Exploiting Constraint Weights for Revision Ordering in Arc Consistency Algorithms

Thanasis Balafoutis and **Kostas Stergiou**
Department of Information & Communication Systems Engineering
University of the Aegean, Greece
email: {abalafoutis,konsterg}@aegean.gr

Abstract. Coarse grained arc consistency algorithms, like AC-3, 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. 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. Recently, Boussemart et al. proposed novel variable ordering heuristics that exploit information about failures gathered throughout search and recorded in the form of constraint weights. 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 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.

1 Introduction

Among the plethora of algorithms that have been devised to solve CSPs, the look-ahead algorithm termed MAC (maintaining arc consistency) [16, 2] is considered as one of the most efficient. As MAC applies arc consistency (AC) on the problem after every variable assignment, speeding up the process of AC application has received a lot of attention in the literature. The numerous AC algorithms that have been proposed can be classified into *coarse grained* and *fine grained*. Typically, coarse grained algorithms like AC-3 [12] and its extensions (e.g. AC2001/3.1 [3] and AC-3_d [7]) apply successive revisions of arcs or, depending on the implementation, variables [13]. On the other hand, fine grained algorithms like AC-2004 [14] and AC-2007 [1] use various data structures to apply successive revisions of variable-value-constraint triplets. 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. Further to this, some of the extensions to AC-3 achieve optimal worst-case complexity while preserving the simplicity of implementation and good average case behavior.

It is well known that the way in which the list of revisions is implemented and manipulated is an important point regarding the efficiency of coarse grained AC algorithms. In a recent empirical investigation Boussemart et al. [4] showed that a variable-oriented implementation of AC-3, as proposed in [13], usually outperforms the standard arc-oriented implementation of [12] and the constraint-oriented implementation of [4]. Perhaps more significantly, the order in which the elements of the revision list are processed, in any implementation, can have a notable effect on the number of constraint checks and list insertion/removal operations. Since MAC applies AC thousands or even millions of times during search, any savings in checks and list operations can be reflected on the overall cpu time efficiency. Having recognized this, Wallace and Freuder proposed a number of revision ordering heuristics aiming at speeding up AC processing as early as 1992. Since then this issue has been further investigated and alternative heuristics have been proposed [8, 7, 11, 4]. All the proposed 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. However, it has to be noted that even the most successful revision heuristic of variable-oriented propagation only offers a 25% speed-up compared to a *fifo* implementation of the revision list [4].

In recent years, powerful variable ordering heuristics have been proposed and their integration with MAC has led to significant speed-ups of existing solvers. The *conflict-driven* weighted degree (*wdeg*) heuristics of Boussemart et al. are designed to enhance variable selection by incorporating knowledge gained during search, in particular knowledge derived from failures [5]. These heuristics work as follows. All constraints are given an initial weight of 1. During search the weight of a constraint is incremented by 1 every time the constraint causes a domain wipe-out (DWO) during constraint

propagation. The weighted degree (*wdeg*) of a variable is then the sum of the weights of the constraints that include this variable and at least another unassigned variable. The weights are continuously updated during search by using information learnt from previous failures. The basic *wdeg* heuristic selects the variable having the largest weighted degree. In addition to the basic *wdeg* heuristic, combining weighted degree and domain size yields a heuristic that selects the variable with the smallest ratio of current domain size to current weighted degree (*dom/wdeg*). The advantage that these heuristics offer is that they use previous search states as guidance, while older standard heuristics either use the initial state or the current state only.

In this paper we show how information about constraint weights can be exploited not only to perform variable selection, but also to efficiently order the revision list when AC is maintained during search. We investigate several new constraint weight based approaches to ordering the revision list in all the alternative implementations of AC-3: arc-oriented, variable-oriented and constraint-oriented. Experimental results from various random, academic and real world problems show that some of the proposed heuristics, when used in conjunction with a conflict-driven variable ordering heuristic such as dom/wdeg, demonstrate a measurable improvement in constraint checks compared to the most efficient existing revision ordering heuristic.

Notably, the new revision heuristics can not only reduce the numbers of constraint checks and list operations, but also cut down the size of the explored search tree by focusing search on more relevant variables. Due to this, in the variable-oriented implementation of AC-3, which is the most efficient among the three alternatives, the new heuristics can offer significant savings in cpu times. This opens up interesting directions for future work since, apart from an implementation tool, revision ordering heuristics can be viewed as methods to have a really important impact on the search process.

The rest of the paper is organized as follows. Section 2 gives the necessary definitions and notation and briefly describes the three alternative implementations of AC-3. Section 3 summarizes existing work on revision ordering heuristics for constraint propagation. In Section 4 we propose new revision ordering heuristics based on constraint weights. In Section 5 we experimentally compare the proposed heuristics to the best existing revision heuristics on a variety of problems. Conclusions are presented in Section 6.

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 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. 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 revision 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 $D(x_j)$. 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 *DWO*. That is, it results in an empty domain.

In the following, we will use the basic coarse grained algorithm to establish arc consistency, namely, AC-3. This does not limit the generality of the proposed heuristics as they can be easily integrated into any coarse grained AC algorithm. In the reported experiments we use MAC as our search algorithm and the *dom/wdeg* heuristic for dynamic variable ordering.

2.1 AC-3 variants

The AC-3 arc consistency algorithm can be implemented using a variety of propagation schemes. We recall here the three variants, as presented in [4], which respectively correspond to algorithms with an arc-oriented, variable-oriented and 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 1 depicts the main procedure. As explained, 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. 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.

The variable-oriented propagation scheme was proposed by McGregor [13] and later studied in [6]. Instead of keeping arcs in the revision list, this variant of AC-3 keeps variables. The main procedure is depicted in Algorithm 3. 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

Algorithm 1 ARC-ORIENTED AC3

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

Algorithm 2 REVISE-3(x_i, x_j)

```
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
```

Algorithm 3 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 needsNotBeRevised(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 4 needsNotBeRevised(c_{ij}, x_i) : boolean

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

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).

Function *needsNotBeRevised* given in Algorithm 4, 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 5. This algorithm is an analogue to Algorithm 3.

Algorithm 5 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 needsNotBeRevised(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
```

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 arc (c_{ij}, x_j) is fruitful, then the reinsertion of the constraint c_{ij} in the list is needed. As in variable-oriented scheme, the same counters are also used here to avoid useless revisions.

3 Related work

Revision ordering heuristics 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 revision ordering heuristics that can be used with the arc-oriented variant of AC3 [17]. 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. Vari-

ables 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 [17].

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 [17] and confirmed in [4], the (*deg down*) heuristic does not offer any improvement.

Gent et al. [8] 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. performed an empirical investigation of the heuristics of [17] with respect to the different variants (arc, variable and constraint) of AC-3 [4]. 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 [17] 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.

4 Revision ordering heuristics based on constraint weights

The heuristics described in the previous section, 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 mainly due to the reduction in list additions and constraint checks. A key principle that can also have a positive effect on the performance is the "fail-first principle" of Haralick and Elliot [10] which states that "to succeed, try first where you are most likely to fail". 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 earlier to a *DWO*.

To apply the "fail-first principle" in revision ordering heuristics, we must use some metric to compute which arc (or variable) in the AC revision list is the most probable to cause failure. Until now, constraint weights have only been used for variable selection. In our proposed revision ordering heuristics, we use information about constraint weights as a metric to order the AC revision list. These heuristics can efficiently be used in conjunction with conflict-driven variable ordering heuristics in order 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 significantly be reduced. Moreover, with such a design we may be able to avoid many *redundant revisions*.

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 list was organized during AC was impossible to 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 additions and constraint checks.

However, when a conflict-driven variable ordering heuristic like *wdeg* or *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 1 Assume we are using MAC with an arc-oriented implementation of AC-3 to solve a CSP (X, D, C). Also assume that a conflict-driven variable ordering heuristic (e.g. dom/wdeg) is used, and that at some point during search the following AC revision list is formed: Q=$\{(c_{12}, x_1)$, (c_{34}, x_3), $(c_{56}, x_5)\}$. Suppose that (c_{12}, x_1) and (c_{56}, x_5) can both lead to a DWO if they are selected first from the list. If a revision ordering heuristic R_1 selects (c_{12}, x_1) first then the DWO of 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 (c_{56}, x_5) first then the DWO of x_5 will be detected but this time the weight of a different constraint (c_{56}) will increased by 1. 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.

We now describe a number of new revision ordering heuristics for all three AC-3 variants. It is easy to see that all these 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.

- *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.
- *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.
- *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.

- *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 1.

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.

- *wdeg*: selects the variable having the highest weighted degree *wdeg* among all variables in Q.
- *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 3. 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 *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 5. 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 experimented with such heuristics.

5 Experiments and results

In this section we experimentally investigate the behavior of the new revision ordering heuristics proposed above on several classes of real, toy and random problems[1]. In our experiments we included both satisfiable and unsatisfiable instances. We only give 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 [4].

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). In our tests we have used the following measures of performance: cpu time in seconds (t), number of visited nodes (n) and number of constraint checks (c). The solver we used applies d-way branching, dom/wdeg for variable ordering 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.

Tables 1 and 2 show results from some real-world RLFAP instances. In the arc-oriented implementation of AC-3 (Table 1), heuristics *wcon*, mainly, and *dom/wcon*, to a lesser extent, decrease the number of constraint checks compared to *dom*. However, the decrease is not substantial and is rarely translated into a decrease in cpu times. The notable speed-up observed for problem s11-f6 is mainly attributed to the reduction in node visits offered by the two new heuristics. *wdeg* and *dom/wdeg* are less competitive, indicating that information about the variables involved in arcs is less important compared to information about constraints.

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

Table 1. Cpu times (t), constraint checks (c) and nodes (n) from frequency allocation problems (hard instances) using arc and variable oriented propagation. The s prefix stands for scen instances. Best cpu time is in bold.

		ARC ORIENTED					
Inst.		*queue*	*dom*	*wcon*	*wdeg*	*d/wdeg*	*d/wcon*
s11-f9	t	17,1	**11,7**	13,3	13,5	17,3	12,9
	c	18M	13,9M	9,5M	15M	15,1M	12,1M
	n	1760	1688	1689	1671	1681	1697
s11-f8	t	34,2	**18,5**	20,5	20	26	21,4
	c	33,5M	21,1M	13,8M	21,7M	23,7M	19,8M
	n	2902	2679	2699	2746	2682	2822
s11-f7	t	234,6	**133,5**	154,9	241,7	187,5	297,2
	c	193,1M	114,7M	92,5M	202,5M	147,6M	215,9M
	n	25830	21571	23334	30185	22427	43695
s11-f6	t	518	423,9	**281,9**	492,4	760,8	361,2
	c	347M	336,9M	166,1M	372,1M	536,3M	261M
	n	68225	73235	42541	71918	99874	52512
s11-f5	t	2571	2102	2792	2947	2641	**2088**
	c	1,793G	1,539G	1,509G	2,107G	1,868G	1,414G
	n	310,4M	318,3M	440,2M	378,1M	272,3M	274,3M
s11-f4	t	10220	**7084**	7523	9464	11409	9543
	c	7,150G	5,075G	3,812G	6,490G	7,706G	6,186G
	n	1,103G	1,038G	1,116G	1,219G	1,245G	1,152G

The variable-oriented implementation (Table 2) is clearly more efficient than the arc-oriented one. This confirms the results of [4]. Concerning this implementation, heuristic *dom/wdeg* outperforms *dom* and *queue* both in node visits and checks. Importantly, these savings are reflected on notable cpu time gains making the variable-oriented *dom/wdeg* the overall winner. Results also show that as the instances becomes harder, the efficiency of *dom/wdeg* heuristic compared to *dom* increases. The variable-oriented *wdeg* heuristic in most cases outperforms *dom* but is clearly less efficient than *dom/wdeg*.

Table 2. Cpu times (t), constraint checks (c) and nodes (n) from frequency allocation problems (hard instances) using arc and variable oriented propagation. The s prefix stands for scen instances. Best cpu time is in bold.

		VARIABLE ORIENTED			
Inst.		*queue*	*dom*	*wdeg*	*d/wdeg*
s11-f9	t	16,2	9,3	9,9	**9**
	c	16,3M	8,2M	9,3M	7,9M
	n	1767	1635	1677	1664
s11-f8	t	30,1	15,8	16,9	**15,2**
	c	30,3M	12,4M	14,7M	12,1M
	n	2879	2697	2679	2695
s11-f7	t	187,3	144,1	140,8	**98,6**
	c	139,1M	84,1M	113,4M	59,5M
	n	26139	27485	21332	19298
s11-f6	t	286	356,3	395,8	**245,6**
	c	220,3M	189,4M	297,6M	138,6M
	n	36331	68391	60919	46174
s11-f5	t	2254	2966	1840	**1579**
	c	1,492G	1,522G	1,081G	832,8M
	n	327,9M	582,1M	278,9M	292,6M
s11-f4	t	12729	10806	8648	**6077**
	c	8,676G	5,405G	4,975G	3,110G
	n	1,682G	1,982G	1,374G	1,048G

In Table 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 *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. *dom/wdeg* cuts down the number of node visits by more than 5 times resulting in a similar speed-up. It is interesting that *dom/wdeg* is considerably more efficient than *wdeg* and *dom*, indicating that information about domain size or weighted degree alone is not sufficient to efficiently order the revision list.

Table 3. Cpu times (t), constraint checks (c) and nodes (n) from structured problems using variable oriented propagation. Best cpu time is in bold.

Instance		*queue*	*dom*	*wdeg*	*d/wdeg*
langford-2-9	t	49,4	42,7	55	**42,1**
	c	71,7M	58,8M	71,9M	58,7M
	n	71729	58897	71907	59095
langford-2-10	t	450	392,4	381,7	**309,9**
	c	241,8M	204,1M	198M	142,4M
	n	497,359	410819	381161	305480
langford-3-11	t	633,5	590,9	768,6	**467,9**
	c	294M	253,8M	337,3M	184,7M
	n	119036	99619	152063	96567
langford-4-10	t	76,3	52,6	90,6	**37,5**
	c	37,8M	23,9M	42,9M	15,6M
	n	5253	4352	5759	3896
driverlogw-08c	t	26,4	13,4	**13,1**	13,3
	c	15M	6,2M	7,9M	6,5M
	n	7576	4451	2870	3895
driverlogw-09	t	206,1	374,5	315,6	**63,9**
	c	109M	181M	146,5M	28,4M
	n	30720	60084	46188	10917

Finally, in Table 4 we present results from benchmark random problems. Here, there is a large diversity in the results. All heuristics seems to lack robustness and there is no clear winner. The constraint weight based heuristics can be up to one order of magnitude faster than *dom* (instance geo50-20-d4-75-2), 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 possible explanation for the diversity in the performance of the heuristics on 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 *dom/wdeg* will give priority for revision to variables that are involved in hard subproblems and hence will carry out DWO-revisions faster. This 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. 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 direct search tree to different parts of the search space with unpredictable results. Note that for structured problems only a 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.

6 Conclusions

In this paper we showed how information about constraint weights can be exploited not only to perform variable selection, but also to order the revision list when arc consistency is applied during search. As a result, we proposed a number of simple and lightweight revision ordering heuristics for coarse grained arc consistency algorithms. The proposed heuristics order the revision list by trying to carry out possible DWO-revisions as soon as possible. Importantly, the heuristics can not only reduce the numbers of constraint checks and list operations but they can also have a significant effect on search. Among the heuristic we experimented with, the one with best performance was *dom/wdeg* in the variable-oriented implementation of arc consistency. Experimental results from various domains displayed the potential of the proposed heuristics.

Table 4. Cpu times (t), constraint checks (c) and nodes (n) from random problems using variable oriented propagation. Best cpu time is in bold.

Instance		*queue*	*dom*	*wdeg*	*d/wdeg*
frb30-15-1	t	26	19	26,7	**12,8**
	c	11,9M	8M	11,8M	5,4M
	n	6142	5648	6058	3659
frb30-15-2	t	69,4	**27,1**	108,3	86,6
	c	32,9M	15,7M	64,8M	49,6M
	n	18099	11617	36818	35822
frb35-17-1	t	114,6	176,5	**107,5**	228,6
	c	67,6M	103,6M	64,6M	130,2M
	n	27213	59585	28062	74098
rand-2-30-15	t	1130,1	**67,8**	89,3	98,5
	c	82,4M	38,2M	52,2M	56,2M
	n	42056	29056	29563	42115
geo50-20-d4-75-2	t	213,5	366,1	**31,7**	36
	c	138M	223,3M	20,3M	20,7M
	n	30747	88111	5468	8029

As future work, it would be interesting to study the interaction of revision ordering heuristics with other modern variable ordering heuristics apart from dom/wdeg. For example, the impact-based heuristics of [15] and the explanation-based heuristics of [9].

REFERENCES

[1] C. Bessière, E.C. Freuder, and J.C. Régin, 'Using Inference to Reduce Arc Consistency Computation', in *Proceedings of IJCAI'95*, pp. 592–599, (1995).

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

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

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

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

[8] I.P. Gent, E. MacIntyre, P. Prosser, P. Shaw, and T. Walsh, 'The constraindedness of arc cosistency', in *In Proceedings of CP-97*, pp. 327–340, (1997).

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

[10] R.M. Haralick and Elliot, 'Increasing tree search efficiency for constraint satisfaction problems', *Artificial Intelligence*, **14**, 263–313, (1980).

[11] C. Lecoutre, F. Boussemart, and F. Hemery, 'Exploiting multidirectionality in coarse-grained arc cosistency algorithms', in *In Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-2003)*, pp. 480–494, (2003).
[12] A. Mackworth, 'Consistency in networks of relations', *Artificial Intelligence*, **8**, 99–118, (1977).
[13] J.J. McGregor, 'Relational consistency algorithms anf their applications in finding subgraph and graph isomorphism', *Information Science*, **19**, 229–250, (1979).
[14] R. Mohr and T. Henderson, 'Arc and Path Consistency Revisited', *Artificial Intelligence*, **28**, 225–233, (1986).
[15] P. Refalo, 'Impact-based search strategies for constraint programming', in *In Proceedings of CP 2004*, pp. 556–571, (2004).
[16] D. Sabin and E.C. Freuder, 'Contradicting conventional wisdom in constraint satisfaction', in *In Proceedings of CP '94*, pp. 10–20, (1994).
[17] R. Wallace and E. Freuder, 'Ordering heuristics for arc consistency algorithms', in *AI/GI/VI*, pp. 163–169, Vancouver, British Columbia, Canada, (1992).