# On Conflict-driven variable ordering heuristics

Thanasis Balafoutis and Kostas Stergiou

Department of Information and Communication Systems Engineering
University of the Aegean, Samos, Greece
email: {abalafoutis,konsterg}@aegean.gr

**Abstract.** It is well known that the order in which variables are instantiated by a backtracking search algorithm can make an enormous difference to the search effort in solving CSPs. Among the plethora of heuristics that have been proposed in the literature to efficiently order variables during search, a significant recently proposed class uses the learning-from-failure approach. Prime examples of such heuristics are the wdeg and dom/wdeg heuristics of Boussemart et al. which store and exploit information about failures in the form of constraint weights. The efficiency of all the proposed conflict-directed heuristics is due to their ability to learn though conflicts encountered during search. As a result, they can guide search towards hard parts of the problem and identify contentious constraints. Such heuristics are now considered as the most efficient general purpose variable ordering heuristic for CSPs. In this paper we show how information about constraint weights can be used in order to create several new variants of the *wdeg* and *dom/wdeg* heuristics. The proposed conflict-driven variable ordering heuristics have been tested over a wide range of benchmarks. Experimental results show that they are quite competitive compared to existing ones and in some cases they can increase efficiency.

## 1 Introduction

Constraint satisfaction problems (CSPs) and propositional satisfiability (SAT) are two automated reasoning technologies that have a lot in common regarding the approaches and algorithms they use for solving combinatorial problems. Most complete algorithms from both paradigms use constraint propagation methods together with variable ordering heuristics to improve search efficiency. Learning from failure has become a key component in solving combinatorial problems in the SAT community, through clause learning and weighting, e.g. as implemented in the Chaff solver [18]. This approach is based on learning new clauses through conflict analysis and assigning weights to clauses based on the number of times they cause a failure during search. This information can be then exploited by the variable ordering heuristic to efficiently choose the variable to assign at each choice point. Following the pioneering work of [18], numerous such heuristics have been proposed in the SAT literature [13, 17, 20, 9].

In the CSP community, learning from failure has followed a similar direction in recent years, in particular with respect to novel variable ordering heuristics.

Boussemart et al. were the first to introduce SAT influenced heuristics that learn from conflicts encountered during search [5]. In their approach, constraint weights are used as a metric to guide the variable ordering heuristic towards hard parts of the problem. Constraint weights are continuously updated during search using information learned from failures. The advantage that these heuristics have is that they use previous search states as guidance, while most formerly proposed heuristics either use the initial or the current state. The heuristics of [5], called *wdeg* and *dom/wdeg*, are now probably considered as the most efficient general purpose variable ordering heuristic for CSPs. Sunsequently, a number of alternative heuristics based on learning during search were proposed [19, 7, 14].

As discussed by Grimes and Wallace, heuristics based on constraint weights can be conceived in terms of an overall strategy that except from the standard Fail-First Principle also obeys the Contention Principle, which states that variables directly related to conflicts are more likely to cause a failure if they are chosen instead of other variables [14]. x

In this paper we focus on conflict-driven variable ordering heuristics based on constraint weights. After a recall on the existing heuristics for CSPs and a brief discussion on their efficiency, we concentrate on an investigation of new general purpose variants of conflict-driven heuristics. These variants differ from *wdeg* and *dom/wdeg* in the way they assign weights to constraints. First we propose three new variants of the *wdeg* and *dom/wdeg* heuristics that record the constraint that is responsible for any value deletion during search. These heuristics then exploit this information to update constraint weights uppon detection of failure. We also examine a SAT influenced weight aging strategy that gives greater importance to recent conflicts. Finally, we propose a new heuristic that tries to better identify contentious constraints by detecting all the possible conflicts after a failure. Experimental 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.

The rest of the paper is organized as follows. Section 2 gives the necessary background material and an overview on the existing variable ordering heuristics. In Section 3 we propose several new general purpose variants of conflict-driven variable ordering heuristics. In Section 4 we experimentally compare the proposed heuristics to *dom/wdeg* on a variety of real, academic and random problems. Finally, conclusions are presented in Section 5.

## 2 Background

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 constraints $\{c_1, c_2, ..., c_k\}$ between variables in subsets of $X$. Each $c_i \in C$ expresses a relation defining which variable assignment combinations are allowed for the variables *vars*$(c_i)$ in the scope of the constraint. 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 $c_{ij}$. In this paper we focus on binary CSPs. However, the proposed variable ordering heuristics are generic and can be applied on problems with constraints of any arity.

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 state. A full assignment is one containing all $n$ variables. A solution to a CSP is a full assignment such that no constraint is violated.

An *arc* is a pair $(c, x_i)$ where $x_i \in vars(c)$. As we focus on binary CSPs, any arc $(c_{ij}, x_i)$ will be alternatively denoted by the pair of variables $(x_i,x_j)$, where $x_j \in vars(c_{ij})$. That is, $x_j$ is the other variable involved in $c_{ij}$. An arc $(x_i,x_j)$ 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 arc $(x_i,x_j)$. Accordingly, $a$ is a support of $b$ on arc $(x_j,x_i)$. A problem is AC iff there are no empty domains and all arcs are AC. The application of AC on a problem results in the removal of all non-supported values from the domains of the variables. A *support check* (consistency check) is a test to find out if two values support each other. The *revision* of an arc $(x_i,x_j)$ using AC verifies if all values in $D(x_i)$ have supports in $D(x_j)$. A *DWO-revision* is one that causes a *DWO*. That is, it results in an empty domain.

In the following will use MAC (maintaining arc consistency) [21, 3] as our search algorithm. In MAC a problem is made arc consistent after every assignment, i.e. all values which are arc inconsistent given that assignment, are removed from the current domains of their variables. If during this process a domain wipeout (DWO) occurs, then the last value selected is removed from the current domain of its variable and a new value is assigned to the variable. If no new value exists then the algorithm backtracks.

## 2.1 Overview of existing variable ordering heuristics

The order in which variables are assigned by a backtracking search algorithm has been understood to be of prime importance for a long time. The first category of heuristics used for ordering variables were based on the initial structure of the network. They are called static or fixed variable ordering heuristics (SVOs) as they keep the same ordering of the variables during search. Examples of such heuristics are *lexico* where variables are ordered lexicographically, *min width* which chooses an ordering that minimizes the width of the constraint network [10], *min bandwidth* which minimizes the bandwidth of the constraint graph [25], and *min degree* (*deg*), where variables are ordered according to the initial size of their neighborhood [8].

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 search. The first well known dynamic heuristic, introduced by Haralick and Elliott, was *dom* [15]. This heuristic chooses the variable with the smallest remaining domain. The dynamic variation of *deg*, callled *ddeg* selects the variable with largest dynamic degree. That is, 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* [3, 23] 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 dynamic heuristics, based on measures such as the constrainedness of the problem, include the ones proposed in [12, 16]. 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* [11], *dom+ddeg* [6, 22] and *BZ3* [22].

Bessière et al. [2], 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 $\alpha(x_i)$ 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.

Recently, Boussemart et al. [5] proposed conflict-directed 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 extremely effective on a wide range of problems.

Grimes and Wallace [14] 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* with which they can uncover cases of *global contention*, i.e. contention that holds across the entire search space.

Inspired from 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. He proposes to characterize the impact of a decision by computing the Cartesian product of the domains before and after the considered decision.

Finally, Cambazard and Jussien [7] go 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:

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

## 3 Heuristics based on weighting constraints

As stated in the previous section, 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 maintained at every step, a DWO for a variable $x_i$ will be identified inside the *revise* procedure of Algorithm 1. In line 7, the weight of variable $x_i$ will be increased by one, each time a DWO is detected.

**Algorithm 1** REVICE-3$(x_i, x_j)$ : *boolean*

```
1: for each $a \in D(x_i)$ do
2:     if $\nexists b \in D(x_j)$ such that $c_{ij}(a, b)$ then
3:         delete $a$ from $D(x_i)$
4:     end if
5: end for
6: if $D(x_i) = \emptyset$ then
7:     $weight[c_{ij}] + +$
8: end if
9: return $D(x_i) \neq \emptyset$
```

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 is causes value deletions. However, the obtained heuristics did not demonstrate any advantage compared to dom/wdeg in practice [14]. To better illustrate our conjencture about the accuracy in assignning 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 variable 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 [14] 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 [24, 4]. 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*.

## 3.1 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 [5] using the following equation:

$$\alpha_{wdeg}(x_i) = \sum weight_{H1,H2,H3}[C] | vars(C) \ni x_i \wedge |FutVars(C)| > 1 \qquad (2)$$

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 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 [14]. The heuristics of [14] work as follows:

– constraint weights are increased by the size of the domain reduction leading to a DWO.
– 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. However, the weights 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 [14], 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 2 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.

## 3.2 Constraint weight aging

Most of the state-of-the-art SAT solvers like BerkMin [13] and Chaff [18], use the strategy of weight ”aging”. In such solvers, each variable is assigned 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

**Algorithm 2** $newRevice_H(x_i, x_j) : boolean$

```
1: for each $a \in D(x_i)$ do
2:     if $\nexists b \in D(x_j)$ 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 $D(x_i) = \emptyset$ then
8:     $\forall\ c_{ij} \in responsibleConstraint[x_i][D(x_i)]\ weight[c_{ij}] + +$ //H1 heuristic
       for each $a \in D(x_i)$
         $weight[resposibleConstraint[x_i][a]] + +$ // H2 heuristic
         $weight[responsibleConstraint[x_i][a]] + = 1/sizeof(D(x_i))$ // H3 heuristic
       end for
9: end if
10: return $D(x_i) \neq \emptyset$
```

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.

### 3.3 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 [4]. 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 seemed to be promising, its experimental behavior was not the expected. Experiments on a wide variety of real world problems showed that the variables that remains in the revision list after the detection of the first DWO are very ulikely 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 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 particpates 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.

## 4 Experiments and results

In this section we experimentally investigate the behavior of the new proposed variable ordering heuristics on several classes of real, academic and random problems. All benchmarks are taken from C. Lecoutre's web page[1]. We compare the heuristics with *dom/wdeg*, one the most efficient general purpose heuristics. Regarding the heuristics of Section 3.1, 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. In our tests we have used the following measures of performance: cpu time in seconds (t) and number of visited nodes (n). The solver we used applies d-way branching and lexicographic value ordering. It also employs 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

[1] http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/

aging, we have select to periodically decrease all constraint weights by a factor of 2, with the period set to 20 backtracks.

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 efficient *dom/wdeg* heuristic proposed in [1]. This heuristic selects first for revision the variable having the smallest ratio between current domain size and weighted degree among the variables in the revision list.

**Table 1.** Cpu times (t), and nodes (n) from frequency allocation problems. Best cpu time is in bold. The names of heuristics that employ aging are preceded by 'a'.

| Instance | | *dom/wdeg* | $H1$ | $H2$ | $H3$ | *adom/wdeg* | $aH1$ | $aH2$ | $aH3$ | *fullyAssigned* |
|---|---|---|---|---|---|---|---|---|---|---|
| scen2-f25 | t | 6,2 | 5,9 | 6,4 | 5,3 | **5,2** | 6,2 | 7,2 | **5,2** | 5,5 |
| | n | 1905 | 2031 | 2187 | 1724 | 1637 | 2030 | 2240 | 1724 | 1339 |
| scen3-f10 | t | 1,8 | 1,6 | **1,2** | 2 | 2 | 1,8 | **1,2** | 2,1 | 1,3 |
| | n | 567 | 562 | 470 | 623 | 595 | 574 | 470 | 623 | 463 |
| scen3-f11 | t | 5,1 | 5 | 4 | 3,8 | **3,7** | 6,5 | 4,2 | 4 | 12,7 |
| | n | 1025 | 990 | 799 | 737 | 671 | 1301 | 806 | 737 | 2158 |
| scen11 | t | 4,2 | **4** | 4,2 | 6,4 | 5,5 | 4,3 | 4,4 | 6,2 | 4,6 |
| | n | 830 | 830 | 831 | 1069 | 902 | 804 | 831 | 1013 | 807 |
| scen11-f11 | t | 3,5 | 3,7 | 4,1 | 3,5 | **2,8** | 4,1 | 4 | 3,7 | 6 |
| | n | 646 | 709 | 699 | 716 | 479 | 708 | 728 | 696 | 934 |
| scen11-f10 | t | 3,6 | 3,8 | 3,8 | 3,8 | **2,8** | 4 | 4,1 | 3,7 | 4,6 |
| | n | 674 | 714 | 743 | 685 | 467 | 717 | 770 | 726 | 722 |
| scen11-f9 | t | 10,2 | 10,5 | 11 | 10,3 | **9** | 11,9 | 12,3 | 10,9 | 13,7 |
| | n | 1664 | 1781 | 1764 | 1706 | 1346 | 1753 | 1770 | 1729 | 1733 |
| scen11-f8 | t | **17,2** | 17,3 | 17,9 | 18,2 | 20,4 | 18,6 | 19,2 | 17,6 | 21 |
| | n | 2695 | 2702 | 2758 | 2753 | 2830 | 2699 | 2780 | 2729 | 2766 |
| graph8-f10 | t | 25,3 | 75,6 | 18,4 | 25,2 | 19,9 | 26,8 | 26 | 25,3 | **7,3** |
| | n | 8386 | 22452 | 5618 | 8070 | 5506 | 8121 | 7907 | 8070 | 2321 |
| graph8-f11 | t | 3,7 | 2,3 | **0,9** | 4,7 | 2,8 | 2,1 | **0,9** | 4,6 | 1 |
| | n | 1025 | 542 | 144 | 1073 | 720 | 441 | 144 | 1073 | 137 |
| graph14-f27 | t | 41,1 | 39 | 28,3 | 41 | 93,7 | 76,7 | 193 | 41,3 | **24,8** |
| | n | 23818 | 21792 | 15721 | 23818 | 48512 | 34299 | 88846 | 23818 | 14315 |
| graph14-f28 | t | 50,1 | 51,5 | 19,3 | 321,2 | 48,5 | 2,7 | 61,6 | 135,5 | **2,3** |
| | n | 24944 | 24958 | 9577 | 162707 | 19353 | 982 | 29338 | 61165 | 857 |

Table 1 shows results from some real world RLFAP instances. 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 by removing some frequencies (denoted by f followed by a value). For example, scen11-f8 corresponds to the instance scen11 for which the 8 highest frequencies have been removed. Results from Table 1 show that the proposed heuristics display in general a slightly better performance than *dom/wdeg*, which achieves the best cpu time only on instance scen11-f8. The strategy of aging constraint weights increases the solver's efficiency in almost all cases. The "aged" version of the *dom/wdeg* heuristic achieves the best cpu time in five instances and heuristic *H2* in four instances. The "fully assigned" heuristic has a better performance on graph instances.

**Table 2.** Results from structured problems. Best cpu time is in bold.

| Instance | | $dom/wdeg$ | $H1$ | $H2$ | $H3$ | $adom/wdeg$ | $aH1$ | $aH2$ | $aH3$ | $fullyAssigned$ |
|---|---|---|---|---|---|---|---|---|---|---|
| driver-5c | t | 2,3 | 1,8 | 1,6 | 3,1 | 1,4 | 1,3 | 1,3 | 4,4 | **1** |
| | n | 1163 | 952 | 837 | 1654 | 805 | 658 | 700 | 955 | 448 |
| driver-8c | t | 13 | 7,2 | 5,4 | 12,7 | 4,6 | **2,9** | 4 | 3,8 | 4,5 |
| | n | 3895 | 2370 | 1769 | 3954 | 1042 | 952 | 1070 | 1101 | 891 |
| driver-8cc | t | 15,2 | 6,8 | 7,1 | 9 | 4,5 | 4,8 | **2,7** | 4,4 | 3,8 |
| | n | 4413 | 2043 | 2176 | 2659 | 1126 | 1235 | 867 | 1061 | 850 |
| driver-9 | t | 64,4 | 30,1 | **27,8** | 102,8 | 188,4 | 46,5 | 66,6 | 215,8 | 41,1 |
| | n | 10917 | 5272 | 4821 | 16169 | 15589 | 4998 | 6056 | 16873 | 4163 |
| langford2-9 | t | **40,4** | 43,3 | 48,3 | 47,7 | 60,9 | 52,3 | 53,1 | 51,7 | 60,5 |
| | n | 58203 | 70635 | 62502 | 62183 | 80483 | 79715 | 65950 | 64965 | 74328 |
| langford2-10 | t | **262,9** | 284,2 | 345,8 | 335,7 | 424,2 | 366,5 | 360,1 | 350 | 384,2 |
| | n | 306263 | 367680 | 321641 | 319070 | 421494 | 410258 | 333286 | 343049 | 385933 |
| langford3-10 | t | 7,9 | 13 | 3,5 | 5 | 2,9 | 21,7 | 2,3 | **1,9** | 20,1 |
| | n | 2439 | 4345 | 773 | 1548 | 709 | 6452 | 495 | 458 | 4768 |
| langford3-11 | t | 540,1 | **462,9** | 625,5 | 594,2 | 593 | 559,2 | 590,9 | 595,3 | 755,6 |
| | n | 96567 | 109834 | 102374 | 101758 | 99675 | 120757 | 109700 | 110155 | 161341 |
| langford4-10 | t | 42,8 | 42 | 43,5 | **30,7** | 60,5 | 46 | 50,9 | 49,9 | 51,3 |
| | n | 3846 | 4164 | 2978 | 2978 | 3661 | 4219 | 3001 | 2993 | 4240 |

In Table 2 we present results from structured instances belonging to benchmark classes *driver* and *langford*, while in Table 3 we give results from instances of the *graph coloring* problem. The results are similar to the ones from RLFAPs shown in Table 1. We must notice here that the advantage of heuristics that employ aging observed in Table 1 is not generic. Results in Table 2 show that aging constraint weights does not always lead to better performance. Especially the "aged" version of the *dom/wdeg* heuristic, which has a good performance in the RLFAP instances, does not achieve the best cpu time performance in any of the instances given in Table 2. We must also comment that in *driver* instances the "fully assigned" heuristic visits less nodes than all the other heuristics.

**Table 3.** Results from graph coloring problems. Best cpu time is in bold.

| Instance | | $dom/wdeg$ | $H1$ | $H2$ | $H3$ | $adom/wdeg$ | $aH1$ | $aH2$ | $H3$ | $fullyAssigned$ |
|---|---|---|---|---|---|---|---|---|---|---|
| myciel6-4 | t | **10,5** | 10,6 | 10,7 | 10,9 | 11,6 | 15 | 15,2 | 15,2 | 12,1 |
| | n | 10150 | 11273 | 11273 | 11122 | 11208 | 16396 | 16396 | 16163 | 7629 |
| myciel7-4 | t | **22,6** | 32 | 30,3 | 42,5 | 27,9 | 33,8 | 33,7 | 65,3 | 46,1 |
| | n | 6994 | 11059 | 11059 | 15396 | 8250 | 10995 | 10995 | 21926 | 7563 |
| 2-fullins-5-4 | t | 6,4 | 3 | 3 | 7,1 | 3 | **2,2** | **2,2** | 2,5 | 3,2 |
| | n | 722 | 263 | 263 | 771 | 288 | 179 | 179 | 243 | 111 |
| 3-fullins-5-5 | t | 52,5 | **24,9** | 25,3 | 37,2 | 46,8 | 35,4 | 35,5 | 47,8 | 58,6 |
| | n | 722 | 263 | 263 | 771 | 288 | 179 | 179 | 243 | 738 |
| fpsol2-i-1-5 | t | 23,4 | 19,8 | 19,5 | 22,9 | 18,4 | 15,8 | **15,5** | 16,9 | 38,5 |
| | n | 722 | 263 | 263 | 771 | 288 | 179 | 179 | 243 | 723 |
| huck-8 | t | 109,7 | 154 | 155,2 | 175,3 | 93 | **80,4** | 81,7 | 106 | 142,4 |
| | n | 220740 | 220740 | 220740 | 220740 | 221696 | 221052 | 221052 | 221029 | 220740 |

Finally, in Table 4 we present results from benchmark 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.** Results from random problems. Best cpu time is in bold.

| Instance | | $dom/wdeg$ | $H1$ | $H2$ | $H3$ | $adom/wdeg$ | $aH1$ | $aH2$ | $aH3$ | $fullyAssigned$ |
|---|---|---|---|---|---|---|---|---|---|---|
| frb30-15-1 | t | 8,8 | 12,4 | 14,6 | 13,2 | 14,5 | **1,8** | 5,7 | 13,5 | 14 |
| | n | 3659 | 5329 | 6457 | 5629 | 6175 | 766 | 2358 | 5872 | 4296 |
| frb30-15-2 | t | 84,3 | 67,9 | **39** | 88,2 | 123,8 | 113,1 | 117,7 | 117,1 | 41,8 |
| | n | 35822 | 27924 | 15538 | 38218 | 50066 | 48892 | 49756 | 49750 | 13343 |
| frb30-15-3 | t | 30,1 | **6,3** | 83,5 | 76,3 | 6,8 | 9 | 15,3 | 76,2 | 10,3 |
| | n | 11994 | 2572 | 33172 | 30689 | 2833 | 3817 | 6435 | 31826 | 2965 |
| frb30-15-4 | t | **12,7** | 73,5 | 76,9 | 48,7 | 16,1 | 77,4 | 74,2 | 19,7 | 90,2 |
| | n | 5316 | 31451 | 35337 | 21510 | 6790 | 33810 | 33783 | 8903 | 28310 |
| geo50-20-2 | t | 33,7 | 30,1 | 232,7 | 97,4 | 21,7 | **21,5** | 115,9 | 114,8 | 161,4 |
| | n | 8029 | 6856 | 61465 | 26483 | 5090 | 4763 | 27590 | 30950 | 24670 |
| geo50-20-10 | t | 9,4 | 5,3 | 31,8 | 8 | 11,9 | **4,5** | 47,9 | 6,9 | 10,3 |
| | n | 2134 | 1136 | 7383 | 1909 | 2804 | 930 | 11560 | 1702 | 1653 |
| geo50-20-11 | t | 19,2 | 25,6 | 12,9 | 12,9 | 3,1 | 4,3 | **2** | 65,5 | 7,3 |
| | n | 4374 | 5908 | 2808 | 2811 | 720 | 1009 | 420 | 16545 | 1148 |

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. However, further experiments (e.g. with non-binary problems) are required.

## 5 Conclusions

In this paper several new general purpose variable ordering heuristics are proposed. These 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 in a different way compared to the heuristics of Boussemart et al. 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 uppon detection of failure. The proposed conflict driven variable ordering heuristics have been tested over a wide range of benchmarks. Experimental results shows they are quite competitive to the existing ones and is some cases they can increase efficiency.

## References

1. T Balafoutis and K. Stergiou. Exploiting constraint weights for revision ordering in Arc Consistency Algorithms. In *Submitted to the ECAI-08 Workshop on Modeling and Solving Problems with Constraints*, 2008.
2. C. Bessière, A. Chmeiss, and L. Sais. Neighborhood-based variable ordering heuristics for the contraint satisfaction problem. In *Proceedings of CP'01*, pages 61–75, 2001.
3. C. Bessière and J.C. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?). In *In Proceedings of CP-1996*, pages 61–75, Cambridge MA, 1996.

4. 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.
5. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *In Proceedings of 16th European Conference on Artificial Intelligence (ECAI '04)*, 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. R. Dechter and I. Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In *In Proceedings of IJCAI'89*, pages 271–277, 1989.
9. N. Dershowitz, Z. Hanna, and A. Nadel. A Clause-Based Heuristic for SAT Solver. In *Proceedings of SAT-2005*, pages 46–60, 2005.
10. E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.
11. D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *In Proceedings of IJCAI'95*, pages 572–578, 1995.
12. 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 CP'96*, pages 179–193, 1996.
13. E. Goldberg and Y. Novikov. BerkMin: a Fast and Robust Sat-Solver. In *Proceedings of DATE'02*, pages 142–149, 2002.
14. D. Grimes and R.J. Wallace. Sampling strategies and variable selection in weighted degree heuristics. In *In Proceedings of CP 2007*, pages 831–838, 2007.
15. R.M. Haralick and Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–314, 1980.
16. M.C. Horsch and W.S. Havens. An empirical study of probabilistic arc consisteny as a variable ordering heuristic. In *Proceedings of CP'00*, pages 525–530, 2000.
17. Y.S. Mahajan, Z. Fu, and S. Malik. ZChaff2004: An Efficient SAT Solver. In *Proceedings of SAT-2004*, pages 360–375, 2004.
18. M. Moskewicz, C. Madigan, and S. Malik. Chaff: Engineering an efficient sat solver. In *In Proceedings of Design Automation Conference*, pages 530–535, 2001.
19. P. Refalo. Impact-based search strategies for constraint programming. In *In Proceedings of CP 2004*, pages 556–571, 2004.
20. L. Ryan. Efficient Algorithms for Clause-Learning SAT Solvers. Master's thesis, Simon Faser University, 2004.
21. D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *In Proceedings of CP '94*, pages 10–20, 1994.
22. B.M. Smith. The brelaz heuristic and optimal static orderings. In *In Proceedings of CP'99*, pages 405–418, 1999.
23. B.M. Smith and S.A. Grant. Trying harder to fail first. In *In Proceedings of ECAI'98*, pages 249–253, 1998.
24. R. Wallace and E. Freuder. Ordering heuristics for arc consistency algorithms. In *AI/GI/VI*, pages 163–169, Vancouver, British Columbia, Canada, 1992.
25. R. Zabih. Some applications of graph bandwith to constraint satisfaction problems. In *In Proceedings of AAAI'90*, pages 46–51, 1990.