Reversible DAC and Other Improvements for Solving Max-CSP

Javier Larrosa[*]
Univ. Pol. de Catalunya
Pau Gargallo 5,
08028 Barcelona
Spain
larrosa@lsi.upc.es

Pedro Meseguer[*]
IIIA-CSIC
Campus UAB
08193 Bellaterra
Spain
pedro@iiia.csic.es

Thomas Schiex
INRA
Chemin de Borde Rouge, BP 27
31326 Castanet-Tolosan Cedex
France
tschiex@toulouse.inra.fr

Gérard Verfaillie
ONERA-CERT
2 Av. E. Belin, BP 4025
31055 Toulouse Cedex 4
France
verfaillie@cert.fr

Abstract

Following the work of R. Wallace on Max-CSP, later improved by J. Larrosa and P. Meseguer, we tested a number of possible improvements of the usage of directed arc consistency for the *partial forward checking* algorithm (PFC). The main improvement consists in exploiting a non standard form of DAC, called *reversible DAC* where each constraint is exploited in a direction which is not necessarily determined by the variable ordering and can change dynamically during the search. Other improvements include: (i) avoiding some constraint checks when forward-checking by exploiting the constraint checks performed during DAC preprocessing (ii) using a dynamic variable ordering during the search, (iii) maintaining the directed arc-consistency counts during the search as values get deleted. These improvements have been assessed empirically on random CSP instances. Some of them lead to very large performance gains with respect to the initial algorithm.

Constraint Satisfaction Problems (CSP) consist in assigning values to variables under a given set of constraints. A solution is a total assignment that satisfies every constraint. In practice, such an assignment may not exist and it may be of interest to find a total assignment that best respects, in some sense, all the constraints. This type of problem is of interest in many applications and has been captured by general frameworks such as the Semiring or Valued CSP frameworks defined in (Bistarelli, Montanari, & Rossi 1995; Schiex, Fargier, & Verfaillie 1995). In this paper, we focus on the so-called Max-CSP problem where a solution is a total assignment that minimizes the number of violated constraints, but the reader should be aware that the algorithms presented here can easily be extended to the Valued CSP framework if needed.

The P-EFC3-DAC2 algorithm (Larrosa & Meseguer 1996), or PFC-DAC for short, is an improvement of the DAC based algorithm introduced in (Wallace 1995). It is among the best complete algorithms for Max-CSP. It is a branch and bound algorithm using forward-checking and directional arc consistency as a preprocessing step. To prune the search tree, a lower bound on the number of unsatisfied constraints is computed that takes into account (i) constraints between assigned variables using backward-checking (ii) constraints between assigned and unassigned variables using forward-checking and (iii) constraints between unassigned variables, using *directed arc consistency counts* (DAC). It needs a static variable ordering.

In this paper we present further improvements to this algorithm. In the main improvement, we relax the condition that DAC should be computed in a direction defined by the variable ordering. Moreover, (i) we save constraint checks during forward-checking by exploiting the constraint checks performed during DAC preprocessing, (ii) we make it possible to use a dynamic variable ordering during search (iii) we maintain DAC during search when values get deleted rather than simply precomputing them.

The paper is organized as follows. In the next section, we present related algorithms for solving Max-CSP. We then provide preliminaries and definitions required in the sequel of the paper. Then, we consider the original PFC-DAC algorithm and present our improvements. The result of these improvements is finally assessed empirically on random Max-CSP instances, before the conclusion.

Related Work

The general scheme of most algorithms for solving Max-CSP is a *branch and bound* scheme. The algorithm performs a systematic traversal on the search tree where a node corresponds to a set of assigned (or *past*) variables and a set of unassigned (or *future*) variables. At each node, one future variable is selected (*current* variable) and all its feasible values are considered for instantiation. As search proceeds, branch and bound keeps track of the best solution obtained so far which is the total assignment which violates a minimum number of constraints in the explored part of the search tree. In order to avoid the exploration of the complete search tree, the algorithm computes at each node a *lower bound* on the cost of the best solution that could possibly be found under the current node. If this lower bound is larger than or equal to the cost of the best solution found so far (called the *upper bound*), the current line of search is abandoned because it cannot lead to a better solution that the current one. In practice, the efficiency of

[*]The research of Javier Larrosa and Pedro Meseguer is supported by the Spanish CICYT project TIC96-0721-C02-02

branch and bound algorithms deeply depends on the quality of the lower bound which should be both as large as possible and as cheap to compute as possible.

At a given node, the simplest lower bound one can imagine is defined by the number of constraints violated by the partial assignment associated with the node, also called the *distance* of the node. The PFC algorithm uses *forward checking* to improve this lower bound (see (Freuder & Wallace 1992) for a more detailed description). Further improvements have been introduced by the *Russian Doll Search* algorithm (Verfaillie, Lemaître, & Schiex 1996) or by algorithms using directed arc consistency.

The notion of *directed arc consistency* was first introduced by (Dechter & Pearl 1988), in the context of classical CSP. The use of DAC in Max-CSP as a method to improve the lower bound quality of the PFC algorithm has been first proposed in (Wallace 1995) and improved by (Larrosa & Meseguer 1996; Wallace 1996). In a preprocessing step, so-called directed arc consistency counts are computed for each value following a given variable ordering. This order must be used as a static variable ordering for variable instantiation in the DAC-based branch and bound algorithm.

Preliminaries

A discrete binary CSP is defined by a finite set of variables $X = \{1, \ldots, n\}$. Each variable $i \in X$ takes its values in a finite domain D_i and is subject to constraints R_{ij}. A constraint R_{ij} is a subset of $D_i \times D_j$ which defines the allowed pairs of values for variables i, j. We note n and e the number of variables and constraints and d the maximum cardinality of the domains D_i. In the sequel, $i, j, k \ldots$ denote variables, $a, b, c \ldots$ denote values, and a pair such as (i, a) denotes the value a of variable i.

It is usual, for arc-consistency algorithms, to associate with a given CSP a symmetric directed graph. For directed arc-consistency, we define a directed graph of the CSP to be a graph with one vertex i for each variable i and *only one* edge, either (i, j) or (j, i), for each constraint R_{ij}. Such a graph is not uniquely defined for a given CSP. Its edges will be denoted by EDGES(G). We will note PRED(i, G) (resp.SUCC(i, G)) the set of variables j such that (j, i) (resp. (i, j)) is an edge of G. We define DIRECTION(i, j, G) to be 1 is $(i, j) \in G$, -1 if $(j, i) \in G$ and 0 if there is no constraint between i and j.

Directed arc-consistency being defined up to a variable ordering, we define $G^{\succ}$, the directed graph induced by a variable ordering $\succ$, as the directed graph of the CSP such that $(i, j) \in$ EDGES$(G) \Rightarrow (i \succ j)$. $G^{\succ}$ is therefore the directed graph where all edges are oriented in a direction opposed to the variable ordering. The directed arc consistency count associated with value a of variable i, noted dac_{ia}, is defined as the number of variables in PRED$(i, G^{\succ})$ which are *arc-inconsistent* with value (i, a) *i.e.*, which have no value in their domain which is compatible with (i, a). All DAC can be simply precomputed in $O(ed^2)$ time and $O(nd)$ space. $\min_a(dac_{ia})$ is a lower bound on the number of inconsistencies that i will have with variables after i in the ordering in any total assignment.

The PFC-DAC algorithm (Larrosa & Meseguer 1996) is a branch and bound algorithm. It is described by functions PFC-DAC and LookAhead below. For future variables, the lower bound exploits the directed arc consistency counts defined above combined with so-called *inconsistency counts* introduced in the PFC algorithm (Freuder & Wallace 1992). Given a partial assignment, the inconsistency count associated with value a of a future variable i, noted ic_{ia}, is the number of constraints between any assigned variable and i that would be violated if value a is assigned to variable i (Freuder & Wallace 1992). If FV is the set of future variables, then the sum $\sum_{i \in FV} \min_a(ic_{ia} + dac_{ia})$ is a lower bound on the number of constraints that have to be violated between currently unassigned variables and all CSP variables in any total assignment extending the current partial assignment. This lower bound is used to prune branches (line 1.1) and to prune values (lines 2.1 and 2.2).

Function 1: PFC-DAC main function, S is the current partial assignment, d its distance, *FV* and *FD* are the future variables and their domains

PFC-DAC(S,d,FV,FD);
if $FV = \varnothing$ **then**
  if $(d < bestd)$ **then**
    $bestd := d$;
    $BestS := S$;
else
  i := SelectVariable(FV);
  $FV := FV - \{i\}$;
  $values$:= SortValues(FD_i);
  while $(values \neq \varnothing)$ **do**
    a := First($values$);
    $newd := d + ic_{ia}$;
1.1   **if** $(newd + dac_{ia} + \sum_{j \in FV} \min_b(ic_{jb} + dac_{jb}) < bestd)$ **then**
      $newFD$:= LookAhead(i,a,FV,FD);
      if *not WipeOut(newFD)* **then**
        PFC-DAC($S \cup \{(i, a)\}$, $newd$, FV, $newFD$);
    $values := values - \{a\}$;

Function 2: PFC-DAC propagation function, (i, a) is the assignment to propagate, *FV* and *FD* are the future variables and their domains

LookAhead(i,a,FV,FD);
foreach $j \in FV$ **do**
  foreach $b \in FD_j$ **do**
2.1   **if** $(newd + \sum_{k \in FV - \{j\}} \min_c(ic_{kc} + dac_{kc}) + ic_{jb} + dac_{jb} \geq bestd)$ **then** Prune(j, b);
    else if *Inconsistent(i,a,j,b)* **then**
      Increment(ic_{jb});
2.2   **if** $(newd + \sum_{k \in FV - \{j\}} \min_c(ic_{kc} + dac_{kc}) + ic_{jb} + dac_{jb} \geq bestd)$ **then** Prune(j, b);
return *Updated domains*;

DAC Improvements

The basic procedure PFC-DAC for MAX-CSP can be improved in different ways, provided that the DAC preprocessing produces a data structure *GivesDac*(i, a, j) that records the contribution of any variable j connected to i to the count dac_{ia}. The information contained in this data structure is used during search to perform the improvements detailed below. The size of this data structure is in $O(ed)$. In the following, we present four different and independent improvements on DAC usage:

1. Saving checks associated with DAC.
2. Dynamic variable ordering.
3. Reversible DAC.
4. Maintaining DAC.

Saving checks associated with DAC: If *Gives-Dac*(i, a, j) is true ($i \prec j$), it means that no value of j is compatible with (i, a). When variable i becomes current and is assigned value a, we already know that the IC of every feasible value of variable j must be incremented by one (because (i, a) is arc-inconsistent with D_j), so IC updating of j values can be done without any constraint check. In practice, instead of incrementing all ic_{jb}, we increase the current distance by 1 and prevent the updating of all ic_{jb} when *GivesDac*(i, a, j) is true. This simple idea allows lookahead to reuse results obtained in DAC preprocessing, and saves the repetition of all the constraint checks associated with detected arc-inconsistencies.

Dynamic variable orderings: Original PFC-DAC requires to follow the same static order for variable selection that was used for DAC computation. However, this is no longer required, provided that individual contributions to DAC are available. Let i and j be variables such that $\text{GIVESDAC}(i, a, j)$ is true. DAC propagate the effect of R_{ij} from j to i, so the violation of this constraint is recorded in DAC of variable i. If variable j is instantiated before i, lookahead will propagate the possible violation of R_{ij} from j to i, incrementing some IC of i (and in particular, ic_{ia}). But this inconsistency has already been recorded in dac_{ia} using the *forward* edge (j, i), so adding $ic_{ia} + dac_{ia}$ would count twice the same inconsistency.

This problem can be overcome as far as the lookahead detects the situation and avoids the redundant contribution to the lower bound. If we know that R_{ij} has been used from j to i, j can be safely instantiated before i as far as we prevent the updating of all ic_{ia} such that *GivesDac*(i, a, j) is true. In this way, it is guaranteed that inconsistencies are counted only once, and IC and DAC of future values can be added to form a lower bound of inconsistencies for that value.

Reversible DAC: Originally (Wallace 1995) discarded the use of full arc-inconsistency counts because they could record the same inconsistency twice, so addition could not be safely used to compute lower bounds. Instead, he proposed directed arc-inconsistency counts, which do not suffer from this drawback. Following the work of (Dechter & Pearl 1988) on directed arc consistency, Wallace required a static variable ordering $\succ$, where DAC are computed in the direction of the corresponding edge in $G^{\succ}$. However, for lower bound computation, this restriction is arbitrary and can be removed. Each constraint can contribute to DAC in one of the two possible directions, to avoid inconsistency repetition, but the selected direction has not to be induced by a variable ordering.

With this idea in mind, given any directed graph G of a CSP, one can define directed arc inconsistency counts dac_{ia} based on this graph as the number of variables in $\text{PRED}(i, G)$ which are arc-inconsistent with (i, a). The lower bound on the number of violated constraints that may exist in a complete assignment is defined as before as:

$$\sum_{i \in X} \min_a dac_{ia}$$

During search, the following lower bound can be used, as far as DAC have their contributions from past variables removed,

$$distance + \sum_{i \in FV} \min_a (ic_{ia} + dac_{ia})$$

At each node, one could try to find an optimal directed graph G, i.e., to determine an order for each future constraint that maximizes the lower bound. We have taken a simpler approach: to *locally* optimize G at each node during tree search using simple greedy heuristics. In our approach, we start from the graph inherited from the previous search state, and reverse constraints between future variables if it improves the lower bound. The process iterates until no such constraints can be found. When a constraint is reversed, DAC need to be appropriately modified. The *GivesDac* data-structure is suitable to do it efficiently.

As in the dynamic variable ordering case, forward edges may be met during assignment and again, if we know that R_{ij} is used from j to i, j can be safely instantiated before i as far as we prevent the updating of all ic_{ia} such that *Gives-Dac*(i, a, j) is true.

Maintaining DAC: PFC-DAC requires a strong condition for variables to contribute to DAC counts: arc-inconsistencies need to hold before search starts, when no value is pruned. The PFC-DAC algorithm prunes future values during its execution. This implies that DAC counters, precomputed initially before search, are not updated during search. The maintaining DAC approach consists on keeping updated those DAC during search, taking into account current domains. This causes higher DAC to be computed, which leads to a higher lower bound: more branches may be pruned, more value deletions can occur, which are again propagated, etc. To perform maintaining DAC, any arc consistency algorithm (along with adequate data-structures) can be adapted to propagate the effect of value removal, either in one static direction of constraints (as in the basic PFC-DAC procedure), or in both directions, if the reversible DAC approach is taken.

Implementation

Each of the improvements introduced in this paper can be used alone. For the sake of simplicity and because of the limited space, we will define a version of the algorithm that includes all the refinements.

The data-structure *GivesDac*(i,a,j) has been described in the previous section: *GivesDac*(i,a,j) is true iff variable j contributes to dac_{ia}. It is needed for our three first improvements. In order to implement the fourth improvement (DAC maintainance), we have used AC4 like data-structures: *NSupport*(i,a,j) contains the number of supports of value (i,a) on variable j and *LSupport*(i,a,j) contains the list of all values (j,b) that are supported by (i,a). These data-structures have space complexity $O(ed)$ and $O(ed^2)$ respectively and are useless when DAC are not maintained. One can observe that *NSupport*$(i,a,j) = 0 \Leftrightarrow$ *GivesDac*$(i,a,j) =$ true and therefore the *GivesDac* data-structure is redundant with *NSupport* and can be removed in this case.

Extra data structures have been introduced to easily maintain the quantities $\mathrm{argmin}_a(ic_{ia}+dac_{ia})$, $\min_a(ic_{ia}+dac_{ia})$ and the sum of these minima on future variables. All these quantities are respectively maintained in the arrays *MinICDACValue*(i), *MinICDAC*(i) and in the variable *SumMinICDAC*. All these data structures are updated by a simple function noted UpdateMin in the sequel (not described here because of its simplicity).

Function 3: PFC-MRDAC main function, S is the current assignment, d its distance, *FV* and *FD* are the future variables and their domains and G is the current directed graph

```
    PFC-MRDAC-DVO(S,d,FV,FD,G);
    if FV = ∅ then
      if (d < bestd) then
        bestd := d;
        BestS := S;
    else
      i := SelectVariable(FV);
      FV := FV − {i};
      SumMinICDAC := SumMinICDAC-MinICDAC(i);
      values := SortValues(FD_i);
      while (values ≠ ∅) do
        a := First(values);
3.1     newd := d + ic_ia + dac_ia;
        if (newd + SumMinICDAC < bestd) then
          newFD := LookAhead(i,a,FV,FD);
          if not WipeOut(newFD) then
3.2         newG := GreedyOpt(G);
            if (newd + SumMinICDAC) < bestd) then
3.3           PruneValues(newd,i);
              PFC-MRDAC-DVO(S ∪ {(i,a)}, newd, FV, newFD,
              newG);
        values := values − {a};
```

The algorithm is embodied in the main function PFC-MRDAC-DVO. The reader should be aware that all context restoration mechanisms are not explicited in the code, for the sake of simplicity[1]. Before main function PFC-MRDAC-DVO is called, one should initialize data-structures dac and *GivesDac* or (*NSupport* and *LSupport*) if the fourth improvement is used. The initialization is straightforward and not described here. It can be performed in time $O(ed^2)$.

Compared to the initial PFC-DAC algorithm, line 3.1 has been modified in order to implement the first improvement. Lines 3.2 and 3.3 have been inserted to implement the reversible DAC improvement. The function GreedyOpt, used on line 3.2, implements the greedy optimization of the current directed graph. After this call, a new lower bound may be available and new values may be pruned. This is done by the PruneValues function on line 3.3.

The LookAhead function is in charge of propagating the assignment (i,a). Lines 4.4, 4.5 and 4.7 have been inserted to update the data-structure *MinICDAC* and *SumMinICDAC* when needed. Our three first improvements need to prevent the updating of ic_{jb} in some cases, because the corresponding costs have already been taken into account in PFC-MRDAC-DVO, on line 3.1. This is done by the test on lines 4.2 and 4.3. The last change in this function consists in calls to the new function PropagateDel on lines 4.1 and 4.6, which implements our fourth improvement: this function is in charge of propagating the deletion of value (j,b) on the previous line and updating the directed arc consistency counts.

Function 4: PFC-MRDAC propagation function, (i,a) is the assignment to propagate, *FV* and *FD* are the future variables and their domains

```
    LookAhead(i,a,FV,FD);
    foreach j ∈ FV do
      NewMin := false;
      foreach b ∈ FD_j do
        if (newd + SumMinICDAC − MinICDAC(j) + ic_jb + dac_jb ≥
        bestd) then Prune(j,b);
4.1     PropagateDel(j,b,FV);
4.2     else if (DIRECTION(i,j,G) = −1 & (not GivesDac(i,a,j)))
        or
4.3     (DIRECTION(i,j,G) = 1 & (not GivesDac(j,b,i))) then
          if Inconsistent(i,a,j,b) then
            Increment(ic_jb);
4.4         if MinICDAC(j) = ic_jb + dac_jb then
4.5           NewMin := true;
            if (newd + SumMinICDAC − MinICDAC(j) + ic_jb +
            dac_jb ≥ bestd) then
              Prune(j,b);
4.6           PropagateDel(j,b,FV);
4.7   if NewMin then UpdateMin(j);
    return Updated domains;
```

This propagation is done simply using the AC4-like data-structures: the number of supports of all the feasible values which are supported by the value being deleted are decremented. If a value loses all its supports on one constraint

[1] All context restorations are done using lists to memorize changes and later undo these changes upon backtrack.

and if the correct direction of the constraint is used in the directed graph, the corresponding dac is incremented and the data-structures *MinICDAC* and *SumMinICDAC* updated by function UpdateMin if needed. One should note that function LookAhead and PropagateDel together do not completely "maintain DAC". Indeed, if a deletion in LookAhead is propagated by PropagateDel, a deletion in PropagateDel is not propagated again. This appeared to be useless in our experiments.

Function 5: Updating dac after deletions, (i, a) is the deleted value, *FV* is the set of future variables

```
PropagateDel(i,a, FV);
NewMin := false;
foreach j ∈ FV do
    foreach b ∈ LSupport(i, a, j) do
        if b is feasible then
            Decrement(NSupport[j][b][i]);
            if NSupport[j][b][i] = 0 & DIRECTION(i, j) = 1 then
                Increment(dac_jb);
                if MinICDAC(j) = ic_jb + dac_jb then
                    UpdateMin (j);
                    NewMin := true;
return NewMin;
```

The function GreedyOpt is in charge of optimizing the current directed graph G in order to improve the current lower bound. It is a simple greedy function: each edge is reversed (and the directed arc consistency counts modified accordingly). The lower bound is then recomputed and the change is kept only if it leads to an improvement. The function stops when no improvement could be obtained. Note that before edge (i, j) is reversed, a simple test is performed on line 6.1: this test is an obvious necessary condition for an improvement to be possible.

Function 6: Finding a "good" directed graph, G is the current directed graph

```
    GreedyOpt(G);
    Stop := false;
    while not Stop do
        SaveMin := SumMinICDAC;
        foreach i, j ∈ FV s.t. (i, j) ∈ EDGES(G) do
            MinF := MinICDAC(i);
            ValMinF := MinICDACValue(i);
            MinT := MinICDAC(j);
            ValMinT := MinICDACValue(j);
6.1         if (not GivesDac(j, ValMinT, i)) & GivesDac(i, ValMinF, j)
            then
                Reverse(i, j, G);
                UpdateMin(i);
                UpdateMin(j);
                if MinICDAC(i)+MinICDAC(j) < MinF + MinT then
                    Reverse(i, j, G);
                    UpdateMin(i);
                    UpdateMin(j);
        if SaveMin = SumMinICDAC then
            Stop := true;
    return Updated graph
```

Finally, the function PruneValues is in charge of value deletions once the lower bound has been improved by the GreedyOpt function. It can be considered as a simplified LookAhead function. Each value deletion is again propagated using function PropagateDel. Contrarily to function LookAhead however, a fix point is reached: value deletions are propagated until no new deletion occurs. This has been found to be useful in practice although gains are minor.

Function 7: Pruning values after a lower bound improvement, d is the current distance and *FV* is the set of future variables

```
PruneValues(d,FV);
Stop := false;
while not Stop do
    Copy := SumMinICDAC;
    foreach j ∈ FV do
        foreach b ∈ FD_j do
            if (d+ SumMinICDAC − MinICDAC(j) + ic_jb + dac_jb ≥
            bestd) then
                Prune(j, b);
                PropagateDel(j, b,FV);
        if Wipe out then Empty := Stop := true;
    if Copy = SumMinICDAC then Stop := true;
return not Empty;
```

Experimental Results

We have evaluated the performance of our algorithms on over-constrained random CSP. A random CSP is characterized by $\langle n, d, p_1, p_2\rangle$ where n is the number of variables, d the number of values per variables, p_1 the graph *connectivity* defined as the ratio of existing constraints, and p_2 the constraint *tightness* defined as the ratio of forbidden value pairs. The constrained variables and the forbidden value pairs are randomly selected (Prosser 1994). Using this model, we have experimented on the following problems classes:

1. $\langle 10, 10, \frac{45}{45}, p_2\rangle$,
2. $\langle 15, 5, \frac{105}{105}, p_2\rangle$,
3. $\langle 15, 10, \frac{50}{105}, p_2\rangle$,
4. $\langle 20, 5, \frac{100}{190}, p_2\rangle$,
5. $\langle 25, 10, \frac{37}{300}, p_2\rangle$,
6. $\langle 40, 5, \frac{55}{780}, p_2\rangle$.

Observe that (1) and (2) are highly connected problems, (3) and (4) are problems with medium connectivity, and (5) and (6) are sparse problems. For each problem class and each parameter setting, we generated samples of 50 instances.

Each problem is solved with three algorithms: PFC-DAC as described in (Larrosa & Meseguer 1996), PFC with reversible DAC and dynamic variable ordering (DVO), and PFC maintaining reversible DAC without DVO. We will refer to these algorithms as: PFC-DAC, PFC-RDAC-DVO and PFC-MRDAC, respectively. PFC-DAC and PFC-MRDAC use *forward degree*, breaking ties with *backward degree* (Larrosa & Meseguer 1996) as static variable ordering. PFC-RDAC-DVO uses *minimum domain* breaking ties with *graph degree* as dynamic variable ordering. We do not use DVO with PFC-MRDAC because we observed that it did not give any gain to the algorithm. Values are always selected by increasing IC+DAC. All three algorithms share

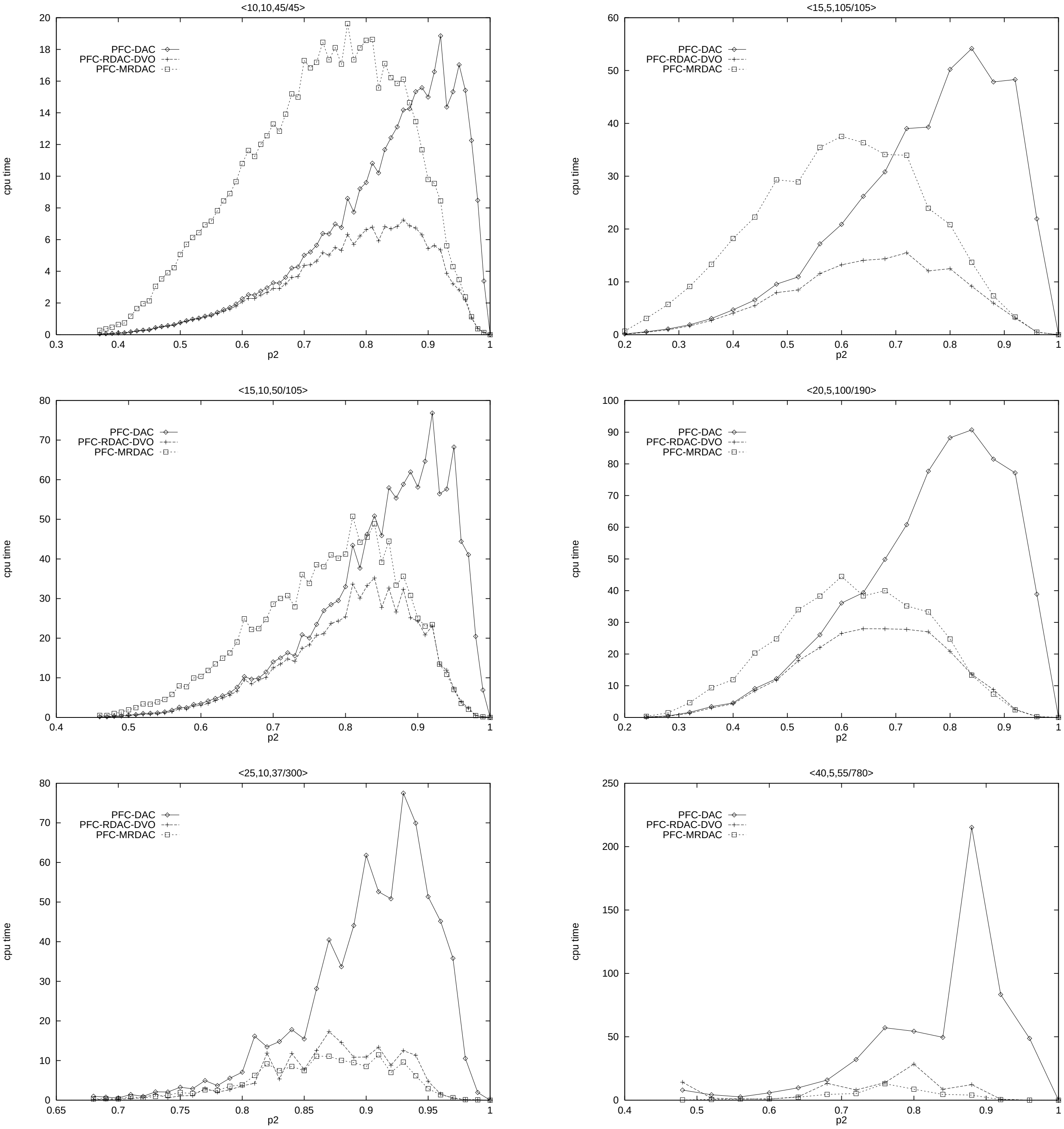

Figure 1: Measure of the cpu-time in various locations of the random CSP space

Random class	PFC DAC	PFC RDAC+DVO	PFC MRDAC
$\langle 10, 10, \frac{45}{45}, \frac{92}{100}\rangle$	191667	26706	12246
$\langle 15, 5, \frac{105}{105}, \frac{21}{25}\rangle$	565664	51241	23889
$\langle 15, 10, \frac{50}{105}, \frac{95}{100}\rangle$	442827	25943	10517
$\langle 20, 5, \frac{100}{190}, \frac{21}{25}\rangle$	748673	67555	24473
$\langle 25, 10, \frac{37}{300}, \frac{93}{100}\rangle$	412160	41672	15158
$\langle 40, 5, \frac{55}{780}, \frac{22}{25}\rangle$	1315303	44346	8287

Table 1: Visited nodes by the three algorithms on the hardest class for PFC-DAC.

code and data structures whenever it is possible. Experiments were performed using a Sun Sparc 2 workstation.

Figure 1 reports the average cost required to solve the six problem classes. Since the overhead produced by RDAC and MDAC is consistency check-free, we use CPU-time to compare search effort. As it can be observed, PFC-RDAC improves PFC-DAC in practically all problem classes. The gain grows with problem tightness. PFC-RDAC can be up to 900 times faster than PFC-DAC on the tightest sparse instances. Typical improvement ratios range from 1.5 to 20 for tightness greater than 0.6. Regarding the contribution of the individual improvements, we can state with no doubt that reversible DAC is the main responsible for the gain. Avoiding the repetition of checks associated with DAC, and using DVO have a limited effect that can cause improvement ratios from 1.2 to 2.

Regarding PFC-MRDAC, we observe that maintaining RDAC does only pay off on the tightest instances, and on the most sparse problems. On sparse problems PFC-MRDAC can be from 1.5 to 3 times faster than PFC-RDAC. This fact is understandable since we are using an AC4 based implementation. On loose constraints, lists of support are longer and supports are higher. Thus, propagating a deletion is more costly and is less likely to produce a new DAC contribution. We believe that moving to more elaborated local consistency algorithms (i.e. AC6-7) will increase the range of problems where maintaining RDAC pays off.

Regarding the number of visited nodes, Table 1 contains the average number for each algorithm on the hardest problem class for PFC-DAC. One can observe a decrease of an order of magnitude from PFC-DAC to PFC-RDAC-DVO (two orders for the $\langle 40, 5\rangle$ class). Besides, from PFC-RDAC-DVO to PFC-MRDAC, the average number of visited nodes is divided by a constant between 2 and 5. Savings in CPU time are not as large as in visited nodes because improved algorithms perform more work per node than PFC-DAC.

Conclusion

Several observations may be extracted from this work. Quite surprisingly, we have observed that the introduction of dynamic variable orderings, when it brings something, provides only very minor savings. This contradicts traditional wisdom in classical CSP and further studies are needed to check whether better DVO heuristics can be found.

Finally, the fact that reversible DAC is the improvement that brings the largest savings confirms the importance of the lower bound quality in branch and bound algorithms for Max-CSP. It is our feeling that lower bound quality remains the major issue in complete algorithms for Max-CSP. Considering the simplicity of the greedy optimization algorithm used in this paper for optimizing RDAC, there is probably an opportunity to still improve this lower bound.

References

Bistarelli, S.; Montanari, U.; and Rossi, F. 1995. Constraint solving over semirings. In *Proc. of the 14th IJCAI*.

Dechter, R., and Pearl, J. 1988. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence* 34:1–38.

Freuder, E., and Wallace, R. 1992. Partial constraint satisfaction. *Artificial Intelligence* 58:21–70.

Larrosa, J., and Meseguer, P. 1996. Exploiting the use of DAC in Max-CSP. In *Proc. of CP'96*, 308–322.

Prosser, P. 1994. Binary constraint satisfaction problems: Some are harder than others. In *Proc. of the 11st ECAI*.

Schiex, T.; Fargier, H.; and Verfaillie, G. 1995. Valued constraint satisfaction problems: hard and easy problems. In *Proc. of the 14th IJCAI*, 631–637.

Verfaillie, G.; Lemaître, M.; and Schiex, T. 1996. Russian doll search. In *Proc. of AAAI-96*, 181–187.

Wallace, R. 1995. Directed arc consistency preprocessing. In Meyer, M., ed., *Selected papers from the ECAI-94 Workshop on Constraint Processing*, number 923 in LNCS. Berlin: Springer. 121–137.

Wallace, R. 1996. Enhancements of branch and bound methods for the maximal constraint satisfaction problem. In *Proc. of AAAI-96*, volume 1, 188–195. Portland, OR: AAAI Press/MIT Press.