ELSEVIER

Artificial Intelligence 107 (1999) 149–163

Artificial Intelligence

Research Note

Maintaining reversible DAC for Max-CSP

Javier Larrosa [a,1], Pedro Meseguer [b,*], Thomas Schiex [c,2]

[a] *Departamento Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Jordi Girona Salgado, 1-3, 08034 Barcelona, Spain*

[b] *Institut d'Investigació en Intel·ligència Artificial, Consejo Superior Investigaciones Científicas, IIIA-CSIC, Campus Universitat Autònoma de Barcelona, 08193 Bellaterra, Spain*

[c] *INRA, Chemin de Borde Rouge, BP 27, 31326 Castanet-Tolosan Cedex, France*

Received 10 August 1998; received in revised form 23 November 1998

Abstract

We introduce an exact algorithm for maximizing the number of satisfied constraints in an overconstrained CSP (Max-CSP). The algorithm, which can also solve weighted CSP, probabilistic CSP and other similar problems, is based on directed arc-inconsistency counts (DAC). The usage of DAC increases the lower bound of branch and bound based algorithms for Max-CSP, improving their efficiency. Originally, DAC were defined following a static variable ordering. In this paper, we relax this condition, showing how DAC can be defined from a directed constraint graph. These new graph-based DAC can be effectively used for lower bound computation. Interestingly, any directed constraint graph of the considered problem is suitable for DAC computation, so the selected graph can change dynamically during search, aiming at optimizing the exploitation of directed arc-inconsistencies. In addition, directed arc-inconsistencies are maintained during search, propagating the effect of value pruning. With these new elements we present the PFC *maintaining reversible* DAC algorithm (PFC-MRDAC), a natural successor of PFC-DAC for Max-CSP. We provide experimental evidence for the superiority of PFC-MRDAC on random and real overconstrained CSP instances, including problems with weighted constraints.

Keywords: Maximal/partial constraint satisfaction; Branch and bound; Partial forward checking; Directed arc-consistency

* Corresponding author. Email: pedro@iiia.csic.es.

[1] Email: larrosa@lsi.upc.es.

[2] Email: tschiex@toulouse.inra.fr.

0004-3702/99/$ – see front matter
PII: S0004-3702(98)00108-8

1. Introduction

A discrete binary constraint satisfaction problem (CSP) is defined by a finite set of variables $X = \{1, \ldots, n\}$, a set of finite domains $\{D_i\}_{i=1}^{n}$ and a set of binary constraints $\{R_{ij}\}$. Each variable i takes values in its corresponding domain D_i. A constraint R_{ij} is a subset of $D_i \times D_j$ which only contains the allowed value pairs for variables i, j. An assignment of values to variables is complete if it includes every variable in X, otherwise it is incomplete. A *solution* for a CSP is a complete assignment satisfying every constraint. If the problem is overconstrained, such an assignment does not exist, and it may be of interest to find a complete assignment that best respects all constraints [2,12]. In this paper, we focus on the Max-CSP problem, for which the solution of an overconstrained CSP is a complete assignment satisfying as many constraints as possible. The number of variables is n, the maximum cardinality of domains is d and the number of constraints is e. Letters $i, j, k, \ldots$ denote variables, $a, b, c, \ldots$ denote values, and a pair (i, a) denotes the value a of variable i.

Most exact algorithms for solving Max-CSP follow a *branch and bound* schema. These algorithms perform a depth-first traversal on the search tree defined by the problem, where internal nodes represent incomplete assignments and leaf nodes stand for complete ones. Assigned variables are called *past* (P), while unassigned variables are called *future* (F). The *distance* of a node is the number of constraints violated by its assignment. At each node, branch and bound computes the *upper bound* (UB) as the distance of the best solution found so far (complete assignment with minimum distance in the explored part of the search tree), and the *lower bound* (LB) as an underestimation of the distance of any leaf node descendant from the current one. When $UB \leqslant LB$, we know that the current best solution cannot be improved below the current node. In that case, the algorithm prunes all its successors and performs backtracking.

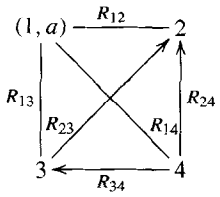
The efficiency of branch and bound based algorithms largely depends on the quality of the lower bound, which should be both as large and as cheap to compute as possible. At the current node, the simplest lower bound is $distance(P)$, the number of inconsistencies among past variables. It is improved in the *partial forward checking* algorithm (PFC) [6], which records lookahead effects on future variables in *inconsistency counts* (IC). The inconsistency count of value a of a future variable i, ic_{ia}, is the number of past variables inconsistent with (i, a). PFC lower bound is $distance(P) + \sum_{i \in F} \min_a(ic_{ia})$. PFC also computes the lower bound associated with value b of future variable j, as $distance(P) + ic_{jb} + \sum_{i \in F - \{j\}} \min_a(ic_{ia})$. Value b can be pruned when its associated lower bound reaches UB. PFC lower bound is improved including inconsistencies among future variables by the usage of *directed arc-inconsistency counts* (DAC) [14]. Given a static ordering in X, the directed arc-inconsistency count of value a of variable i, dac_{ia}, is the number of variables in X which are arc-inconsistent[3] with (i, a) and appear after i in the ordering. A new lower bound is $distance(P) + \sum_{i \in F} \min_a(ic_{ia}) + \sum_{i \in F} \min_a(dac_{ia})$ [14], providing variables are assigned following the static order. The second and third terms of this expression can be combined to form a better lower bound as $distance(P) + \sum_{i \in F} \min_a(ic_{ia} + dac_{ia})$ in the PFC-DAC algorithm [7]. Another way to

[3] A variable j is arc-inconsistent with (i, a) when no value of j is compatible with (i, a).

$X = \{1, 2, 3, 4\}$ $D_1 = D_2 = D_3 = D_4 = \{a, b, c\}$

$R_{12} = \{(a, b), (b, a), (c, b)\}$ $R_{13} = \{(a, a), (b, b), (c, c)\}$ $R_{14} = \{(a, b), (b, c), (c, a)\}$
$R_{23} = \{(a, b), (a, c), (c, b)\}$ $R_{24} = \{(b, a), (b, c)\}$ $R_{34} = \{(b, a), (b, c)\}$

$P = \{(1, a)\}$, $F = \{2, 3, 4\}$, DAC computed under lexicographical order



	ic	dac
a	0	1
b	1	0
c	1	1

	ic	dac
a	1	1
b	0	1
c	1	1

	ic	dac
a	1	0
b	0	0
c	1	0

$distance(P) = 0$

$distance(P) + \sum_{i \in F} \min_a(ic_{ia}) = 0 + 0 = 0$

$distance(P) + \sum_{i \in F} \min_a(ic_{ia}) + \sum_{i \in F} \min_a(dac_{ia}) = 0 + 0 + 1 = 1$

$distance(P) + \sum_{i \in F} \min_a(ic_{ia} + dac_{ia}) = 0 + 2 = 2$

Fig. 1. A simple problem and the computation of the four different lower bounds explained in Section 1, after assigning a to variable 1. Future variables are lexicographically ordered. Constraints among future variables are oriented, each constraint pointing to the variable which records its inconsistencies. The *ic* and *dac* counts of each future variable are shown.

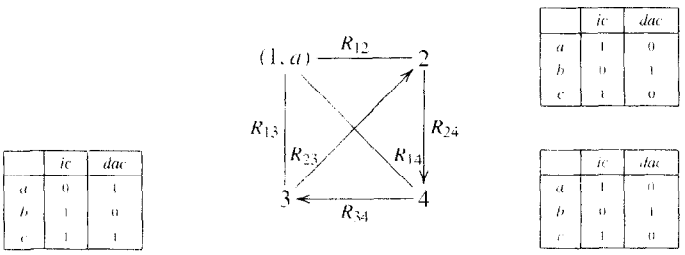
include inconsistencies among future variables is *Russian doll search* [13]. An example on the computation of the above lower bound expressions appears in Fig. 1.

Originally, DAC were defined following a static variable ordering. In this paper, we relax this condition, showing how DAC can be defined from a directed constraint graph. These new graph-based DAC can be effectively used for lower bound computation. Interestingly, any directed constraint graph of the considered problem is suitable for DAC computing, so the selected graph can change dynamically during search, in order to optimize the exploitation of directed arc-inconsistencies. In addition, directed arc-inconsistencies are maintained during search, propagating the effect of value pruning. With these new elements we present the PFC *maintaining reversible* DAC algorithm (PFC-MRDAC), a natural successor of PFC-DAC for Max-CSP. This algorithm has been extended to the weighted CSP case and could be easily adapted to deal with other frameworks such as probabilistic or lexicographic (hierarchical) CSP [12].

2. Reversible DAC

Originally, Wallace discarded the use of full *arc-inconsistency counts* (AC) (the arc-inconsistency count of value a of variable i, ac_{ia}, is the number of variables which are arc-inconsistent with (i, a)), because they could record the same inconsistency in two different counts, so they could not be safely added for lower bound contribution [14] (see [1] for

$P = \{(1, a)\}$, $F = \{2, 3, 4\}$, $\text{EDGES}(G^F) = \{(3, 2), (4, 3), (2, 4)\}$



$$LB(P, F, G^F) = distance(P) + \sum_{i \in F} \min_a (ic_{ia} + dac_{ia}(G^F)) = 0 + 3 = 3$$

Fig. 2. Lower bound computed with graph-based DAC after assigning a to variable 1. The only difference with the constraint graph of Fig. 1 is the edge (4,2), now reversed into (2,4).

a new way to overcome this fact). Instead, he proposed DAC which do not suffer from this drawback. Following the work of Dechter and Pearl on directional consistency [5], Wallace required a static variable ordering, each constraint being directed in the opposite sense of that ordering. From these directed constraints, DAC were precomputed before search. These DAC do not change during search. In addition, IC and DAC of each future value can be safely added because they always register different inconsistencies (IC register inconsistencies with past variables, while DAC register inconsistencies with subsequent variables in the ordering). These two properties are very convenient and make the algorithm conceptually simple and easy to implement.

The essential point in the above description is that binary constraints have to be *directed*. A directed constraint only contributes to the DAC of one of its two variables, so its violation cannot be recorded in two different counts. However, constraint directions do not have to be induced by a static variable ordering. For lower bound computation this restriction is arbitrary and can be removed, at the cost of making the algorithm more complex. Therefore, we can define DAC based on a directed constraint graph. These new graph-based DAC can be effectively used to compute lower bounds, because no inconsistency duplication may occur. Interestingly, any directed constraint graph of the considered problem is suitable for DAC computation. Therefore, during search we can change the graph from which DAC are computed, looking for a good directed graph causing a high lower bound.

2.1. Graph-based DAC

A directed constraint graph of a CSP can be obtained from the usual constraint graph by selecting a direction for each edge. There are two possible directions for the edge-connecting variables i and j: from i to j, noted as (i, j), or from j to i, noted as (j, i). The edge direction indicates the variable that records the possible arc-inconsistencies: (j, i) means that arc-inconsistencies of R_{ij} will be recorded in the DAC of i. Given a directed graph G, we note $\text{EDGES}(G)$ the list of all the directed edges of G and $\text{PRED}(i, G)$ the set of variables j such that $(j, i) \in \text{EDGES}(G)$. The directed arc-inconsistency count of value

a of variable i based on G, $dac_{ia}(G)$, is defined as the number of variables in PRED(i, G) which are arc-inconsistent with (i, a).

Regarding lower bound, it is clear that $\sum_{i \in X} \min_a(dac_{ia}(G))$ is a lower bound of the number of inconsistencies of any complete assignment. During search, the expression $distance(P) + \sum_{i \in F} \min_a(ic_{ia} + dac_{ia}(G))$ is no longer a lower bound, because the addition $ic_{ia} + dac_{ia}(G)$ may duplicate inconsistencies if there are edges in G going from variables in P to variables in F. To obtain a correct lower bound, one can simply compute DAC from variables in F only. The new lower bound $LB(P, F, G^F)$ is,

$$LB(P, F, G^F) = distance(P) + \sum_{i \in F} \min_a \left(ic_{ia} + dac_{ia}\left(G^F\right)\right),$$

where G^F is the subgraph of G restricted to variables in F. Obviously, $ic_{ia} + dac_{ia}(G^F)$ can be added because they always record different inconsistencies. An example of graph-based DAC for lower bound computation appears in Fig. 2, where the selected directed graph differs in one edge from the graph induced under lexicographical ordering of future variables in Fig. 1. The lower bound associated with value b of future variable j, $LB(P, F, G^F)_{jb}$, is as usual,

$$LB(P, F, G^F)_{jb} = distance(P) + ic_{jb} + dac_{jb}\left(G^F\right) + \sum_{i \in F,\, i \neq j} \min_a \left(ic_{ia} + dac_{ia}\left(G^F\right)\right).$$

During search the set F of future variables changes. As a consequence, the subgraph G^F changes and DAC based on it also change. To efficiently compute DAC at each search node, we introduce a new data-structure *GivesDac*, which records the individual potential contribution of each variable to DAC in both directions for each constraint. If $i, j \in X$, $a \in D_i$, $GivesDac(i, a, j)$ is *true* if j is arc-inconsistent with (i, a), and *false* otherwise. *GivesDac* requires O(ed) space. During search, we compute DAC at each node as follows. Let F and F' be two sets of future variables such that they differ in the current variable i, $F = F' \cup \{i\}$. DAC based on $G^{F'}$ are computed from DAC based on G^F by removing the contribution of variable i to DAC of variables in F'. This contribution is recorded in *GivesDac*, so the required computation is,

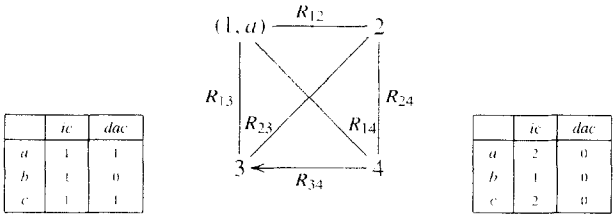
$$dac_{jb}(G^{F'}) = \begin{cases} dac_{jb}(G^F) - 1 & \text{if } GivesDac(j, b, i) = true \text{ and} \\ & \quad (i, j) \in \text{EDGES}(G^F), \\ dac_{jb}(G^F) & \text{otherwise.} \end{cases}$$

Incidentally, PFC with graph-based DAC is no longer subject to static variable ordering, so any dynamic ordering can be used.

2.2. Dynamic graph selection

Values of graph-based DAC depend on the selected directed subgraph G^F. The actual contribution of DAC to the lower bound depends on their addition to IC, which change during search. Given that any directed constraint graph is suitable for DAC computation, we can dynamically change the subgraph G^F looking for the *best* subgraph for the current

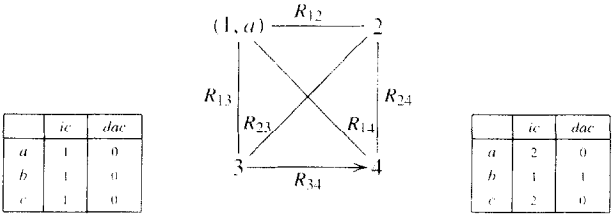
$P = \{(1, a), (2, a)\}$, $F = \{3, 4\}$, $\text{EDGES}(G_1^F) = \{(4, 3)\}$, $LB(P, F, G_1^F) = 1 + 2 = 3$



	ic	dac
a	1	1
b	1	0
c	1	1

	ic	dac
a	2	0
b	1	0
c	2	0

Reversing edge: $(4, 3) \longrightarrow (3, 4)$, $\text{EDGES}(G_2^F) = \{(3, 4)\}$, $LB(P, F, G_2^F) = 1 + 3 = 4$



	ic	dac
a	1	0
b	1	0
c	1	0

	ic	dac
a	2	0
b	1	1
c	2	0

Fig. 3. After assigning a to variable 2, reversing edge (4,3) causes to increase the lower bound.

IC, that is, the subgraph causing the largest increment to the lower bound. However, we have proved that finding the best subgraph is an NP-hard problem [11]. We avoid this problem by restricting ourselves to find a *good* subgraph, by local optimization methods (see Section 4 for details).

In the optimization process, the only elementary change allowed in G^F is edge reversal (the set of nodes is fixed by F). Because of that, this approach is called *reversible* DAC. It requires updating the DAC of a variable if some of its incoming or outcoming edges are reversed. An example of edge reversal appears in Fig. 3.

When an edge is reversed, DAC are updated as follows. Let G_1^F and G_2^F be two graphs that differ in one reversed edge: $(i, j) \in \text{EDGES}(G_1^F)$, $(j, i) \in \text{EDGES}(G_2^F)$. DAC based on G_2^F are computed from DAC based on G_1^F by removing the contribution of i and adding the contribution of j. Given that individual contributions are recorded in *GivesDac*, DAC updating is as follows,

$$dac_{jb}(G_2^F) = \begin{cases} dac_{jb}(G_1^F) - 1 & \text{if } GivesDac(j, b, i) = true, \\ dac_{jb}(G_1^F) & \text{if } GivesDac(j, b, i) = false, \end{cases}$$

$$dac_{ia}(G_2^F) = \begin{cases} dac_{ia}(G_1^F) + 1 & \text{if } GivesDac(i, a, j) = true, \\ dac_{ia}(G_1^F) & \text{if } GivesDac(i, a, j) = false, \end{cases}$$

$$dac_{kc}(G_2^F) = dac_{kc}(G_1^F) \quad \text{if } k \neq i, k \neq j.$$

Regarding value pruning, the same strategy can be applied. A value b of a future variable j can be pruned if there exists some subgraph (G^F) such that $LB(P, F, G^F)_{jb}$ reaches UB. Therefore, we can also modify the subgraph in order to find the best edges to prune future values.

Finding a good subgraph may not cause benefits if the final lower bound does not reach UB, because no pruning is done. To prevent useless efforts, it is interesting to estimate how much we can improve the lower bound. It is easy to see that, for any future variable i, any value a and any subgraph G^F, $dac_{ia}(G^F) \leqslant ac_{ia}(F)$, where $ac_{ia}(F)$ is the full arc-inconsistency count for value a of variable i computed from variables in F. Then, substituting DAC by AC in the lower bound expression, we get $\overline{LB}(P, F)$ which is an upper bound of the best lower bound we could find among all possible directed subgraphs G^F,

$$\overline{LB}(P, F) = distance(P) + \sum_{i \in F} \min_a (ic_{ia} + ac_{ia}(F)).$$

$\overline{LB}(P, F)$ is not a lower bound, but it is greater than or equal to the best graph-based lower bound at the current node. Therefore, if the above expression is lower than the current upper bound, there is no point in searching for better subgraphs because no subgraph will increase the lower bound enough to prune. The same analysis holds when looking for a good subgraph for value pruning. If value b of future variable j is considered for pruning, the expression,

$$\overline{LB}_{jb}(P, F, G^F) = distance(P) + ic_{jb} + ac_{jb}(F) + \sum_{i \in F,\, i \neq j} \min_a (ic_{ia} + dac_{ia}(G^F))$$

is greater than or equal to any graph-based lower bound associated with value b obtainable by reversing edges starting or ending in j. Therefore, if the above expression is lower than the current upper bound, there is no point in looking for a better subgraph by reversing those edges because none of those subgraphs will increase the lower bound enough to prune b.

3. Maintaining reversible DAC

The PFC-DAC algorithm computes DAC in a preprocessing step, and DAC do not change during search. This means that all recorded directed arc-inconsistencies hold before search starts. However, PFC-DAC prunes future values during its execution. This may lead to new directed arc-inconsistencies which are not recorded in the initial DAC because these are not updated during search. The *maintaining* reversible DAC approach consists in keeping DAC updated during search, considering value deletions in current domains. Maintained DAC are always higher than or equal to precomputed DAC, so they cause a higher lower bound: more branches may be pruned, more value deletions can occur, which are again propagated, etc. Maintaining DAC may cause a cascade effect that will anticipate dead-end detections.

This idea is similar to *maintaining arc-consistency* (MAC) in the CSP context [10]. Both algorithms follow the same idea: detecting arc-inconsistencies caused by pruned values. However, differently from MAC, new directed arc-inconsistencies do not cause immediate value pruning. They are recorded in the corresponding DAC, which are later used for lower bound computation, value pruning, etc. To maintain DAC, any arc consistency algorithm

(along with adequate data structures) can be adapted to propagate the effect of value removal in both directions for each constraint.

4. The PFC-MRDAC algorithm

For the sake of simplicity and brevity, we only give a high-level description of the algorithm (e.g., context restorations after value deletions, edge reversals and *ic/dac* modifications are left behind the scene). For the finest level of detail, we urge the reader to fetch the source code, available at `http://www.lsi.upc.es/~larrosa/PFC-MRDAC`. A first version of the algorithm was detailed in [8], although it has been significantly improved since. Differences with this initial version are emphasized in the sequel. The code presented depends on a specific propagation mechanism with its data-structures. Any modern arc-consistency propagation mechanism (AC4, AC6, AC7, ...) can be used. A *DeletionStream* is used to keep a list of values whose deletion must be propagated.

The general principle of the algorithm is to rely first on available lower bounds $LB(P, F, G^F)$ and $LB(P, F, G^F)_{jb}$, obtained using the graph inherited from the parent node. If these lower bounds are not large enough to backtrack, better lower bounds are sought by optimizing the directed graph and propagating deletions in each case. Before the main function MRDAC is called, one should initialize the arc-consistency propagation data-structures which will also initialize the dac_{ia} and $GivesDac_{iaj}$ data-structures. The initialization is straightforward and not described here. It can be performed in time $O(ed^2)$.

After a value a has been assigned to a variable i, and if $LB(P, F, G^F)$ is lower than the current upper bound *Bestd*, the LookAhead function is in charge of updating the ic_{jb} of every value of all future variables connected to i and also to detect a first set of prunable values on all future variables. Prunable values are those (j, b) such that $LB(P, F, G^F)_{jb}$ is larger than the current upper bound. Contrary to the initial version described in [8], each value deletion is added to *DeletionStream* instead of being immediately propagated.

If LookAhead causes no domain wipe-out, the GreedyOpt function tries to improve the lower bound $LB(P, F, G^F)$ by optimizing the directed graph G^F inherited from the parent node. The problem being NP-hard [11], we use a simple greedy local search algorithm: for each $(i, j) \in \text{EDGES}(G^F)$, we check whether reversing it into (j, i) may increase the lower bound or not. Let $a \in D_i$ and $b \in D_j$ be values having the minimum $(ic + dac)$ of variables i and j at the current node. Reversing edge (i, j) may increase the lower bound only when it does not contribute to dac_{jb} and its reversal (j, i) will contribute to dac_{ia}. In terms of *GivesDac*, these conditions are expressed as follows,

$$GivesDac(j, b, i) = false \text{ and } GivesDac(i, a, j) = true$$

If this condition does not hold, reversing (i, j) cannot cause any improvement to the lower bound. Therefore, only edges satisfying the above condition are reversed. The local search process terminates when no edge reversal can produce a lower bound increment. The resulting subgraph is noted as $\widehat{G}^F$.

Now, if the new lower bound is not large enough to backtrack, the Delete function performs further pruning: for each value (i, a) considered for deletion, this function first tries to delete the value using $LB(P, F, \widehat{G}^F)_{ia}$. If this is not possible and if $\overline{LB}_{ia}(P, F, G^F)$

```
MRDAC(S, d, F, FD, G^F);
if F = ∅ then
    Bestd ← d;
    BestS ← S;
else
    i ← PopAVariable(F);
    while (FD_i ≠ ∅) do
        a ← PopAValue(FD_i);
        Newd ← d + ic_ia + dac_ia;
        if (Newd + Σ_{j∈F} min_{b∈FD_j}(ic_jb + dac_jb) < Bestd) then
            newFD ← LookAhead(i, a, F, FD);
            if (¬WipeOut(newFD)) then
                NewG^F ← GreedyOpt(G^F, F, newFD);
                if (Newd + Σ_{j∈F} min_{b∈newFD_j}(ic_jb + dac_jb) < Bestd) then
                    newFD ← Delete(F, NewFD, NewG^F);
                    if (¬WipeOut(newFD)) then
1.1                     NewFD ← PropAndDel(F, newFD);
                        if (¬WipeOut(newFD)) then MRDAC(S ∪ {(i, a)}, Newd, F, NewFD, NewG^F);
```

Function 1. PFC-MRDAC main function, S is the current assignment, d its distance, F and FD are the future variables and their domains, G^F is the current directed graph.

```
LookAhead(i, a, F, FD);
foreach j ∈ F do
    foreach b ∈ FD_j do
        if ((j, i) ∈ EDGES(G) and ¬GivesDac(i, a, j)) or
           ((i, j) ∈ EDGES(G) and ¬GivesDac(j, b, i)) then
            if (Inconsistent(i, a, j, b)) then Increment(ic_jb);
        if (Newd + Σ_{k∈F−{j}} min_{c∈FD_k}(ic_kc + dac_kc) + dac_jb + ic_jb ≥ Bestd) then
            Prune(j, b);
            DeletionStream ← DeletionStream ∪ {(j, b)};
return Updated domains;
```

Function 2. PFC-MRDAC look-ahead function, (i, a) is the assignment to propagate, F is the set of future variables.

is larger than the current upper bound, it temporarily reverses all edges such that $GivesDac(i, a, j)$ is *true*. If the resulting lower bound LB_{ia} is large enough, (i, a) is deleted and added to *DeletionStream*. The subgraph $\widehat{G}^F$ is restored and the process iterates considering a new future value. This local optimization of the graph for each value was not present in the initial version of the algorithm [8]. It significantly improves performance. If no wipe-out occurs, the PropAndDel function propagates all pending deletions, and updates DAC accordingly (which may cause further deletions), until a fixpoint is reached or a wipe-out occurs. It is not detailed here and it can rely on any modern AC propagation mechanism. The only difference with classical AC behavior is that when a value (i, a) loses all support

```
GreedyOpt(G, F, FD):
Stop ← false;
while ¬Stop do
    SaveMin ← Σ_{j∈F} min_{b∈FD_j}(ic_jb + dac_jb);
    foreach i, j ∈ F s.t. (i, j) ∈ EDGES(G) do
        MinFrom ← min_{a∈FD_i}(ic_ia + dac_ia);
        MinTo ← min_{b∈FD_j}(ic_jb + dac_jb);
        if (¬GivesDac(j, argmin_{b∈FD_j}(ic_jb + dac_jb), i) and
            GivesDac(i, argmin_{a∈FD_i}(ic_ia + dac_ia), j)) then
            Reverse((i, j), G);
            if (min_{a∈FD_i}(ic_ia + dac_ia) + min_{b∈FD_j}(ic_jb + dac_jb) < MinFrom + MinTo) then
                Reverse((j, i), G);
    if (SaveMin = Σ_{j∈F} min_{b∈FD_j}(ic_jb + dac_jb)) then Stop ← true;
return Updated graph
```

Function 3. Finding a "good" directed graph. G is the current directed graph.

```
Reverse((i, j), G);
EDGES(G) ← EDGES(G) − {(i, j)} ∪ {(j, i)};
foreach a ∈ D_i do if GivesDac(i, a, j) then Increment(dac_ia);
foreach b ∈ D_j do if GivesDac(j, b, i) then Decrement(dac_jb);
```

Function 4. Reverse edge (i, j) in G and update dac accordingly.

```
Delete(F, FD, G);
foreach i ∈ F do
    foreach a ∈ FD_i do
        if (Newd + Σ_{k∈F−{i}} min_{c∈FD_k}(ic_kc + dac_kc) + dac_ia + ic_ia ≥ Bestd) then
            Prune(i, a);
            DeletionStream ← DeletionStream ∪ {(i, a)};
        else if (Newd + Σ_{k∈F−{i}} min_{c∈FD_k}(ic_kc + dac_kc) + ac_ia + ic_ia ≥ Bestd) then
            Stack ← ∅;
            foreach j ∈ F s.t. (i, j) ∈ EDGES(G) do
                if (GivesDac(i, a, j) = true) then
                    Reverse((i, j), G);
                    Stack ← Stack ∪ {(j, i)};
            if (Newd + Σ_{k∈F−{i}} min_{c∈FD_k}(ic_kc + dac_kc) + dac_ia + ic_ia ≥ Bestd) then
                Prune(i, a);
                DeletionStream ← DeletionStream ∪ {(i, a)};
            foreach e ∈ Stack do Reverse(e, G);
return Updated domains
```

Function 5. Prunes values using a locally optimized lower bound.

on a constraint R_{ij}, the value (i, a) is not deleted. Instead, $GivesDac(i, a, j)$ becomes *true* and the corresponding dac_{ia} is incremented if the edge (i, j) is currently directed towards i. This may increase the minimum of $(ic_{ia} + dac_{ia})$. The condition for pruning used here is the same as in the LookAhead function.

5. Experimental results

In our first experiment, we have evaluated the performance of our algorithms on over-constrained binary random CSP. A binary random CSP class is characterized by $\langle n, d, p_1, p_2\rangle$ where n is the number of variables, d the number of values per variable, 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 [9]. Using this model, we have experimented on the following problem classes:

(1) $\langle 10, 10, 1, p_2\rangle$, (2) $\langle 15, 5, 1, p_2\rangle$,

(3) $\langle 15, 10, 50/105, p_2\rangle$, (4) $\langle 20, 5, 100/190, p_2\rangle$,

(5) $\langle 25, 10, 37/300, p_2\rangle$, (6) $\langle 40, 5, 55/780, p_2\rangle$.

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 by three algorithms: PFC-DAC as described in [7] and PFC maintaining reversible DAC using an AC6-based propagation mechanism. As in [8], we also tested out a simplified version of PFC-MRDAC, called PFC-RDAC that does not propagate deletions (the call to the PropAndDel function on line 1.1 of the main function is not performed). PFC-DAC uses *forward degree*, breaking ties with *backward degree* [7] as static variable ordering. PFC-RDAC and PFC-MRDAC use *domain size* divided by *forward degree* as dynamic variable ordering. Values are always selected by increasing $ic + dac$. All three algorithms share code and data structures whenever it is possible. Experiments were performed using a Sun Sparc 2 workstation.

Fig. 4 reports the average visited nodes to solve the six problem classes. For all problem classes, it is observed that PFC-RDAC and MRDAC visit significatively less nodes than PFC-DAC, and this gain increases with problem tightness. PFC-MRDAC visits less nodes than PFC-RDAC, and this gain tends to decrease with problem tightness. Regarding computational effort, Fig. 5 reports the average cpu-time required to solve the six problem classes (since the overhead produced by RDAC and MRDAC is consistency-check free, we use cpu-time instead the number of consistency checks). Cpu-time results are in agreement with those of visited nodes: PFC-RDAC improves PFC-DAC in practically all problem classes, and the gain grows with problem tightness. PFC-RDAC can be up to 2000 times faster than PFC-DAC on the tightest sparse instances. Typical improvement ratios range from 2 to 30 for tightly constrained problems. Regarding PFC-MRDAC, it visits less nodes than PFC-RDAC but it performs more work per node. As global effect, PFC-MRDAC performance is very close to that of PFC-RDAC. Typically, maintaining DAC on dense

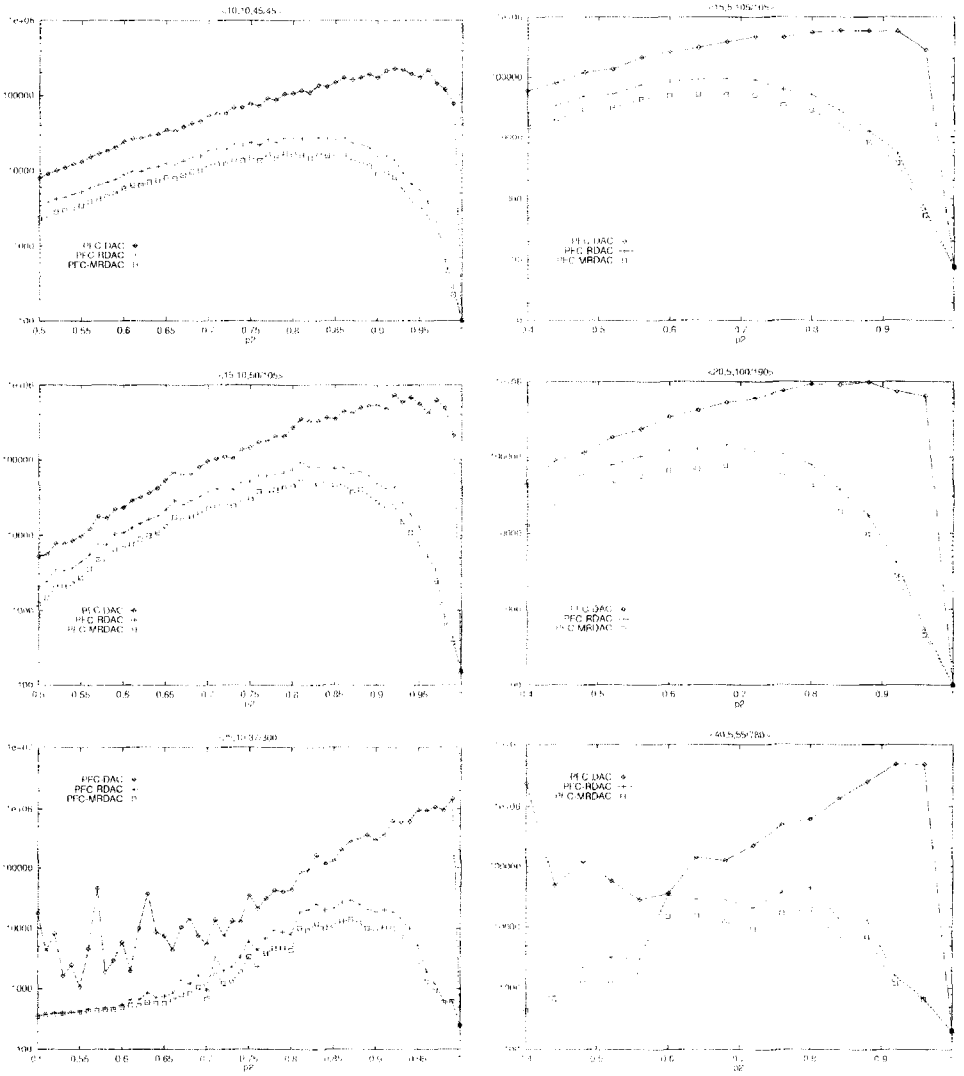


Fig. 4. Average visited nodes versus tightness for six classes of binary random problems (a logarithmic scale is used on the y axis).

problems is not cost effective, and PFC-RDAC is slightly faster than PFC-MRDAC. The situation is reversed on sparse problems where PFC-MRDAC is from 1.2 to 1.75 times faster than PFC-RDAC. We have observed that using a dynamic variable ordering in conjunction with PFC-MRDAC causes only a small performance improvement.

Our second experiment considers the Radio Link Frequency Assignment Problem (RLFAP). It is a communication problem where the goal is to assign frequencies to a set of radio links in such a way that all the links may operate together without noticeable

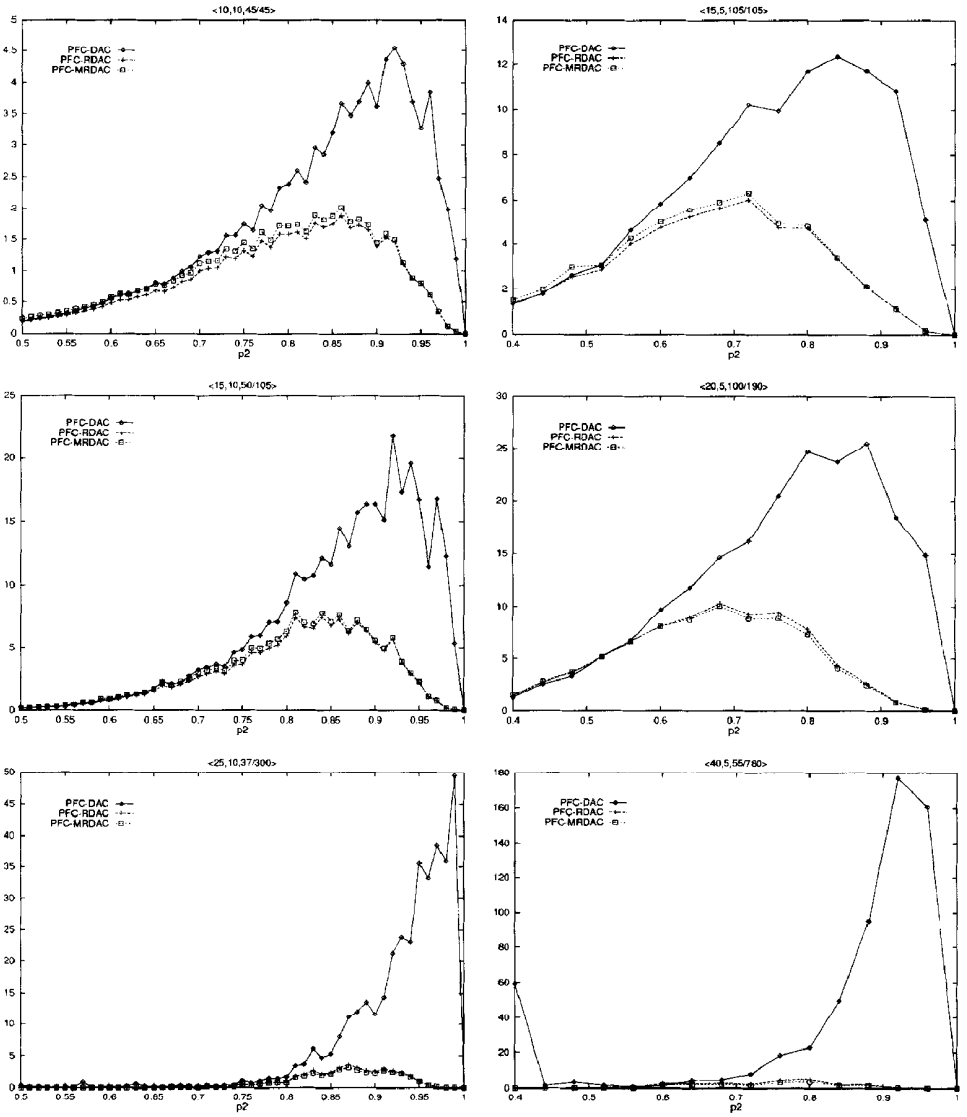


Fig. 5. Average CPU versus tightness for six classes of binary random problems.

interference [3]. Some RLFAP instances can be naturally cast as *weighted* binary Max-CSP where each forbidden tuple has an associated penalty cost. We have extended DAC-based algorithms to this framework and tested them on four publicly available RLFAP subinstances called CELAR6-SUB$_i$ ($i = 1, \ldots, 4$) in [3]. Each CELAR6-SUB$_i$ is a sub-instance of CELAR6-SUB$_{i+1}$ and is therefore presumably easier to solve. As in the previous experiment, each instance is solved with the same three algorithms. The same static and dynamic variable ordering heuristics are used. However, they are extended to the

Table 1
Cost of solving instances 1 to 4 of the CELAR6 problems

	PFC-DAC		PFC-RDAC		PFC-MRDAC	
	Nodes	Cpu time	Nodes	Cpu time	Nodes	Cpu time
SUB_1	8.1×10^8	5.2×10^4	1.9×10^7	3.5×10^3	1.8×10^6	2.6×10^3
SUB_2	4.8×10^9	3.0×10^5	1.2×10^8	2.9×10^4	1.6×10^7	2.3×10^4
SUB_3	1.2×10^{10}	8.4×10^5	2.7×10^8	7.1×10^4	3.7×10^7	6.8×10^4
SUB_4			1.2×10^9	3.4×10^5	1.2×10^8	2.6×10^5

weighted case: each constraint has a contribution to its connecting variables degree equal to the sum of its penalty costs.

Table 1 shows the cost required to prove optimality (i.e., the upper bound is initialized with the optimum cost) by DAC-based algorithms. We report both number of visited nodes and cpu-time in seconds. PFC-RDAC clearly outperforms PFC-DAC regarding both time and visited nodes. In the first three instances PFC-RDAC is more than ten times faster than PFC-DAC. In the last instance PFC-DAC was aborted when it did not finish its execution within ten times the time required by PFC-RDAC. Regarding PFC-MRDAC, it visits about ten times less nodes than PFC-RDAC. However, the gain is not as large in terms of cpu-time because of the overhead of propagation. Nonetheless, PFC-MRDAC is always the fastest algorithm and its speedup over PFC-RDAC is clearly worth.

The CELAR-SUB_3 and SUB_4 instances have also been solved using an improved version of RDS [13] informed with good initial upper bounds. Their resolution took respectively more than 2.5×10^6 and 5×10^6 seconds of Sparc5 cpu-time (private communication of Simon de Givry [4]). Although they cannot be directly compared with our results (different initialization and different computers), they give an approximate idea of the behaviour of different algorithms with problem size.

6. Conclusions

Branch and bound efficiency deeply depends on the lower bound quality. The reversible DAC-based lower bound we have introduced strongly improves existing lower bounds both in terms of branch pruning and value deletions. As a global effect, reversible DAC has enhanced the efficiency of the initial DAC-based algorithm by several orders of magnitude. It is worth noting that we have followed a rather simple local optimization strategy for subgraph optimization. More sophisticated strategies may provide further benefits.

Maintaining RDAC causes a significant decrease in the number of visited nodes, although it does not always pay-off in problems where all constraints have the same importance. In the weighted CELAR problems, maintaining RDAC provides an important speed-up. This aspect may be of great interest for other real Max-CSP problems as well.

For real problems involving both hard and soft constraints, the algorithms we introduced are ideal candidates for an hybridation with MAC algorithms [10]: contrary to other recent algorithms for solving Max-CSP [7,13,15], they can use dynamic variable orderings and will not artificially restrict the hybrid to static orderings.

Acknowledgements

The research of Javier Larrosa and Pedro Meseguer is supported by the Spanish CICYT project TIC96-0721-C02-02. We thank Michel Lemaître and Gérard Verfaillie, who read a previous version of this paper providing useful comments. We also thank Miquel Angel Garcia for revising the paper text.

References

[1] M.S. Affane, H. Bennaceur, A weighted arc consistency technique for Max-CSP, in: Proc. ECAI-98, Brighton, UK, 1998, pp. 209–213.
[2] S. Bistarelli, U. Montanari, F. Rossi, Constraint solving over semirings, in: Proc. IJCAI-95, Montréal, Quebec, 1995.
[3] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, J.P. Warners, Radio link frequency assignment, Constraints (to appear).
[4] S. de Givry, Algorithmes d'optimisation sous contraintes étudiés dans un cadre temps-réel, Ph.D. Thesis, CERT/ONERA-SupAéro, 1988.
[5] R. Dechter, J. Pearl, Network-based heuristics for constraint-satisfaction problems, Artificial Intelligence 34 (1988) 1–38.
[6] E.C. Freuder, R.J. Wallace, Partial constraint satisfaction, Artificial Intelligence 58 (1992) 21–70.
[7] J. Larrosa, P. Meseguer, Exploiting the use of DAC in Max-CSP, in: Proc. CP-96, Boston, MA, 1996, pp. 308–322.
[8] J. Larrosa, P. Meseguer, T. Schiex, G. Verfaillie, Reversible DAC and other improvements for solving Max-CSP, in: Proc. AAAI-98, Madison, WI, 1998, pp. 347–352.
[9] P. Prosser, Binary constraint satisfaction problems: Some are harder than others, in: Proc. ECAI-94, Amsterdam, The Netherlands, 1994, pp. 95–99.
[10] D. Sabin, E.C. Freuder, Contradicting conventional wisdom in constraint satisfaction, in: Proc. ECAI-94, Amsterdam, The Netherlands, 1994, pp. 125–129.
[11] T. Schiex, Maximizing the reversible DAC lower bound in Max-CSP is NP-hard, Technical Report 1998/02, INRA, July 1998.
[12] T. Schiex, H. Fargier, G. Verfaillie, Valued constraint satisfaction problems: hard and easy problems, in: Proc. IJCAI-95, Montréal, Quebec, 1995, pp. 631–637.
[13] G. Verfaillie, M. Lemaître, T. Schiex, Russian doll search, in: Proc. AAAI-96, Portland, OR, 1996, pp. 181–187.
[14] R. Wallace, Directed arc consistency preprocessing, in: M. Meyer (Ed.), Selected papers from the ECAI-94 Workshop on Constraint Processing, Lecture Notes in Computer Science, Vol. 923, Springer, Berlin, 1995, pp. 121–137.
[15] R. Wallace, Enhancements of branch and bound methods for the maximal constraint satisfaction problem, in: Proc. AAAI-96, Portland, OR, 1996, pp. 188–195.