The Importance of Variable Ordering in Constraint Satisfaction Problems

Thanasis Balafoutis

Department of Information and Communication Systems Engineering
University of the Aegean, Samos, Greece
abalafoutis@aegean.gr

Summary. *Constraint programming is a powerful technique for solving combinatorial search problems that draws on a wide range of methods from artificial intelligence and computer science. The basic idea in constraint programming is that the user states the constraints and a general purpose constraint solver is used to solve the resulting constraint satisfaction problem. A key factor that can dramatically reduce the search space during constraint solving is the criterion under which the variable to be instantiated next is selected. For this purpose numerous strategies have been proposed. Some of the best of such strategies exploit information about failures gathered throughout search and recorded in the form of constraint weights, while others measure the importance of variable assignments in reducing the search space. In this work we give an introduction on the constraint satisfaction problems. We have also collect and present the most recent and powerful variable ordering heuristics that have been proposed in the literature. Our intention is to provide a comprehensive view of the relative strengths and weaknesses of these heuristics. We also provide insight as to which heuristic is preferable as a general purpose variable ordering strategy.*

1 Introduction

Constraint programming is a powerful technique for solving combinatorial search problems that draws on a wide range of methods from artificial intelligence, computer science, operations research, programming languages and databases. Constraint programming is currently applied with success to many domains, such as scheduling, planning, vehicle routing, configuration, networks, and bioinformatics. The basic idea in constraint programming is that the user states the constraints and a general purpose constraint solver is used to solve the resulting constraint satisfaction problem. Since constraints are relations, a Constraint Satisfaction Problem (CSP) states which relations hold among the given decision variables.

Constraints are a powerful and natural means of knowledge representation and reasoning. As an example lets consider the sports league scheduling, 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).

CSPs can be solved either systematically, as with backtracking, or using forms of local search which may be 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. In other words, 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 come with a 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.

When solving a CSP using backtracking search, a sequence of decisions must be made as to which variable to instantiate next. These decisions are referred to as the variable ordering decisions. It has been shown that for many problems the choice of variable ordering can have a dramatic effect on the performance of the backtracking algorithm with huge variances even on a single instance [13, 23].

A variable ordering can be either static, where the ordering is fixed and determined prior to search, or dynamic, where the ordering is determined as the search proceeds. Dynamic variable orderings are considerably more efficient and have thus received much attention in the literature. One common dynamic variable ordering strategy, known as "fail-first", is to select as the next variable the one likely to fail as quickly as possible. All other factors being equal, the variable with the smallest number of viable values in its (current) domain will have the fewest subtrees rooted at those values, and therefore, if none of these contain a solution, the search can quickly return to a path that leads to a solution.

Recent years have seen the emergence of numerous modern heuristics for choosing variables during CSP search. The so called conflict-driven heuristics exploit information about failures gathered throughout search and recorded in the form of constraint weights, while other heuristics measure the importance of variable assignments in reducing the search space. Most of them are quite successful and choosing the best general purpose heuristic is not easy.

This paper is targeted to readers that are not familiar with CSPs. Initially we give an extended introduction on this field. Moreover, we have collect and present the most recent and powerful variable ordering heuristics that have been proposed in the literature. Our intention is to provide a comprehensive view of the relative

strengths and weaknesses of these heuristics. We also provide insight as to which heuristic is preferable as a general purpose variable ordering strategy.

The rest of the paper is organized as follows. Section 2 gives an extended introduction on the constraint satisfaction problems, while Section 3 briefly summarizes the related work that has been done in variable ordering heuristics. In Section 4 we make a general discussion on the relative strengths and weaknesses of these heuristics. Finally, conclusions are presented in Section 5.

2 The Constraint Satisfaction Problem

A *Constraint Satisfaction Problem* (CSP) is a tuple (X, D, C), where X is a set containing n variables $\{x_1, x_2, ..., x_n\}$; D is a set of domains $\{D(x_1)$, $D(x_2)$,..., $D(x_n)\}$ for those variables, with each $D(x_i)$ consisting of the possible values which x_i may take; and C is a set of e constraints $\{c_1, c_2, ..., c_e\}$ between variables in subsets of X. Each $c_i \in C$ expresses a relation defining which variable assignment combinations are allowed for the variables in the scope of the constraint, *vars(c_i)*. Two variables are said to be *neighbors* if they share a constraint. 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 binary constraint between variables x_i and x_j will be denoted by c_{ij}.

A partial assignment is a set of tuple pairs, each tuple 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.

In binary CSPs any constraint c_{ij} defines two directed arcs (x_i,x_j) and (x_j,x_i). A directed constraint (x_i,x_j) is *arc consistent* (AC) iff for every value $a \in D(x_i)$ there exists at least one value $b \in D(x_j)$ such that the pair (a,b) satisfies c_{ij}. In this case we say that b is a *support* of a on the directed constraint (x_i,x_j). Accordingly, a is a support of b on the directed constraint (x_j,x_i). A problem is AC iff there are no empty domains and all arcs 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), is a direct extension of the definition of AC. A non-binary constraint c, with $vars(c)$=$\{x_1, x_2, ..., x_k\}$, is GAC iff for every variable $x_i \in vars(c)$ and every value $a \in D(x_i)$ there exists a tuple τ that satisfies c and includes the assignment of a to x_i [16, 15]. In this case τ is a support of a on constraint c. A problem is GAC iff all constraints are GAC. In the rest of the paper we will assume that (G)AC is the propagation method applied to all constraints.

Many consistency properties and corresponding propagation algorithms stronger than AC and GAC have been proposed in the literature. One of the most studied is singleton (G)AC which, as we will explain in the following section, has also been used to guide the selection process for a certain variable ordering heuristic. A variable x_i is *singleton generalized arc consistent* (SGAC) iff for each value $a_i \in D(x_i)$, after assigning a_i to x_i and applying GAC in the problem, there is no empty domain [9].

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.

Complete search algorithms for CSPs are typically based on backtracking depth-first search where branching decisions (i.e. variable assignments) are interleaved with constraint propagation. The most efficient search algorithm that is used in the main academic and commercial solvers is MGAC (maintaining generalized arc consistency) or MAC in the case of binary problems [20, 4]. This algorithm can be implemented using either a *d-way* or a *2-way* branching scheme. The former works as follows. Initially, the whole problem should be made GAC before starting search. After the first variable x with domain $D(x) = \{a_1, a_2, ..., a_d\}$ is selected, d recursive calls are made. In the first call value a_1 is assigned to x and the problem is made GAC, i.e. all values which are not GAC given the assignment of a_1 to x are removed. If this call fails (i.e. no solution is found), the value a_1 is removed from the domain of x and the problem is made again GAC. Then a second recursive call under the assignment of a_2 to x is made, and so on. The problem has no solution if all d calls fail. In *2-way* branching, after a variable x and a value $a_i \in D(x)$ are selected, two recursive calls are made. In the first call a_i is assigned to x, or in other words the constraint x=a_i is added to the problem, and GAC is applied. In the second call the constraint $x \neq a_i$ is added to the problem and GAC is applied. The problem has no solution if neither recursive call finds a solution. The main difference of these branching schemes is that in *2-way* branching, after a failed choice of a variable assignment (x,a_i) the algorithm can choose a new assignment for any variable (not only x). In *d-way* branching the algorithm has to choose the next available value for variable x.

3 Variable ordering heuristics

The order in which variables are assigned by a backtracking search algorithm has been understood for a long time to be of primary importance. The first category of heuristics used for ordering variables was based on the initial structure of the network. These are called static or fixed variable ordering heuristics (SVOs) as they simply replace the lexicographic ordering by something more appropriate to the structure of the network before starting search. Examples of such heuristics are *min width* which chooses an ordering that minimizes the width of the constraint network [11], *min bandwidth* which minimizes the bandwidth of the constraint graph [25], and *max degree* (*deg*), where variables are ordered according to the initial size of their neighborhood [10].

A second category of heuristics includes dynamic variable ordering heuristics (DVOs) which take into account information about the current state of the problem at each point in the search. The first well known dynamic heuristic, introduced by Haralick and Elliott, was *dom* [14]. 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, for binary CSPs, the variable that is constrained with the largest number of unassigned variables. By combining *dom* and *deg* (or *ddeg*), the heuristics called *dom/deg* and *dom/ddeg* [4, 22] 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.

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* [12], *dom+ddeg* [6, 21] and *BZ3* [21].

Bessière et al. [3], 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:

$$H_a^{\circledcirc}(x_i) = \frac{\sum_{x_j \in \Gamma(x_i)} (\alpha(x_i) \circledcirc \alpha(x_j))}{|\Gamma(x_i)|^2} \tag{1}$$

where $\Gamma(x_i)$ is the set of variables that share a constraint with x_i and $\alpha(x_i)$ 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.

Boussemart et al. [5], inspired from SAT (satisfiability testing) solvers like Chaff [17], proposed conflict-driven variable ordering heuristics. In these heuristics, 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. 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.

Grimes and Wallace [24] proposed alternative conflict-driven heuristics that consider value deletions as the basic propagation events associated with constraint weights. That is, the weight of a constraint is incremented each time the constraint causes one or more value deletions. 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. Using this method *global contention*, i.e. contention that holds across the entire search space, can be uncovered.

Inspired by integer programming, Refalo introduced an *impact* measure with the aim of detecting choices which result in the strongest search space reduction [19]. 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:

$$\overline{I}(x_i = \alpha) = \frac{\sum_{k \in K} I^k(x_i = \alpha)}{|K|} \tag{2}$$

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:

$$I(x_i) = \sum_{\alpha \in D(x_i)} 1 - \overline{I}(x_i = \alpha) \tag{3}$$

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.

Correia and Barahona [8] proposed variable orderings, by integrating Singleton Consistency propagation procedures with look-ahead heuristics. This heuristic is similar to “node impacts”, but instead of computing the accurate impacts, it computes the reduction in the search space after the application of Restricted Singleton Consistency (RSC) [18], 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 [7] 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.

Zanarini and Pesant [26] 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 large, it has been shown that for certain widely-used global constraints, such information can be computed efficiently.

Finally, Balafoutis and Stergiou proposed [2] new variants of conflict-driven heuristics. These variants differ from *wdeg* in the way they assign weights. They propose heuristics that record the constraint that is responsible for any value deletion during search, heuristics that give greater importance to recent conflicts, and finally heuristics that try to identify contentious constraints by detecting all possible conflicts after a failure. The last heuristic, called “fully assigned”, 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 (revisions that delete at least one value). Hence, this heuristic records all variables that delete at least one value during constraint propagation and if a DWO is detected, it increases the weight of all these variables by one.

4 General Discussion

The heuristics described in the previous section can be divided into two different strategies: i) heuristics that exploit information about failures gathered throughout

search and recorded in the form of constraint weights (conflict-driven heuristics) and ii) heuristics that measure the importance of variable assignments in reducing the search space (impact based heuristics).

In the first strategy belongs the *wdeg* and *dom/wdeg* heuristics from [5], the random probing heuristics from [24] and the *fully assigned* from [2]. In the second strategy belongs the *impact* based heuristics from [19] and its variants from [8].

In a recent work [1] where all these heuristics have been tested experimentally, results have shown that the best general purpose variable ordering heuristic is the *dom/wdeg*. The main advantage that this heuristic have, is the ability to learn from failures during search with a low computational cost.

In order to get a graphical view of the relative performance between *dom/wdeg* and the *Impact* heuristic, we present two figures. In these figures we have included cpu time and number of visited nodes for the *dom/wdeg* heuristic and we have compared it graphically to the *Impact* heuristic (which has the best performance among the impact based heuristics).

The results are from a well known real world problem called RLFAP (Radio Link Frequency Assignment Problem). This problem 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.

Results are collected in Figure 1. The left plot in this figure correspond to cpu time and the right plot to visited nodes. Each point in these plots, shows the cpu time (or nodes visited) for one instance from all the tested RLFAPs. 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)). Therefore, a point above line $y = x$ represents an instance which is solved faster (or with less node visits) using *dom/wdeg* heuristic. Both axes are logarithmic.

As we can clearly see from Figure 1 (left plot), *dom/wdeg* heuristic is 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 *dom/wdeg* is 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 left plot of Figure 1. But the additional computational overhead of computing the “best” choices, really affect the overall performance of the impact heuristic (Figure 1, right plot). 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.

On the other hand in other problem classes where variables have only a few values in their domains (as in the Boolean instances where variables have only two values in their domains) the impact heuristic is quite competitive.

5 Conclusions

In this paper we make an extended introduction on the Constraint Satisfaction Problems, for readers that are not familiar with this filed. Moreover, we have collect

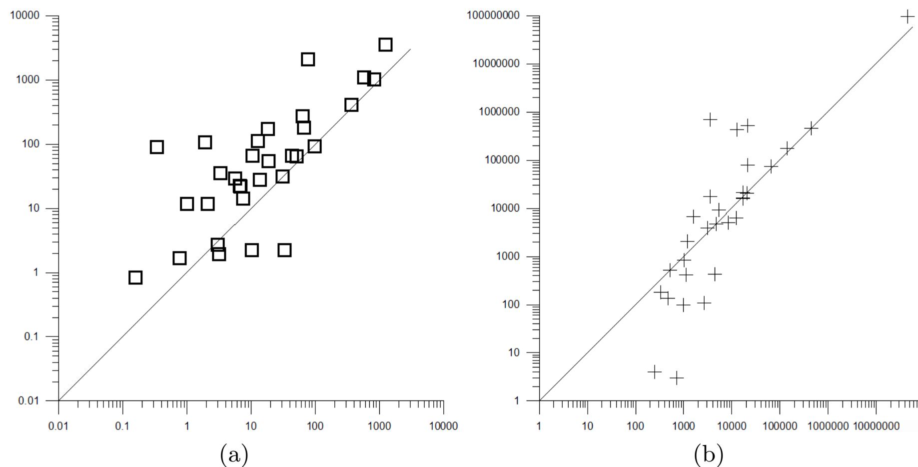

Fig. 1. A summary view of run times (left figure) and nodes visited (right figure), for *dom/wdeg* and *impact* heuristics (figures (a),(b)).

and present the most recent and powerful variable ordering strategies that are used in order to boost search in constraint satisfaction problems. These heuristics 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. 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.

References

1. 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, online at CEUR Workshop Proceedings (ISSN 1613-0073), Vol. 451*, 2008.
2. T. Balafoutis and K. Stergiou. On conflict-driven variable ordering heuristucs. In *Proceedings of the ERCIM workshop - CSCLP. Available at http://pst.istc.cnr.it/CSCLP08/*, 2008.
3. 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.
4. C. Bessière and J.C. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?). In *Proceedings of the 2nd Conference on Principles and Practice of Constraint Programming (CP-1996)*, pages 61–75, Cambridge MA, 1996.

5. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of 16th European Conference on Artificial Intelligence (ECAI-2004)*, pages 146–150, Valencia, Spain, 2004.
6. D. Brelaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.
7. H. Cambazard and N. Jussien. Identifying and Exploiting Problem Structures Using Explanation-based Constraint Programming. *Constraints*, 11:295–313, 2006.
8. 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.
9. R. Debruyne and C. Bessière. Domain Filtering Consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.
10. 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.
11. E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.
12. 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.
13. 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.
14. R.M. Haralick and Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–314, 1980.
15. A. Mackworth. On reading sketch maps. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-1977)*, pages 598–606, Cambridge MA, 1977.
16. R. Mohr and T. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.
17. M. Moskewicz, C. Madigan, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of Design Automation Conference*, pages 530–535, 2001.
18. 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.
19. P. Refalo. Impact-based search strategies for constraint programming. In *Proceedings of the 10th Conference on Principles and Practice of Constraint Programming (CP-2004)*, pages 556–571, 2004.
20. D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings 2nd Workshop on Principles and Practice of Constraint Programming (CP-1994)*, pages 10–20, 1994.
21. 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.
22. 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.

23. P. van Beek. Backtracking Search Algorithms. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 4. Elsevier, 2006.
24. 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.
25. R. Zabih. Some applications of graph bandwith to constraint satisfaction problems. In *Proceedings of AAAI'90*, pages 46–51, 1990.
26. 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.