Ordering Heuristics for Arc Consistency Algorithms*

Richard J. Wallace and Eugene C. Freuder
Department of Computer Science
University of New Hampshire
Kingsbury Hall M208, College Road
Durham, NH 03824 USA
e-mail: ecfrjw@cs.unh.edu

Abstract

Arc consistency algorithms are used in solving constraint satisfaction problems and are important in constraint logic programming languages. Search order heuristics for arc consistency algorithms significantly enhance the efficiency of their implementation. In this paper we propose and evaluate several ordering heuristics. Care is taken with experimental design, involving random problems, and statistical evaluation of results. A heuristic is identified which yields about 50% savings on average, using the standard measure of consistency pair checks, with reasonable heuristic computation cost.

1 Introduction

Arc consistency insures that any two mutually constraining problem variables are mutually consistent: given a value for one, we can find a value for the other which satisfies the constraint between them. The constraint specifies which pairs of values can be simultaneously assumed by the pair of variables.

Arc consistency is a fundamental concept in constraint-based reasoning [Mackworth, 1987] and has played a significant role in constraint logic programming [Dincbas et al., 1988] [Van Hentenryck, 1989]. Arc consistency can also be used to answer temporal reasoning questions [Meiri, 1991]. Various forms of arc consistency have been utilized in various roles to solve constraint satisfaction problems (CSPs), i.e. to find values, for a set of variables, that satisfy a set of constraints. Many artificial intelligence problems, from scene analysis to scheduling, have been viewed as CSPs.

For some classes of problems arc consistency processing alone essentially finds a solution [Freuder, 1982] [Deville and Van Hentenryck, 1991]. Some algorithms repeatedly employ generalized forms of arc consistency to find solutions to arbitrary problems [Mackworth, 1977a] [Freuder, 1978]. Arc consistency can be used for “preprocessing” before further search, and may simplify a problem to the point where little if any subsequent search effort is needed, especially when initial “boundary conditions” are known [Waltz, 1975]. Full or partial arc consistency can be combined with backtrack search in “hybrid” algorithms [Nadel, 1989].

While there has been some indication in the literature that ordering heuristics can improve the performance of the relaxation (constraint propagation) algorithms that achieve arc consistency [Waltz, 1975], these heuristics have not received systematic study. (This is to be contrasted with the considerable attention devoted to ordering heuristics for constraint satisfaction backtrack search.)

This paper initiates such a systematic study. We identify factors that determine the efficiency of constraint propagation. Probabilistic arguments suggest the relationship between these factors and certain easily measurable problem characteristics. Several ordering heuristics are proposed based on these problem characteristics. Careful testing on random problems verifies our expectations of increased efficiency.

Our best heuristics halve the number of constraint checks (a standard measure of performance). The worst-case complexity of the overhead involved in computing the ordering is linear in the number of constraints, and does not add to the overall worst-case complexity of the arc consistency algorithm. Computation time data confirms the efficacy of the heuristic.

Most of our experiments are based on the standard AC-3 arc consistency algorithm [Mackworth, 1977b]. AC-4 [Mohr and Henderson, 1986] has a better worst-case bound than AC-3 (indeed its worst-case bound is optimal), but incurs the time/space costs of building and maintaining more elaborate data structures. AC-4 is also subject to ordering improvement, and we demonstrate this experimentally as well; however, in our tests AC-4 was very much less efficient than AC-3.

We would expect ordering heuristics to improve some partial arc consistency algorithms [Haralick and Elliott, 1980] [Dechter and Pearl, 1988] [Nadel, 1989] [Freuder and Wallace, 1991]. The ideas should be generalizable to higher level consistency algorithms [Montanari, 1974] [Mackworth, 1977b] [Freuder, 1978] [Cooper, 1989].

Section 2 reviews basic concepts. Section 3 discusses

---

*This material is based on work supported by the National Science Foundation under Grant No. 1RI-8913040. The government has certain rights to this material. Some of this work was done while the second author was a Visiting Scientist at the MIT Artificial Intelligence Laboratory.
factors that determine efficiency of relaxation for any problem and Section 4 discusses why some orderings would be expected to enhance performance a priori. Sections 5 and 6 describe methods and results, respectively, of experiments with random constraint satisfaction problems designed to assess the effectiveness of orderings based on these heuristics. Section 7 considers the effort required to obtain good orderings with specific heuristics. Section 8 summarizes the conclusions.

2 Basic Concepts

A constraint satisfaction problem (CSP) involves a set of $n$ variables, $v_i$, each having a domain of values, $d_i$ that it can assume. In addition, the problem is subject to some number of binary constraints, $C_{ij}$, each of which is a subset of the Cartesian product of two domains, $d_i \times d_j$. A binary constraint specifies which pairs of values can be simultaneously assumed by the pair of variables. (Only binary constraints are considered here; higher-order CSPs can also be represented by binary CSPs [Rossi et al., 1989].) A CSP is associated with a constraint graph, where nodes represent variables and arcs represent constraints (Figure 2).

AC-3 involves a sequence of tests between pairs of constrained variables, $v_i$ and $v_j$: we say that $v_i$ is relaxed against $v_j$. Specifically, values in $v_i$ are checked against the constraint between $v_i$ and $v_j$ to see if they are supported, i.e. are consistent with at least one value in the domain of $v_j$; unsupported values are deleted. The AC-3 algorithm is shown in Figure 1. All ordered pairs of constrained variables are first put in list $L$. Each pair, $(v_i, v_j)$, is removed and $v_i$ is relaxed against $v_j$. When values are deleted, pairs may need to be added to $L$ to determine if these deletions lead to further deletions.

Initialize $L$ to $\{ (v_i, v_j) \}$

a constraint exists between $v_i$ and $v_j$.

While $L$ is not empty

Select and remove $(v_i, v_j)$ from $L$.

Relax $v_i$ against $v_j$.

If relaxation removes any values from $v_i$,

add to $L$ any pairs $(v_k, v_l)$, $k \neq j$, such that there is a constraint between $v_k$ and $v_l$ and $(v_k, v_l)$ is not already present in $L$.

3 Factors That Determine Efficiency of Relaxation

Differences in the efficiency of relaxation that depend on ordering can be ascribed to two factors: (i) if a value is deleted from $d_i$ when $v_i$ is relaxed against $v_j$, less work is performed if other constraints that include $v_i$ are tested after this restriction rather than before (minimizing the number of tests before deletion), (ii) work is reduced if pairs with nodes adjacent to the node relaxed are already on the list (minimizing the number of list additions, or relaxations). Both factors are consistent with the principle that, in good orderings, domain values are removed as quickly as possible (the “ASAP principle”). This idea was first stated by [Waltz, 1975] in his discussion of filtering descriptions of line drawings, viz., “The basic heuristic for speeding up the program is to eliminate as many possibilities as early as possible” (p. 60). However, it is not so much a heuristic as a characteristic of efficient performance, since conformity to this principle cannot be determined in advance.

![Figure 2: CSP illustrating relaxation principles.](image)

One aspect of CSP structure that can affect efficiency via the order of relaxation is the presence of sequential dependencies in the pattern of domain support. For example, suppose the values within domain $d_j$ that support a value in $d_i$ are not supported by values in another domain, $d_k$. In this case, relaxing $v_j$ against $v_k$ removes the values in $d_j$ that supported the values in $d_i$, and, if $v_i$ is then relaxed against $v_j$, the latter values are removed the first time this variable pair is tested. On the other hand, if $v_i$ is relaxed against $v_j$ before the latter is relaxed against $v_k$, these values cannot be removed, and more values are retained that will eventually be eliminated. Since failure to observe such dependencies will result in pairs being put back on the list after they have been examined, this is a special case of minimizing additions to the list of variable pairs. (A specific reference to sequential dependencies is made in [Gaschnig, 1974].)

These principles are illustrated with a simple example (Figures 2-3). The CSP in this case has three vari-
ables and two constraints, so its constraint graph is a
tree. Domains have the same number of values and con-
straints the same number of acceptable pairs. Figure 2
also includes a restriction diagram, showing all values
that could possibly be deleted by relaxation and any de-
pendencies between them. Variable pairs with at least
one domain subject to restriction are shown on the left,
and each column to the right includes all values with
the same depth of dependency (subscripted with vari-
able names). The diagram, therefore, gives a “parallel
view” of relaxation. Dependency relations between spe-
cific values are shown by arrows.

<table>
<thead>
<tr>
<th>start</th>
<th>(abc)</th>
<th>(def)</th>
<th>(ghi)</th>
</tr>
</thead>
<tbody>
<tr>
<td>good ordering (conforms to ASAP rule)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 ag 1 remove a,b</td>
<td>7 cks</td>
<td>(c)</td>
<td>(def)</td>
</tr>
<tr>
<td>2 ag 0 remove g</td>
<td>3</td>
<td>(c)</td>
<td>(def)</td>
</tr>
<tr>
<td>0 ag 2 —</td>
<td>—</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1 ag 0 —</td>
<td>—</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>fails to minimize pair checks (no list additions)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 ag 0 —</td>
<td>9 cks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 ag 1 remove a,b</td>
<td>7</td>
<td>(c)</td>
<td>(def)</td>
</tr>
<tr>
<td>2 ag 0 remove g</td>
<td>3</td>
<td>(c)</td>
<td>(def)</td>
</tr>
<tr>
<td>0 ag 2 —</td>
<td>—</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>fails to minimize additions (no sequent depend. violations)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 ag 0 —</td>
<td>9 cks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 ag 2 remove a</td>
<td>6</td>
<td>(bc)</td>
<td>(def)</td>
</tr>
<tr>
<td>0 ag 1 remove b</td>
<td>4</td>
<td>(bc)</td>
<td>(def)</td>
</tr>
<tr>
<td>2 ag 0 remove g</td>
<td>3</td>
<td>(bc)</td>
<td>(def)</td>
</tr>
<tr>
<td>1 ag 0 —</td>
<td>—</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>fails to respect sequential dependency</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 ag 0 —</td>
<td>8 cks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 ag 1 remove a,b</td>
<td>7</td>
<td>(c)</td>
<td>(def)</td>
</tr>
<tr>
<td>0 ag 2 —</td>
<td>—</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1 ag 0 —</td>
<td>—</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2 ag 0 remove g</td>
<td>3</td>
<td>(c)</td>
<td>(def)</td>
</tr>
</tbody>
</table>

Figure 3: Relaxation with different orderings.

The course of relaxation of this problem for different
AC-3 list orderings is shown in Figure 3. Constraint
checks are shown on successive rows, including the vari-
able relaxed (i ag j indicates that v_i is relaxed against
v_j), the values (of d_i) removed, the number of value pairs
that were checked (assuming a lexicographic order), and
the current domains after each instance of domain re-
striction. The first example shows an ordering consistent
with the ASAP principle: when v_0 is relaxed against v_1,
the pair (0 2), which includes an adjacent node is still
on the list, the dependency between b in d_0 and g in d_2
is respected, and v_0 is relaxed against v_2 and v_1 against
v_0 after their domains have been reduced. In the sec-
ond example v_1 is relaxed against v_0 at the beginning,
when their domains have their original sizes, so more pair
checks are required. In the third example, v_0 is relaxed
against v_2 when the pair (1 0) has already been removed
from the list; since a value is deleted from the domain of
v_0, (1 0) must be put back on. In the fourth example,
the sequential dependency between b in d_0 and g in d_2
is violated; nothing is deleted from d_2 when v_2 is first
relaxed against v_0, and the lack of support for g in the
former domain must be discovered in a second test.

4 Rationale for Order Heuristics

Many ordering heuristics can be devised, based on three
major features of constraint satisfaction problems: (i) the
number of acceptable pairs in each constraint (the con-
straint size or satisfiability [Nadel, 1988]), (ii) the
number of values in each domain, (iii) the number of bi-
nary constraints that each variable participates in, equal
to the degree of the node of that variable in the con-
straint graph. Simple examples of heuristics that might
be expected to improve the efficiency of relaxation are:
(i) ordering the list of variable pairs by increasing rela-
tive satisfiability, i.e. the proportion of possible pairs
that are acceptable, (ii) ordering by increasing size of
the domain of the variable relaxed against, (iii) ordering
by descending degree of node of the variable relaxed.

However, it is important to note that the principles of
efficient relaxation described in Section 3 do not depend
on differences in such features. Thus, in the example
given there, domain size and satisfiability are both con-
stant. (It is easy to construct similar examples where
degree of node does not vary.) Conversely, variation in
these features does not guarantee that a particular heuris-
tic will enhance efficiency. For example, a smaller
domain size does not itself imply that relaxation is more
likely, since the acceptable pairs can include all domain
values. This means that the case for ordering heuristics
depends on probabilistic arguments. In other words, we
must show that a given ordering can be expected to en-
hance efficiency, if we can make some assumptions about
expected failure of domain support.

For any binary constraint, the range of possible sati-
sfiables, from 0 to the product of the participating
domains, can be divided into three parts, in which relax-
ation is inevitable, possible, or impossible, respectively.
These subranges are:

(inevitable) 0 to d_max - 1
(possible) d_max to [(d_max * d_min) - d_min]
(impossible) [(d_max * d_min) - d_min + 1] to d_max * d_min

The basis for this division is not hard to see: (i) if the
constraint size is less than the size of the largest domain,
not all values can be supported, (ii) if at least one value of
the larger domain is not supported, the largest possible
constraint size cannot be larger than the product of the
smaller domain size and one less than the larger one.
If v_i is relaxed against v_j, an alternative description of
these intervals is:

(inevitable) 0 to d_i - 1
(possible) d_i to [(d_i * d_j) - d_j]
(impossible) [(d_i * d_j) - d_j + 1] to d_i * d_j

In accordance with the ASAP rule, an ideal ordering
heuristic would insure that any pairs for which relaxation
is more likely would be put at the beginning of the list. In
particular, those pairs for which relaxation is inevitable
would be tested first. Any reordering after a domain
restriction should also have this property.
Since, for any domain pair, relaxation is at least as likely if the satisfiability is smaller, ordering based on increasing satisfiability is an obvious candidate heuristic. Two variants of this idea are ordering by relative satisfiability and by satisfiability per se, which does not discount the effects of domain size. A third heuristic of this type is suggested by the following argument. Suppose that associated with each pair of values is a probability, \( p \), that it will be included in the set of acceptable pairs. Then the following relation holds between the sizes of the domains, \( d_i \) and \( d_j \) and the expected size of the set of constraint pairs based on these domains, \( |C_{ij}| \):

\[
|C_{ij}| = |d_j| \times |d_i| \times p
\]

In this case, if \( v_k \) is to be relaxed against \( v_j \), some domain restriction would be expected on average when \( |d_j| \times p \) is less than 1, because in this case the expected size of the constraint set is smaller than the domain size. Hence, ordering by \( |d_j| \times p \), or equivalently, by \( |C_{ij}| / |d_i| \), may be an effective heuristic.

Using the partitioning of satisficiencies, an argument can also be made for the potential usefulness of heuristics based on domain size. Suppose that domains of size \( k \) are to be relaxed against domains of size 1, 2, and 4. For these cases, the relative number of possible values for the satisfiability that are in the “inevitable” range is \((k - 1)/k, (k - 1)/2k \) or \((k - 1)/4k\), respectively; in general, this number is approximately \( 1/|d_j| \). Similarly, the relative numbers of possible satisfiability values within the possible and impossible ranges are approximately \( 1 - 1/|d_j| \), \( 1/|d_j| \) and \( 1/|d_j| \), respectively. Thus, with larger \( |d_j| \), the relative number of values in the impossible range is smaller, and, by the symmetry of the binomial expansion, so is the relative number of possible constraints that fall within this range. This suggests that ordering by increasing \( |d_j| \) can be an effective heuristic. Another possibility is to order the pairs by decreasing difference in size between the domain of the variable relaxed and the domain of the variable relaxed against, with signs of the differences considered.

With other factors equal, if a variable is associated with more constraints, it is more likely that a given value in its domain will not be supported in one of these constraints. This suggests that variables with more constraints should be tested first, i.e. that ordering by decreasing degree of the node in the constraint graph of the problem would be an effective heuristic.

5 An Example

The effect of some of these heuristics is illustrated with a variation of the four queens problem. In the basic problem, four queens must be placed on a 4 X 4 board so that no two can attack each other. One way to represent this as a CSP is to define the rows, from top to bottom, as variables and the columns, from left to right, as domain values. In the present variant, the first domain has only one value, associated with a queen in column 2. Relaxation reduces this problem to the solution, \((12)(4)(1)(3)\), i.e., column 2 for variable 1, column 4 for variable 2, etc.

Table 1 shows the number of additions to the list and the total number of value pairs checked, for some of the heuristics introduced in the last section. (Variable pairs that are equivalent under a heuristic are placed on the list in lexicographic order, e.g., \((12),(24),(31)\). Results for two reference orderings are also included: (i) the lexicographic ordering maintained as a queue was used explicitly by [Nadel, 1989], while queue ordering appears to be the method generally used, as indicated above, (ii) an alternative baseline is provided by the mean of ten random orderings (whose derivation is described below). Results for reversed orderings are also introduced here; this is a more sensitive method for detecting the influence of a heuristic, even when it does not yield results that are appreciably different from the reference orders.

For this problem, the effect of ordering is apparent and sometimes dramatic. Orderings that might be expected to yield good performance, viz., increasing satisfiability and increasing size of the domain relaxed against, yield lower values for the measures of work in comparison with the standards. By the same token, an excessive amount of work is required for relaxation when the ordering reverses one of the “good” orders. It should also be noted that treating the list as a stack rather than a queue does not improve efficiency. This was the typical finding for all problems, when these or other, more elaborate, placement strategies were tested.

<table>
<thead>
<tr>
<th>Ordering</th>
<th>Additions</th>
<th>Pair Checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Order (Mn.)</td>
<td>7</td>
<td>60</td>
</tr>
<tr>
<td>Lexicogr./Queue</td>
<td>5</td>
<td>35</td>
</tr>
<tr>
<td>Lexicogr./Stack</td>
<td>8</td>
<td>40</td>
</tr>
<tr>
<td>Increas. Satisfiab</td>
<td>2</td>
<td>29</td>
</tr>
<tr>
<td>Decreas. Satisfiab</td>
<td>13</td>
<td>110</td>
</tr>
<tr>
<td>Incre. Rel. Satisfiab</td>
<td>5</td>
<td>52</td>
</tr>
<tr>
<td>Decr. Rel. Satisfiab</td>
<td>9</td>
<td>65</td>
</tr>
<tr>
<td>Increas. Domain J</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>Decreas. Domain J</td>
<td>12</td>
<td>87</td>
</tr>
</tbody>
</table>

6 Experiments with Random Problems

6.1 Methods

Problem Generation
Evidence concerning the efficiency of ordering heuristics was obtained experimentally with random CSPs. The main problem sets had six or 12 variables, with a maximum domain size of ten or 12, respectively. The number of binary constraints, the specific variable pairs subject to constraint, the size of each domain, the number of acceptable pairs in each constraint, and the specific value pairs in that constraint were chosen in this order using random methods. This procedure samples from the entire set of possible CSPs within the specified limits, producing problems that are heterogeneous in the features used to order the AC-3 list (satisfiability, domain
size, and degree of constraint graph node. To examine the behavior of heuristics for CSPs with graphs of different density, the factor of constraint number was “blocked” by dividing the range of possible values for each problem size into quarter-ranges. Samples of ten problems with at least one solution and five without solutions were generated for each quarter-range for six- and twelve-variable problems, and for the lowest two quarter-ranges for twelve-variable problems. (This method rarely generates twelve-variable problems with solutions from the higher subranges.) CSPs with and without solutions were treated separately because relaxation stops immediately after all values of a domain or constraint have been eliminated. Limits for the number of solutions were set at 200 and 1000 for six- and twelve-variable problems, respectively, to balance representativeness and tractability. (Ten twelve-variable problems from the first quarter-range with more than 1000 solutions gave qualitatively similar results, with somewhat smaller differences.)

Evaluation of Performance
Relaxation performance was evaluated using the basic measure of number of value pairs checked for inclusion in a constraint. Other measures were used to identify factors underlying improvement in performance that could be related to the principles described in Section 3: (i) number of list additions, (ii) mean size of domain restriction, (iii) mean location of value deletions in the sequence of constraint checks (the sum of the products of location and number of deletions divided by the total number of deletions). It was also necessary to assess the work required to sort the list initially and to keep it in order after successive relaxations. This is discussed below (Section 7) in connection with the issue of efficient sorting methods for specific heuristics.

For each problem, a baseline for performance was obtained by running AC-3 with randomly ordered lists. In these tests, the original list was produced by placing each variable pair in one of the initial set of positions (equal to twice the number of constraints), specified by a pseudorandom number. During relaxation, execution orders were produced by placing each added pair at random in one of \( q + 1 \) possible positions in a \( q \)-element list.

An ordering heuristic partitions the set of variable pairs according to some feature of the problem. Therefore, unless each equivalence class is a singleton, there are many orders consistent with the heuristic at each major step of the procedure, i.e. with the choice of the next variable pair to check. A complete assessment of a heuristic would consider the range of efficiency possible for all such orderings. Since this is not feasible in most cases, it is necessary to rely on random samples, i.e. random orderings of equivalent pairs in the initial list and random insertion into a set of equivalent pairs during execution. For both random lists and lists ordered by heuristics, samples of size ten were used, since preliminary tests showed that these gave sufficiently stable means, compared with samples of 15 or 30.

Differences among orderings were evaluated statistically, using repeated measures analysis of variance (ANOVA) followed by nonorthogonal contrasts for comparisons of individual means. (Here the “mean” refers to a specific problem set, i.e. six- or twelve-variable problems; in addition, the number of pair checks for each problem was a mean of ten runs, as just described.) Separate analyses were carried out for six- and twelve-variable problems with range of constraints and heuristics as factors. Paired comparison \( t \) tests were used to evaluate differences in mean performance (across all constraint ranges) between orderings based on a heuristic and those based on the reversed ordering.

6.2 Results
For all sets of random CSPs, there were marked differences due to ordering the list of variable pairs. For all problem sets, the best orderings reduced the number of pair checks by about 50% in comparison with either reference ordering. (For illustration, Figure 4 gives results for problem sets in the second subrange of constraints.) The main ANOVAs included the two reference orderings: random (RAND) and a lexicographic ordering maintained as a queue (LEX/Q). Three heuristic orderings were also included, based on increasing satisfiability (SAT UP), increasing size of the domain relaxed against (DOM J UP), and decreasing degree of node (DEG DOWN). For both six- and twelve-variable problems, the effects of range and heuristic were statistically significant \( (p < .001) \). Contrasts between the two reference orderings and each heuristic were statistically significant for SAT UP and DOM J UP but not for DEG DOWN. A second ANOVA based on the three satisfiability heuristics, increasing satisfiability, relative satisfiability and increasing value of \( C_{ij} / d_i \), yielded significant effects for range \( (p < .05) \) and heuristic \( (p < .001) \). Contrasts between relatively satisfiability and the other two heuristics were statistically significant \( (p < .001) \); the contrast between the latter was not. The effectiveness of simple satisfiability therefore depends on more than satisfiability per se; another factor may be the size of the domains, which tend to be small for low satisfabilities.

Figure 4: Relaxation effort with different heuristics.
For each heuristic shown in Figure 4, comparisons with the reversed order for six-variable problems yielded statistically significant differences in favor of the original heuristic ($p < .001$). This shows that each heuristic had some effect on efficiency, even when it cannot be distinguished statistically from the reference orderings.

That good heuristics are consistent with the principles of efficient relaxation described in Section 3 is indicated by the analyses of list additions and domain restrictions during relaxation. An ANOVA for list additions based on the same five orderings as the one for pair checks yielded statistically significant effects for both range and heuristic ($p < .001$), and the nonorthogonal contrasts showed the same pattern of statistically significant results. In fact, the best orderings required only 1/4 and 1/5 of the additions required by the reference orderings for six- and 12-variable problems, respectively. The major part of the improvement in performance was, therefore, due to reduction in list additions. However, small but highly consistent differences ($p < .001$) were found for average size of domain restriction, with largest values associated with the best orderings and smallest values with the reverse orderings. This, together with the finding of more rapid relaxation, reflected in lower mean values for location of deletions ($p < .001$; with a ratio of 2:1 for mean values of the reference orderings versus the best heuristics), suggests that good heuristics also minimized the number of value comparisons. The finding of more rapid relaxation is, of course, a direct demonstration of the ASAP principle.

The above results pertain to problems with solutions. For problems without solutions, the number of pair checks required to remove all elements from a domain or constraint showed the same differences between orderings as for problems with solutions. In the present case, the size of these differences was actually greater: mean performance ratios were 3-4:1 for six-variable problems and 5-10:1 for 12-variable problems. Statistical analysis yielded the same pattern of statistically significant differences as for problems with solutions.

Since Phase 1 of AC-4 uses a list of variable pairs, this part of the algorithm was tested for amenability to ordering heuristics, using the 12-variable problems. The list was ordered lexically or with the DOM J UP heuristic. With the ordering heuristic, fewer pair checks were performed during Phase 1 for both samples of ten problems and more values were deleted ($p < .001$ for both effects). Mean pair checks for problems in the second constraint subrange were 1454 for lexical ordering and 1017 for DOM J UP. This far exceeds the numbers required for the entire AC-3 algorithm with these orderings (cf. Figure 4; similar differences between AC-3 and -4 were found for the first subrange) and is due to the need to check each value against all values in adjacent domains, to find the total support for that value. For this reason, no further analysis of AC-4 is presented here, although more extensive examination with respect to average time complexity and effects of ordering would be useful.

7 Effort Required to Keep List Ordered

Although ordering heuristics can have a significant effect on the efficiency of relaxation, costs are incurred in sorting the original set of pairs (initial ordering) and in maintaining the proper order as domains are restricted and pairs added during relaxation (execution ordering). The latter can affect the overall time complexity of relaxation, because adjustments may be necessary at each step of the procedure. It is therefore worthwhile considering heuristics that order the list initially and then use a placement strategy, such as stacking or queuing, during relaxation, as well as efficient methods for maintaining list order in each phase of the process.

Evidence bearing on the efficiency of initial ordering followed by stacking or queuing was collected for the six- and 12-variable problems of the main experiments. Discussion of these results is limited to SAT UP and DOM J UP, since these were the most promising heuristics overall. For six-variable problems the mean increase in value pair checks for initial ordering alone, compared to initial and execution ordering, was 15% for SAT UP and for DOM J UP. For 12-variable problems the corresponding increases were 50% and 45%. This indicates that execution ordering can have a significant effect on performance, and should be carried out if it can be made efficient.

For initial ordering, $O(e)$ performance can be achieved with a pigeonhole sort. For DOM J UP, this is likely to be sufficient, since the number of pigeonholes equals the maximum domain size, which should not be large. For SAT UP, the number of holes must equal the largest product of two domain sizes, so longer times may be required to scan the list, with useless steps due to empty holes. In the present tests, therefore, the number of pigeonholes was set at 1/5 the maximum. The list was partially sorted in this way, and then each hole was sorted with an insertion sort, which is very fast when no pairs are very out-of-order. For both heuristics $O(e)$ performance was approximated for the two sets of experimental problems.

For DOM J UP, execution ordering is also straightforward and efficient. After each domain restriction, the affected pairs are collected from one pigeonhole in at most $O(e)$ steps and transferred in one further step. Since this is the complexity involved in collecting adjacent pairs and checking that they are not already on the list, the overall complexity of the algorithm can be maintained while keeping the list in order. For SAT UP, the process is complicated by the need to update the representation of constraints, which can take up to $O(ed^2)$ steps. Affected pairs must be rearranged in the list, in addition to adding adjacent pairs, which may require $O(e)O(sort)$ steps. Since relaxation based on DOM J UP appears to be only slightly less efficient than SAT UP in terms of the number of pair checks, the overall efficiency of the former is therefore greater.

That DOM J UP is appreciably faster than LEX/Q overall was confirmed by timing tests on the 12-variable problems. Average CPU time per problem was 3.1 and 4.0 seconds, respectively, and for every problem the difference was in favor of the ordering heuristic.
8 Conclusions

Ordering heuristics can increase the efficiency of relaxation, reducing the number of value pair checks by a factor of two, on average, in comparison with reference orderings. These heuristics order the list of variable pairs so it conforms to the principles of efficient relaxation discussed in Section 3; the major effect is to minimize list additions. For some heuristics, it is also possible to limit the work involved in establishing and maintaining an effective ordering, so this factor does not outweigh the benefit gained during relaxation. We conclude that ordering by increasing size of the domain relaxed against is an effective general strategy for enhancing performance of algorithms that establish full arc consistency.

9 Acknowledgements

Richard Lyczak suggested some of the heuristics. Karl Gevecker wrote the program to generate random CSPs.

References


