. These notions were exploited, for example in [45, 10, 67], to group together values when branching and in this way perform set branching. The drawback of these techniques is that their conditions are too strong, as in many problems neighborhood interchangeable and substitutable values are very rare.

Larrosa investigated the merging of similar subproblems during search using forward checking [53]. According to this approach, values whose as-

signment leads to similar subproblems are grouped together and branched on as a set. Experiments performed on crossword puzzle generation problems displayed promising results. However, the measure of subproblem similarity and the algorithm used to partition the domains according to this measure are both problem specific.

Silaghi et al. proposed a method for partitioning the domains of variables based on the Cartesian product representation of the search space [77]. This method is particularly suitable for finding all solutions but it requires an explicit extensional representation of the constraints in the problem.

A generic and simple approach to set branching that can be applied on a wide range of problems was proposed by van Hoeve and Milano [84]. In this approach, values that are "tied" according to their value ordering heuristic score are grouped together and branching is performed on the sets of values created. Assignment of specific values to variables is postponed until lower levels of the search tree (which is also done in Larossa's method). Experiments using both depth-first search and limited discrepancy search displayed promising results. However, this method relies heavily on the particular value ordering heuristic used and the number of ties produced by the value ordering heuristic, which can be quite low in many cases. Also, this method distinguishes between values that have very close but not equal scores and as a result such values will be placed into different sets. As noted in [84], the concept of a tie can be extended to refer to values having close scores. In this chapter we explore this idea further.

7.3 Clustering for Set Branching

As we intend to apply set branching dynamically throughout search, after selecting a variable x with current domain $D(x) = \{a_1, \ldots, a_d\}$, we are faced with the following problem. We have to create a partition $S_{D(x)} = \{s_1, \ldots, s_m\}$ of $D(x)$ into m sets s.t. each value $a_i \in D(x)$ belongs to only one set $s_j \in S$. Ideally, we want all the values that have been assigned to a specific set to be similar according to some measure of similarity.

Following van Hoeve and Milano, we use a generic measure of similarity that is based on the score of the values according to a value ordering heuristic. In order to perform the dynamic partitioning of domains into sets, we propose the use of clustering algorithms from machine learning. Our approach can be summarized as follows. A value ordering heuristic is used to assign a score v_i to each value $a_i \in D(x)$. The collection of d items (values) and the matrix of their scores are given as input to a clustering algorithm. The output of the algorithm will be the partition $S_{D(x)} = \{s_1, \ldots, s_m\}$.

Compared to [84] our approach has the following potential benefits. First, not only will tied values be placed in the same set, but with high probability so will values that have very close scores. Hence, there will be fewer sets, resulting in lower branching factor. Second, in cases where there are no ties, the method of [84] uses d-way branching. In contrast, our approach will still partition the domain if there are groups of values with similar score.

The algorithm we currently use to create the clustering of values is *x-means* [66]. This is an extension of the well known k-means algorithm that is considerably faster and does not require to predetermine the desired number of clusters. The algorithm starts with randomly selected points (values in our case) as cluster centroids and iteratively improves the computed clustering until a fixpoint is reached. Several parameters of the algorithm can be tuned to give more accurate results on a specific application, including the starting centroids, the number of iterations, the measure of distance between points, etc. Although we intend to investigate this in the future, in the experiments reported below we use the Weka implementation of the x-means algorithm as is, without any tuning.

7.4 Empirical evaluation

We have experimented with 350 instances from ten classes of real world, academic, patterned, and random CSPs taken from C.Lecoutre's XCSP repository. We included both satisfiable and unsatisfiable instances. Each selected instance involves constraints defined either in intension or in ex-

tension. The CSP solver used in our experiments is a generic solver and has been implemented in the Java programming language. This solver essentially implements the M(G)AC search algorithm, where (G)AC-3 is used for applying (G)AC. Since our solver does not yet support global constraints (apart from the table constraint) , we have left experiments with problems that include such constraints as future work. All experiments were run on an Intel dual core PC T4200 2GHz with 3GB RAM.

For a fair evaluation of the different branching schemes we use the same propagation method during search (arc consistency), the same variable ordering heuristic (*dom/wdeg* [18]) and value ordering heuristic (*Geelen's promise* [37]). The promise metric is calculated over all the visited nodes of the search tree. This penalizes run times and as a result may be inefficient in some problems, but for the purposes of this initial investigation we only wanted to use a reasonably sophisticated value ordering heuristic throughout all the tried instances. In the future we intend to experiment with different value ordering heuristics and study their effect on the performance of the clustering set branching method.

We compare the following branching schemes:

2-way Values are chosen in descending order of their promise.

d**-way** Values are chosen in descending order of their promise.

domain splitting The values are ordered according to their promise and then the domain is split in half. The part with the top ranked values is tried first.

ties set branching This is the method of [84] where values with the same promise form a set. The sets are tried in descending order of promise.

clustering set branching This is our method where x-means is used to partition the domain into sets based on the promise of the values. The sets are tried in descending order of promise. Note that the clusters are linearly ordered since clustering is done over only one dimension.

The two set branching methods have been implemented using a 2-way and a d-way branching style, giving four alternatives. More specifically, past works on set branching for CSPs perform set branching using a d-way style. That is, once the partition of the domain $S_{D(x)} = \{s_1, \ldots, s_m\}$ is created, search proceeds by removing from $D(x)$ any value a, s.t. $a \notin s_1$, and propagating. If there is a failure, the same process is repeated for s_2 and so on. We have also implemented and evaluated 2-way style set branching. In this case the generated sets are tried in a series of binary choices. That is, after the reduction of $D(x)$ to s_1 fails, we propagate the removal from $D(x)$ of all the values in s_1. If this succeeds then we reduce $D(x)$ to s_2 and so on.

We must clarify here that in all the "2-way style" branching variants (domain splitting, ties, clustering) the set branching method allows to jump from one variable to another as standard 2-way branching does.

In addition, for domain splitting and the set branching methods we have tried two options: 1) Domain splitting (resp. set branching) is performed throughout search on all variables. 2) Domain splitting (resp. set branching) is performed on a variable only if its domain size is greater than a certain percentage of its original domain size. We have tried several values for this percentage, with 25% giving the best results. This can improve the performance of domain splitting by 30% on average, and it can offer (minor) improvement to set branching. Therefore, in the reported experiments with these methods Option 2 is followed.

Table 7.1 compares the various branching methods on specific instances from the tested problem classes. We display CPU times as well as nodes. A node in 2-way branching can correspond to a value assignment or to a value removal, while in d-way branching it can only correspond to a value assignment. Hence, they cannot be compared directly. The instances in this table are chosen to highlight the gaps in performance that can occur when using different branching schemes. As can be seen any method can be the best on a given instance, and there can be very considerable variance in the performance of the methods. For instance, clustering set branching can be very effective on certain problems (e.g. qcp-15-120-8) but it can also be quite ineffective on others (e.g. qcp-15-120-6). How-

Table 7.1: *Cpu times (t), and nodes (n) from specific instances. Cpu times are in seconds. The best result for each instance is given in bold.*

Problem Class		*d-way*	*2-way*	*dom split.*	*d-way ties set branch.*	*2-way ties set branch.*	*d-way clust. set branch.*	*2-way clust. set branch.*
*frb*35-17-2	t	**43.3**	98.4	954	60.1	98.3	134	154
(*sat*)	n	16241	45098	515909	27160	50713	58633	75743
*scen*3-*f*11	t	73.7	**6.9**	33.8	40.1	11.3	43.5	14.5
(*unsat*)	n	11056	1739	5318	11019	4021	13631	5705
pigeons-30-*ord*	t	2435	**572**	762	1259	773	1322	639
(*unsat*)	n	376384	135031	128286	338049	247792	364343	228190
*geo*50-20-*d*4-75-7	t	472	1338	2815	**190**	1309	365	543
(*sat*)	n	108027	404918	686333	58411	443724	111505	174716
langford-2-10	t	300	129	605	**108**	120	116	127
(*unsat*)	n	199104	247286	372733	199609	235912	203580	238314
driverw-09	t	177	145	243	**103**	164	180	143
(*sat*)	n	75625	93236	97180	46823	76510	77509	64798
qcp-15-120-6	t	23.8	12.4	26	28.8	**9.6**	133	94.6
(*sat*)	n	19074	20179	19353	33003	12019	136599	99847
qcp-15-120-8	t	50	35.4	53.2	44.4	130	**1.01**	**1.01**
(*sat*)	n	38227	49680	38551	46188	146342	845	845
*geo*50-20-*d*4-75-11	t	41.6	38.9	94.2	32.5	37.9	**12.2**	15.1
(*sat*)	n	9027	10044	21926	8990	12620	3486	5111
queensKnights-15-5	t	1506	1001	2245	1502	737	999	**594**
(*unsat*)	n	42154	15393	86199	42309	38836	28312	30890

ever, these are some of the most 'extreme' instances. Exponential differences, as predicted between 2-way and d-way in theory, occured rarely[1]. We must also note here, that these results (for 2-way and d-way branching schemes) are not directly compared with the reported results from the previous chapter. Since the value ordering heuristic that we have used in each case is different. In this experimental analysis, we have used the Geelen's promise [37] as the value ordering heuristic, while in Chapter 6 we have used lexicographic value ordering.

In Tables 7.2 and 7.3 we summarize the results of our experimental

[1]But this observation concerns the variable ordering heuristic and propagation method used here and may not generalize as shown in Chapter 6.

Table 7.2: *Average speed-up (positive values) or slow-down (negative values) achieved by 2-way branching compared to the other branching methods. Cpu time (t) in seconds and visited nodes (n) have been measured.*

Problem Class	*% graph density*		*d-way*	*dom split.*	*d-way ties set branch.*	*2-way ties set branch.*	*d-way clust. set branch.*	*2-way clust. set branch.*
langford	1.045	t	2.88	5.08	-1.21	-1.11	-1.20	-1.04
(*unsat*)		n	-1.27	1.52	-1.26	-1.06	-1.23	-1.03
pigeons	1	t	1.13	1.24	-1.53	-1.89	-1.07	-1.32
(*unsat*)		n	-1.21	1.33	-1.7	-1.66	-1.25	-1.12
queensKnights	0.70	t	1.49	1.99	1.75	-1.21	-1.02	-1.48
(*unsat*)		n	2.85	4.96	3.47	3.04	1.87	2.39
forced random	0.65	t	-1.22	1.88	-1.30	-1.03	-1.14	1.14
(*sat*)		n	-1.41	1.52	-1.11	-1.1	-1.24	1.07
geometric	0.35	t	-2.48	2.07	-4.55	-1.03	-3.83	-2.58
(*sat*)		n	-3.02	1.79	-3.77	1.18	-3.53	-2.25
qcp − qwh	0.125	t	1.78	2.34	1.28	1.99	6.08	5.63
(*sat*)		n	-1.09	1.12	-1.06	1.5	4.08	3.84
driver	0.082	t	1.18	1.53	-1.33	1.10	1.21	1.00
(*sat*)		n	-1.23	-1.06	-1.71	-1.24	-1.23	-1.43
rlfap (*ScensMod*)	0.052	t	5.39	3.07	3.52	1.07	3.70	1.26
(*mixed*)		n	4.63	2.73	4.28	1.77	4.94	2.1
graphColoring	0.05	t	-1.50	1.01	-1.58	1.00	-1.49	-1.03
(*mixed*)		n	-1.28	1.15	-1.18	1.14	-1.17	-0.92

evaluation. We use 2-way branching as the standard all other branching methods are compared against. In Table 7.2 we give the average slow-down (or speed-up) of the methods compared to 2-way for each problem class (the two quasigroup classes qcp and qwh are grouped together). We have mostly selected problem classes that contain either only satisfiable or only unsatisfiable instances. However, we have also experimented with "mixed" problem classes. That is classes that contain both satisfiable and unsatisfiable instances. For example, on langford problems all instances are unsatisfiable and 2-way is 2.88 times better than d-way on average, while it is 1.2 times worse than d-way clustering set branching. As mentioned above, it is difficult to accurately compare the numbers of visited nodes under different branching schemes. However, in most prob-

lem classes the differences in Cpu times roughly reflect the differences in visited nodes.

In Table 7.3 we give the percentage of instances, over all the tried instances, where each method was faster (> 1), at least 2 times faster (> 2), and at least 3 times faster (> 3) than 2-way branching. Similarly for instances where each method was slower by < 1, < 2, and < 3 times compared to 2-way.

Table 7.2 shows that although differences between methods can be quite large on single instances, the average differences between the most competitive methods are smaller. Dichotomic domain splitting is apparently the worst among the branching methods. However, it may fare better in problems with very large domain sizes[2]. Excluding domain splitting, the other methods are usually no more that 2 times better or worse than 2-way branching on average. But there are cases where even the average differences are quite large.

The set branching methods, and especially the d-way style ones, have slightly better or very close performance compared to 2-way branching on most classes. Also, these methods clearly outperform d-way branching. Interestingly, the set clustering methods are typically very competitive on the denser classes.

Table 7.3: *% categorization of all tried instances according to the performance of the branching methods compared to 2-way branching.*

Problem Class	*speedup*	*d-way*	*dom split*	*d-way ties set branch.*	*2-way ties set branch.*	*d-way clust. set branch.*	*2-way clust. set branch.*
all instances	>1	29%	11%	47%	68%	50%	45%
	>2	8%	0%	8%	2%	15%	16%
	>3	2%	0%	3%	0%	10%	6%
	<1	71%	89%	53%	32%	50%	55%
	<2	24%	56%	21%	2%	21%	15%
	<3	11%	34%	6%	3%	11%	6%

Table 7.3 shows that 2-way ties set branching is better than 2-way on most instances. However, the margins are usually small. This is because

[2]Most domains included between 2 and 50 values, with maximum 225.

Table 7.4: *Paired t-test measurements for evaluation of the significance of the experimental results. 2-way branching is compared with the other branching schemes.*

	Mean	SD	t-value	95% C.I.
d-way	-29.8	341.7	-0.68	(-116, 57)
domain splitting	-241	456	-4.1	(-357, -125)
d-way ties set branching	9.48	326.3	0.23	(-73.3, 92.3)
2-way ties set branching	31.7	234	1.06	(-27.7, 91.1)
d-way clustering set branching	13.75	217.9	0.49	(-41.6, 69)
2-way clustering set branching	32.4	182.5	1.4	(-13.9, 78.7)

the number of ties that occur during search is usually low, meaning that 2-way ties set branching often emulates the standard 2-way scheme. The other set branching methods are better than 2-way on roughly half of the instances. However, they can be significantly better, and worse, on quite a few.

In order to obtain a global view and to evaluate the statistical significance of our experimental results, a set of paired t-tests were performed. In these tests we compared the CPU performance of the 2-way branching scheme against all the other branching schemes, over all the instances used in the experiments. We measured the mean difference, standard deviation, t-value and the 95% confidence interval. The risk level (called alpha level) was set to 0.05. Results are collected in Table 7.4.

As the results show, d-way branching and domain splitting are clearly inefficient compared to 2-way branching. The mean CPU reduction in the all set branching techniques is always greater than zero with 2-way clustering set branching being slightly better. However, the negative values at the confidence interval indicate that this reduction was not observed in all the tried instances. Although 2-way ties and clustering set branching achieve equivalent mean CPU reduction, the t-values score show that the spread (or variability) of the scores for 2-way clustering set branching is significantly higher compared to 2-way ties set branching. The t-value scores lead us to conclude that 2-way clustering set branching is a promising branching technique, since in our experiments it has displayed the best overall performance.

Finally, we have to mention that the number of clusters produced by x-means during search was usually quite low (2-3). In some cases, typically for small domain sizes, there was only one cluster generated because all values had similar score. In such a case our method switched to either d-way or 2-way branching depending on the style of set branching employed.

7.5 Conclusions

In this chapter, we performed an experimental evaluation of branching methods for CSPs including the commonly used 2-way and d-way schemes as well as other less widely used ones. We also proposed and evaluated a generic set branching method that partitions domains into sets of values by using information provided by the value ordering heuristic as input to a clustering algorithm. Results showed that set branching methods, including our approach, are competitive and often better compared to standard 2-way branching. We now plan to investigate ways to achieve more efficient domain partitions by automatically tuning the parameters of the clustering algorithm. Also, it would be interesting to study clustering of domains using information from multiple value ordering heuristics.

One thing life has taught me: if you are interested, you never have to look for new interests. They come to you. When you are genuinely interested in one thing, it will always lead to something else.

Eleanor Roosevelt

8 Conclusions and Future Work

In this dissertation we investigate adaptive search strategies for the CSP with the aim to increase the practical efficiency of backtracking search. In general, we contribute to the design and implementation of adaptive and autonomous constraint solvers that have the ability to advantageously modify modelers decisions that typically in mainstream CP solvers are taken prior to search. In the next sections we summarize our main contribution and we give interesting directions on how this work can be extended in the future.

8.1 Conclusions

The most important results and contributions from the work presented in this thesis are now reviewed.

Adaptive search-guiding heuristics Adaptive variable ordering heuristics, learn and use information from every node explored in the search tree, whereas traditional static and dynamic heuristics only use information about the initial and current nodes. These conflict-driven heuristics follow the learning-from-failure approach, in which information regarding failures is stored in the form of constraint weights. By recording constraints that are responsible for any value deletion, we derive three new heuristics that use this information to spread constraint weights. We also explore a SAT inspired constraint aging strategy that gives greater importance to recent conflicts. Finally

we proposed a new heuristic that tries to better identify contentious constraints by recording all the potential conflicts upon detection of failure.

Empirical evaluation of modern VOHs For a first time a wide empirical evaluation of modern variable ordering heuristic is performed. All these modern heuristics have been tested over a narrow set of problems in their original papers and they have been compared mainly with older heuristics. Hence, there is no comprehensive view of the relative strengths and weaknesses of these heuristics. State-of-the-art VOHs can be divided in two main categories: heuristics that exploit information about failures gathered throughout search and recorded in the form of constraint weights and heuristics that measure the importance/impact of variable assignments for reducing the search space. Results demonstrate that, in general, heuristics based on failures have much better cpu performance. Although impact based heuristics are in general slow, there are some cases where they perform a smarter exploration of the search tree resulting in fewer node visits.

Adaptive revision orderings The performance of propagation algorithms is affected by the order in which revisions are carried out. Based on our observation concerning the interaction between conflict-driven variable ordering heuristics and revision ordering heuristics, we extend the use of failures discovered during search to devise new, efficient and adaptive revision ordering heuristics. We propose a number of simple revision ordering heuristics based on constraint weights for arc, variable, and constraint oriented implementations of coarse grained arc consistency algorithms, and compare them to the most efficient existing revision ordering heuristic. Importantly, the new heuristics can not only reduce the numbers of constraints checks and list operations, but also cut down the size of the explored search tree. Results from various structured and random problems demonstrate that some of the proposed heuristics can offer significant speed-ups.

Adaptive branching schemes We have developed two generic heuristics

that can be applied at successful right branches once the variable ordering heuristic chooses to branch on a variable other than the current one. At this point the heuristics are used to decide whether the advice of the variable ordering heuristic will be followed or not. The application of these heuristics results in an adaptive branching scheme that dynamically switches between the fixed branching schemes. The first heuristic is based on measuring the difference between the scores that the variable ordering heuristic assigns to its selected variable and the current variable. The second heuristic is based on the use of a secondary advisor to decide if the variable ordering heuristic will be followed or not. Experiments with instantiations of the two generic heuristics confirm that search with adaptive branching outperforms search with a fixed branching scheme on a wide range of problems.

New set branching method We propose and study a generic set branching method where the partition of a domain into sets is created using the scores assigned to values by a value ordering heuristic, and a machine learning clustering algorithm. Most of the approaches to set branching that have been proposed in the past have either used very strict measures of similarity or are problem specific. Our method offers a generic solution to both the problem of similarity evaluation and the partitioning of domains. For the former we exploit the information acquired from the value ordering heuristic, while for the latter we use a clustering algorithm from machine learning.

8.2 Future Work

Our study on ways to create adaptive methods and strategies for solving CSps is by no means complete. There are many issues that require further investigation and many aspects that have not been addressed here. In the next paragraphs we discuss some of them.

By extending our solvers capability to handle global constraints, we intent to experimentally examine the behavior of the proposed and existing

adaptive variable ordering heuristics, on problems with global constraints. Concerning revision ordering heuristics, we plan to evaluate the inverse arc-oriented heuristics: $a\_dom/wdeg\_inverse$ and $a\_dom/wcon\_inverse$, which favor revising arcs (c_{ij},x_i) such that x_j, has small domain size. Also, it would be interesting to apply similar ideas to propagator-oriented solvers.

On set branching methods, we plan to investigate ways to achieve more efficient domain partitions by automatically tuning the parameters of the clustering algorithm. Also, it would be interesting to study clustering of domains using information from multiple value ordering heuristics.

All the adaptive strategies that have been proposed in this thesis, have proven to be beneficial. But in our empirical studies each adaptive strategy have been evaluated separately. It would be interesting to intergrade all of them on an new adaptive constraint solver and to analyze possible effective combinations.

Bibliography

[1] F. Bacchus. Extending forward checking. In *Proceedings of CP-2000*, pages 35–51, 2000.

[2] T. Balafoutis, A. Paparrizou, and K. Stergiou. Experimental Evaluation of Branching Schemes for the CSP. In *TRICS workshop at CP-2010*, pages 1–12, 2010.

[3] T. Balafoutis, A. Paparrizou, K. Stergiou, and T. Walsh. Improving the Performance of maxRPC. In *Proceedings of CP-2010*, pages 69–83, 2010.

[4] T. Balafoutis and K. Stergiou. Experimental evaluation of modern variable selection strategies in constraint satisfaction problems. In *Proceedings of the 15th RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion.*, 2008.

[5] T. Balafoutis and K. Stergiou. Exploiting constraint weights for revision ordering in arc consistency algorithms. In *Proceedings of the ECAI-2008 workshop on Modeling and Solving Problems with Constraints*, 2008.

[6] T. Balafoutis and K. Stergiou. On conflict-driven variable ordering heuristucs. In *Proceedings of the ERCIM workshop - CSCLP*, 2008.

[7] T. Balafoutis and K. Stergiou. Adaptive branching for constraint satisfaction problems. In *ECAI'10*, pages 855–860, 2010.

[8] T. Balafoutis and K. Stergiou. Conflict directed variable selection strategies for constraint satisfaction problems. In *SETN*, pages 29–38, 2010.

[9] T. Balafoutis and K. Stergiou. Evaluating and Improving Modern Variable and Revision Ordering Strategies in CSPs. *Fundamenta Informaticae*, 102(3-4):229–261, 2010.

[10] A. Beckwith and B. Choueiry. On the dynamic detection of interchangeability in finite constraint satisfaction problems. In *Proceedings of CP-01*, page 760, 2001.

[11] C. Bessiere. Constraint Propagation. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 3. Elsevier, 2006.

[12] C. Bessière, A. Chmeiss, and L. Sais. Neighborhood-based variable ordering heuristics for the contraint satisfaction problem. In *Proceedings of the 7th Conference on Principles and Practice of Constraint Programming (CP-2001)*, pages 61–75, 2001.

[13] C. Bessière, E.C. Freuder, and J.C. Régin. Using Inference to Reduce Arc Consistency Computation. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-1995)*, pages 592–599, 1995.

[14] C. Bessière and J.C. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?). In *Proceedings of CP-1996*, pages 61–75, Cambridge MA, 1996.

[15] C. Bessière, J.C. Régin, R. Yap, and Y. Zhang. An Optimal Coarse-grained Arc Consistency Algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

[16] C. Bessiere, K. Stergiou, and T. Walsh. Domain filtering consistencies for non-binary constraints. *Artificial Intelligence*, 172(6-7):800–822, 2008.

[17] F. Boussemart, F. Hemery, and C. Lecoutre. Revision ordering heuristics for the Constraint Satisfaction Problem. In *10th International Conference on Principles and Practice of Constraint Programming (CP-2004), Workshop on Constraint Propagation and Implementation*, Toronto, Canada, 2004.

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

[19] D. Brelaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.

[20] B. Cabon, S. De Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio Link Frequency Assignment. *Constraints*, 4:79–89, 1999.

[21] H. Cambazard and N. Jussien. Identifying and Exploiting Problem Structures Using Explanation-based Constraint Programming. *Constraints*, 11:295–313, 2006.

[22] A. Chmeiss and P. Jégou. Efficient path-consistency propagation. *Journal on Artificial Intelligence Tools*, 7(2):121–142, 1998.

[23] D. Cohen, P. Jeavons, P. Jonsson, and M. Koubarakis. Building tractable disjunctive constraints. *Journal of the ACM*, 47:826–853, 2000.

[24] M. Correia and P. Barahona. On the integration of singleton consistency and look-ahead heuristics. In *Proceedings of the ERCIM workshop - CSCLP*, pages 47–60, 2007.

[25] R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *CP-97*, pages 312–326, 1997.

[26] R. Debruyne and C. Bessière. Some practicable path filtering techniques for the constraint satisfaction problem. In *IJCAI-97*, pages 412–417, 1997.

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

[28] R Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[29] R. Dechter and I. Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-1989)*, pages 271–277, 1989.

[30] Y. Deville and P.V. Hentenryck. An efficient arc consistency algorithm for a class of CSP problems. In *Proceedings of IJCAI-91*, pages 325–330, 1991.

[31] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of FGCS-88*, pages 693–702, 1988.

[32] M. van Dongen. AC-3_d an efficient arc-consistency algorithm with a low space-complexity. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP-2002)*, volume 2470, pages 755–760, 2002.

[33] E. Freuder. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *Proceedings of AAAI-91*, pages 227–233, 1991.

[34] E. Freuder and A. Mackworth. Constraint satisfaction: an emerging paradigm. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 2. Elsevier, 2006.

[35] E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.

[36] D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-1995)*, pages 572–578, 1995.

[37] P. A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of ECAI-92*, pages 31–35, 1992.

[38] I. Gent and T. Walsh. Csplib: a benchmark library for constraints. In *Proceedings of CP-99*, 1999.

[39] I.P. Gent, E. MacIntyre, P. Prosser, P. Shaw, and T. Walsh. The constraindedness of arc cosistency. In *Proceedings of the 3rd Conference on Principles and Practice of Constraint Programming (CP-1997)*, pages 327–340, 1997.

[40] I.P Gent, E. MacIntyre, P. Prosser, B.M. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of the 2nd Conference on Principles and Practice of Constraint Programming (CP-1996)*, pages 179–193, 1996.

[41] E. Goldberg and Y. Novikov. BerkMin: a Fast and Robust Sat-Solver. In *Proceedings of DATE'02*, pages 142–149, 2002.

[42] C.P. Gomes, C. Fernandez, B. Selman, and C. Bessiere. Statistical regimes across constrainedness regions. In *Proceedings of CP-2004*, pages 32–46, 2004.

[43] D. Grimes and R.J. Wallace. Sampling strategies and variable selection in weighted degree heuristics. In *Proceedings of CP-2007*, pages 831–838, 2007.

[44] R.M. Haralick and Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–314, 1980.

[45] A. Haselbock. Exploiting interchangeabilities in constraint satisfaction problems. In *Proceedings of IJCAI-93*, pages 282–287, 1993.

[46] M.C. Horsch and W.S. Havens. An empirical study of probabilistic arc consisteny as a variable ordering heuristic. In *Proceedings of the 6th Conference on Principles and Practice of Constraint Programming (CP-2000)*, pages 525–530, 2000.

[47] J. Hwang and D. Mitchell. 2-Way vs. d-Way Branching for CSP. In *Proceedings of CP-2005*, pages 343–357, 2005.

[48] S.A. ILOG. Ilog solver 6.0 user's manual, 2003.

[49] D.S. Johnson and M.A. Eds Trick. Second dimacs implementation challenge:cliques, coloring and satisfiability. vol 26, of dimacs series in discrete mathematics and theoretical computer science, 1996.

[50] M. Kitching and F. Bacchus. Set Branching in Constraint Optimization. In *Proceedings of IJCAI-09*, pages 532–537, 2009.

[51] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89:365–387, 1997.

[52] F. Laburthe and N. Jussien. Choco constraint programming system. *Available at http://choco.sourceforge.net*, 2003–2011.

[53] J. Larrosa. Merging constraint satisfaction problems to avoid redundant search. In *Proceedings of IJCAI-97*, pages 424–433, 1997.

[54] C. Lecoutre. Optimization of Simple Tabular Reduction for Table Constraints. In *Proceedings of CP-2008*, pages 128–143, 2008.

[55] C. Lecoutre and F. Hemery. A study of residual supports in arc cosistency. In *Proceedings of IJCAI-2007*, pages 125–130, 2007.

[56] C. Lecoutre and S. Tabary. Abscon 109: a generic csp solver. In *Proceedings of the 2nd International CSP Solver Competition, held with CP-2006*, pages 55–63, 2008.

[57] C. Likitvivatanavong, Y. Zhang, J. Bowen, S. Shannon, and E. Freuder. Arc Consistency during Search. In *Proceedings of IJCAI-2007*, pages 137–142, 2007.

[58] D. Long and M. Fox. The third international planning competition., http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume20/long03a-html/node37.html, 2002.

[59] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[60] A. Mackworth. On reading sketch maps. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-1977)*, pages 598–606, Cambridge MA, 1977.

[61] J.J. McGregor. Relational consistency algorithms anf their applications in finding subgraph and graph isomorphism. *Information Science*, 19:229–250, 1979.

[62] R. Mohr and T. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[63] P. Morris. The breakout method for escaping from local minima. In *Proceedings of AAAI-93*, pages 40–45, 1993.

[64] M. Moskewicz, C. Madigan, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of Design Automation Conference*, pages 530–535, 2001.

[65] V. Park. An empirical study of different branching strategies for constraint satisfaction problems, Master's thesis, University of London, 2004.

[66] D. Pelleg and A. Moore. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In *Proceedings of ICML-2000*, pages 727–734, 2000.

[67] S. Prestwich. Full Dynamic Interchangeability with Forward Checking and Arc Consistency. In *Proceedings of the ECAI Workshop on Modeling and Solving Problems With Constraints*, 2004.

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

[69] P. Prosser, K. Stergiou, and T. Walsh. Singleton consistencies. In *Proceedings of the 6th Conference on Principles and Practice of Constraint Programming (CP-2000)*, pages 353–368, 2000.

[70] P. Refalo. Impact-based search strategies for constraint programming. In *Proceedings of CP-2004*, pages 556–571, 2004.

[71] O. Roussel and C. Lecoutre. Xml representation of constraint networks: Format xcsp 2.1. In *CoRR abs/0902.2362*, 2009.

[72] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP-1994*, pages 10–20, 1994.

[73] D. Sabin and E.C. Freuder. Understanding and Improving the MAC Algorithm. In *Proceedings of CP-1997*, pages 167–181, 1997.

[74] C. Schulte, M. Lagerkvist, and G. Tack. Gecode solver. *Available at http://www.gecode.org*, 2011.

[75] C. Schulte and P.J. Stuckey. Efficient constraint propagation engines. *ACM Transactions on Programming Languages and Systems*, 31:2.1–2.43, 2008.

[76] B. Selman and H. Kautz. Domain-independent Extensions to GSAT: Solving Large Structured Satisfiability Problems. In *Proceedings of IJCAI-93*, pages 290–295, 1993.

[77] M. Silaghi, D. Sam-Haroud, and B. Faltings. Intelligent Domain Splitting for CSPs with Ordered Domains. In *Proceedings of CP-99*, pages 488–489, 1999.

[78] B. Smith and P. Sturdy. Value Ordering for Finding All Solutions. In *Proceedings of IJCAI-05*, pages 311–316, 2005.

[79] B.M. Smith. The brelaz heuristic and optimal static orderings. In *Proceedings of the 5th Conference on Principles and Practice of Constraint Programming (CP-1999)*, pages 405–418, 1999.

[80] B.M. Smith and S.A. Grant. Trying harder to fail first. In *Proceedings of 13th European Conference on Artificial Intelligence (ECAI-1998)*, pages 249–253, 1998.

[81] G. Smolka. The OZ Programming Model. In *Computer Science Today (LNCS 1000)*, pages 324–343, 1995.

[82] J. Thornton. *Constraint weighting local search for constraint satisfaction*. PhD thesis, Griffith University, Australia, 2000.

[83] P. van Beek. Backtracking Search Algorithms. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 4. Elsevier, 2006.

[84] J. van Hoeve and M. Milano. Postponing Branching Decisions. In *Proceedings of ECAI-04*, pages 1105–1106, 2004.

[85] R. Wallace and E. Freuder. Ordering heuristics for arc consistency algorithms. In *AI/GI/VI*, pages 163–169, Vancouver, British Columbia, Canada, 1992.

[86] R.J. Wallace and D. Grimes. Experimental studies of variable selection strategies based on constraint weights. *Journal of Algorithms*, 63(1–3):114–129, 2008.

[87] T. Walsh. Search in a small world. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-1999)*, pages 1172–1177, 1999.

[88] K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. A simple model to generate hard satisfiable instances. In *Proceedings of IJCAI-2005*, pages 337–342, 2005.

[89] R. Zabih. Some applications of graph bandwith to constraint satisfaction problems. In *Proceedings of AAAI'90*, pages 46–51, 1990.

[90] A. Zanarini and G. Pesant. Solution counting algorithms for constraint-centered search heuristics. In *Proceedings of the 13th Conference on Principles and Practice of Constraint Programming (CP-2007)*, pages 743–757, 2007.