Partial Lazy Forward Checking *

Javier Larrosa[1] and **Pedro Meseguer**[2]
[1] Universitat Politècnica de Catalunya, Dep. Llenguatges i Sistemes Informàtics
Jordi Girona Salgado 1-3, 08034 Barcelona, SPAIN
E-mail: larrosa@lsi.upc.es
[2] Institut d'Investigació en Intel.ligència Artificial, CSIC
Campus UAB, 08193 Bellaterra, SPAIN.
E-mail: pedro@iiia.csic.es

Abstract. Partial forward checking (PFC) may perform more consistency checks than really needed to detect dead-ends in MAX-CSP. After analyzing PFC, we have identified four causes of redundant check computation: (a) unnecessary lookahead when detecting an empty domain, (b) not always using the better bounds for future value pruning, (c) computing in advance inconsistency counts, and (d) lookahead is performed on the whole set of future variables. We present the partial lazy forward checking (PLFC) algorithm, which follows a lazy approach delaying as much as possible inconsistency count computation, keeping updated the contribution of future variables to the lower bound. This algorithm avoids the causes of redundant checks identified for PFC. It can be easily combined with DACs, producing the PLFC-DAC algorithm. Empirical results on random problems show that PLFC-DAC outperforms previous algorithms in both consistency checks and CPU time.

1 Introduction

Constraint satisfaction problems (CSP) consider the assignment of values to variables under a set of constraints. A solution is a total assignment satisfying every constraint. If such assignment does not exist, the problem is overconstrained, and it may be of interest to find a total assignment satisfying as many constraints as possible. This problem is called the maximal constraint satisfaction problem (MAX-CSP), and a solution is a total assignment satisfying the maximum number of constraints. MAX-CSP is of interest in several areas of application [Fox, 87; Feldman and Golumbic, 90; Bakker *et al.*, 93].

Partial forward checking (PFC)[1] is one of the best algorithms for MAX-CSP [Freuder and Wallace, 92]. It is a branch and bound algorithm including forward checking in order to anticipate dead-ends before they actually occur. To detect dead-ends, PFC computes a lower bound of the number of unsatisfied constraints from (i) the set of past variables, and (ii) the effect of past variables on future ones, by using *inconsistency counts* (IC). Recently, this lower bound has been improved by including

* This research has been supported by the Spanish CICYT under the project #TIC96-0721-C02-02.

[1] In [Freuder and Wallace, 92] several versions of partial forward checking are presented. We use here as PFC their P-EFC3 algorithm.

unsatisfied constraints from the set of future variables. This has been implemented using *directed arc-consistency counts* (DAC) [Wallace, 94], and the reported empirical results show a clear improvement in performance with respect to pure PFC. DAC are computed in a preprocessing step, which is later followed by PFC for variable instantiation. DAC usage has been enhanced by [Larrosa and Meseguer, 96] in their PFC-DAC algorithm[2], which has a better performance than the initial combination of PFC and DAC proposed by [Wallace, 94].

In this paper, we show that the PFC greedy strategy of performing lookahead on every future variable may require more consistency checks than really needed to detect dead-ends. A careful analysis of PFC behaviour has allowed us to identify the following causes of redundant consistency checks: (a) PFC may do unnecessary lookahead when detecting an empty domain, performing checks that are never used, (b) PFC does not always use the better bounds for future value pruning, which may cause to collect more inconsistencies than really needed and therefore, it may require the computation of an extra number of consistency checks, (c) computing in advance inconsistency counts may cause more checks than really needed to prune a value of the current variable, and (d) lookahead is performed on the whole set of future variables, possibly doing more work than really needed to prune a value of the current variable.

Based on this analysis we take a lazy approach that essentially consists on *delaying as much as possible the IC computation, while keeping updated the contribution of future variables to the lower bound.* This lazy strategy is further developed in the following points: (α) future domains are not pruned in advance, (β) the lowest amount of lookahead is done on future domains, to compute the minimal contribution of future variables to the lower bound, and (γ) IC updating and lookahead can be stopped prior completion. Following these points we have developed the *partial lazy forward checking algorithm* (PLFC). Since future domains are not pruned and ICs are not completely updated, PLFC cannot compute without an extra cost some dynamic variable ordering heuristics (minimum domain or largest mean of ICs).

The lazy approach can be easily combined with DACs, just in the same way it was combined with the greedy algorithm, producing the PLFC-DAC algorithm. This combination is particularly adequate: because of DAC usage, this algorithm requires static variable order, so the inability of PLFC to compute some dynamic variable ordering heuristics is not really a drawback, while keeping the advantages caused by DACs for a better lower bound.

This paper is organized as follows. In Section 2 we present related approaches to the paper topic. In Section 3 we provide some preliminaries and definitions required in the rest of the paper. In Section 4, we analyze the behaviour of PFC, showing that it may perform more consistency checks than required, and we present our lazy approach to PFC, the PLFC algorithm. This algorithm is combined with the DAC usage, producing the PLFC-DAC algorithm. In Section 5, we give empirical results of PLFC-DAC on random problems, showing a clear performance improvement in both

---

[2] In [Larrosa and Meseguer, 96] this algorithm was called P-EFC3+DAC2. For simplicity reasons, we refer it as PFC-DAC from now on.

number of consistency checks and CPU time with respect to PFC-DAC. Finally, in Section 6 we summarize the conclusions of this work.

2 Related Work

The simplest algorithm for MAX-CSP is *depth-first branch and bound*, which traverses completely the whole search tree looking for the leaf node violating a minimal number of constraints. In terms of variable assignments, an internal node represents a partial assignment while a leaf node represents a total one. The distance of a node is the number of constraints violated by its assignment. At each internal node, the algorithm computes a *lower bound* of the distance of any leaf node below the current one. It also keeps track of the best leaf node (complete assignment violating less constraints) found so far, and its *best distance* is an *upper bound* of the allowable distance of any future leaf node. When the lower bound is greater than or equal to the upper bound, the current node (and all its successors) can be pruned because no leaf node below the current one will improve the best distance already found. This basic algorithm can be enhanced with more sophisticated strategies, and in particular we consider *partial forward checking* (PFC), which evaluates the impact of the current partial assignment on future variables computing ICs, which allows PFC to improve the lower bound. For a detailed description of other algorithms, see [Freuder and Wallace, 92]. Regarding dynamic variable ordering with PFC, variables may ordered either by the largest mean of inconsistency counts in their domains or by minimum domain size, and dynamic value ordering considers values by increasing IC [Freuder and Wallace, 92]. Two heuristics for dynamic variable and value ordering are given in [Larrosa and Meseguer, 95]. Other static variable ordering heuristics can be found in [Wallace and Freuder, 93].

DAC usage has been an important step to increase the lower bound [Wallace, 94; Larrosa and Meseguer, 96; Wallace, 96]. Another way to improve the lower bound is *russian doll search* [Verfaille et al, 96], where the whole problem is substituted by n nested subproblems such that the solution of the i-th subproblem can be used as the starting lower bound for the i+1th subproblem. Several approaches have tried to improve the upper bound at early search states by heuristic repair [Wallace, 96] and by stochastic methods [Cabon et al, 96].

The idea of lazy evaluation in not new in constraint satisfaction, although it has not been fully exploited. Regarding *forward checking* (FC), several researchers have realized that it performs more lookahead than needed to detect empty domains [Zweben and Eskey, 89; Dent and Mercer, 94; Bacchus and Grove, 95]. FC checks every feasible value of every future domain, while it would be enough to stop checking a future domain as soon as one consistent value has been found. The lazy FC approach, called *minimal forward checking*, has proved to save from 10% to 30% of consistency checks solving several problem classes [Bacchus and Grove, 95]. Regarding *arc consistency* (AC), a similar situation has been reported by [Schiex et al, 96]: AC algorithms perform more checking than needed to detect empty domains.

3 Preliminaries

A discrete binary CSP is defined by a finite set of variables $\{X_i\}$ taking values on discrete and finite domains $\{D_i\}$ under a set of binary constraints $\{R_{ij}\}$. A constraint R_{ij} is a subset of $D_i \times D_j$, containing the permitted values for X_i and X_j. The number of variables is n and, without loss of generality, we will assume a common domain D for all variables, m being its cardinality. A global solution of the CSP is an assignment of values to variables satisfying every constraint. If no solution exists the CSP is overconstrained; in this case we are interested in finding solutions satisfying a maximum number of constraints. This problem is usually referred as MAX-CSP.

Partial forward checking (PFC) is an efficient algorithm to solve MAX-CSP [Freuder and Wallace, 92]. It is a forward checking algorithm over the branch and bound schema and its code appears in Figure 1 (assuming for simplicity of presentation that variables are assigned in lexicographical order). At a given node, we denote by $\boldsymbol{P}$ and $\boldsymbol{F}$ the sets of past and future variables respectively. PFC keeps for every feasible value l of every unassigned variable X_i, the inconsistency count ic_{il} which records the number of inconsistencies that l has with the assignments of past variables. If X_i is the current variable and value l is assigned to it, the lower bound of the current partial assignment is increased in ic_{il}, as the number of inconsistencies between X_i and $\boldsymbol{P}$. The sum $\sum_{j \in \boldsymbol{F}} \min_k \{ic_{jk}\}$ is a lower bound of the number of inconsistencies that will necessarily occur when every future variable is instanciated, so it can included to compute the lower bound of the current partial assignment (lines 13 and 32). In addition, if X_i is a future variable, value l can be pruned if the current distance plus ic_{il} plus the sum of $\min_k \{ic_{jk}\}$ of other future variables is not lower than the best distance (lines 21 and 24). This value is called the lower bound associated with l. Finally, values can be heuristically ordered by increasing ICs.

PFC has been enhanced with the inclusion of DACs: given a static variable ordering, the DAC associated to a value l of a variable X_i, dac_{il}, is the number of variables which are arc-inconsistent with value l for X_i and appear after X_i in the ordering [Wallace, 94]. ICs and DACs were first combined to improve lower bound computation and better value pruning in [Wallace, 94] and later in [Larrosa and Meseguer, 96]. If X_i is the current variable and value l is assigned to it, the lower bound of the current partial assignment is increased in $ic_{il} + dac_{il}$, as the number of inconsistencies between X_i and $\boldsymbol{P}$ plus the number of inconsistencies that necessarily will appear between X_i and $\boldsymbol{F}$ when extending the current partial assignment into a total one. The sum $\sum_{j \in \boldsymbol{F}} \min_k \{ic_{jk} + dac_{jk}\}$ is a lower bound of the number of inconsistencies that will necessarily occur when every future variable is instanciated, so it can be included to compute the lower bound of the current partial assignment. In addition, if X_i is a future variable, value l can be pruned if the current distance plus $ic_{il} + dac_{il}$ plus the sum of $\min_k \{ic_{jk} + dac_{jk}\}$ of other future variables is not lower than the best distance. This value is called the lower bound associated with l (when

DACs are used). Finally, values can be heuristically ordered by increasing

```
procedure PFC  (current_solution,next_variable,distance)
1  X_i := next-variable;
2   values := sort-values(X_i);
3  while values ≠ ∅ do
4     l := first(values);
5     values := values - l;
6     if (feasible(X_i,l)) then
7        new_distance := distance + ic_il;
8       if (i = n) then
9          best_distance := new_distance;
10         best_solution := current_solution + (X_i,l);
11      else
12        if (look_ahead(X_i,l)) then
13          if (new_distance + Σ_{j>i} min_k{ic_jk} < best_distance) then
                  PFC(current_solution+(X_i,l),X_i+1,new_distance);
14          endif
15        endif
16      endif
17    endif
18 endwhile
endprocedure

function look_ahead (X_i,l)
19 for j := i+1 to n do
20    forall k∈Feasibles do
21      if (new_distance + ic_jk + Σ_{p∈F-j} min_q{ic_pq} ≥
                                  best_distance) then prune(X_j,k)
22      elsif (inconsistent(X_i,l,X_j,k)) then
23        ic_jk := ic_jk + 1;
24        if  (new_distance + ic_jk + Σ_{p∈F-j} min_q{ic_pq} ≥
                                  best_distance) then prune(X_j,k)
25        endif
26      endif
27    endforall
28    if (empty  domain(Xj)) then return false endif
29 endfor
30 return true
endfunction

function feasible (X_i,l)
31 return (distance + ic_il + Σ_{j>i} min_k{ic_jk} < best_distance)
endfunction
```

Figure 1. The PFC algorithm.

IC+DACs. The PFC-DAC algorithm [Larrosa and Meseguer, 96] can be easily obtained from PFC substituting lines 13, 21, 24 and 31 of Figure 1 by the following lines:

```
13 if (new_distance + min_{j∈F} min_k{ic_jk+dac_jk} < best_distance)
21 if (new_distance + dac_il + ic_jk + dac_jk + min_{p∈F-j} min_q{ic_pq+dac_pq} ≥
                                    best_distance) then prune(X_j,k)
24 if (new_distance + ic_jk + dac_jk + min_{p∈F-j} min_q{ic_pq+dac_pq} ≥
                                    best_distance) then prune(X_j,k)
31 return (distance+ic_il+dac_il + Σ_{j∈F} min_k{ic_jk+dac_jk} < best_distance)
```

4 PFC and Lazy Evaluation

The idea of lazy evaluation in PFC consists on delaying as much as possible the computation of inconsistency counts, producing as an inmediate consequence the delay of future value pruning. Although this may seem counterintuitive for a forward checking algorithm based on lookahead, in the following we will show that delaying inconsistency count computation may cause a decrement in the number of consistency checks performed by the algorithm, without decreasing its pruning capability. In the following, we analyze the PFC greedy behaviour to identify those situations where it may perform more consistency checks than needed.

4.1 PFC Analysis

PFC tries to anticipate pruning of future values as much as possible. For this reason, the pruning condition in future domains is checked each time the *look_ahead* function is called and, necessarily, inconsistency counts are continuously updated. We analyze the two situations where value pruning occurs:

1. *Early pruning* (lines 21 and 24, Figure 1). Value l of future variable X_i is early pruned as a side-effect of the *look_ahead* function, because the lower bound associated with l is greater than or equal to the current upper bound. In this situation, there are the following independent sources of redundant consistency check computation:
   1.1. If the domain of future variable X_i becomes empty after pruning value l, the lookahead performed from the current variable on any other future variable different from X_i is useless. Therefore, those consistency checks performed by this lookahead are redundant.
   1.2. If the domain of future variable X_i does not become empty after pruning value l, there are two further situations:
      1.2.1. The lower bound associated with value l at the current node, denoted as node A, is as follows,

$$distance(A) + ic_{il}(A) + \sum_{j \in F(A) - i} \min_k \{ic_{jk}\}$$

Let us assume that the evaluation of the pruning condition for value l is delayed to a future node in which X_i is the current variable. Calling it node B, the lower bound associated with l at B is as follows,

$$distance(B) + ic_{il}(B) + \sum_{j \in F(B)} \min_k \{ic_{jk}\}$$

At node A, the contribution of each future variable between the current one and X_i is $\min_k \{ic_{jk}\}$. At node B, the contributions of these variables are greater than or equal to the corresponding contributions at node A, and they are included in $distance(B)$, since these variables have become past variables at node B. At node A, the contribution of each future variable after X_i is $\min_k \{ic_{jk}\}$. At node B, the contributions of these variables are greater than or equal to the corresponding contributions at node A, since their $\min_k \{ic_{jk}\}$ could have increased. Therefore, the following inequality holds,

$$distance(A) + \sum_{j \in F(A) - i} \min_k \{ic_{jk}\} \leq distance(B) + \sum_{j \in F(B)} \min_k \{ic_{jk}\}$$

that is, the contribution of every variable but X_i to the lower bound associated with l at node B is greater than or equal to the same contribution at node A. Therefore, the required contribution from ic_{il} to satisfy the pruning condition of value l at node B is lower than or equal to the same contribution that is required at node A. In other words, delaying pruning evaluation until node B may require collecting a lower (and never higher) number of inconsistencies caused by l than at node A. If ic_{il} is computed at node B, checking that the pruning condition is satisfied will require a lower (and never higher) number of consistency checks than if ic_{il} is computed at node A.

1.2.2. Value l has been pruned at node A using the current upper bound. However, if l pruning is delayed to node B, in which X_i is the current variable, a better upper bound may be available. The upper bound at node B will be lower than or equal to the upper bound at node A. This will cause a less demanding pruning condition, able to be satisfied with less consistency checks than required to prune l at node A.

Analyses 1.2.1 and 1.2.2 are independent and complementary. Pruning delay may decrease the number of consistency checks needed to compute inconsistency counts because other contributions to the lower bound may increase (1.2.1). In addition, pruning delay may also decrease the number of consistency checks needed to compute inconsistency counts, because the upper bound has decreased and the pruning condition is easier to satisfy (1.2.2). Considering both analyses simultaneously magnifies their effects in consistency checks decrement. As conclusion, early pruning does not always use the better bounds to minimize the number of checks, and it may perform

more checks that required.

2. *Pruning values of the current variable* (lines 6 and 13, Figure 1). Value l of current variable X_i is pruned because its lower bound is greater than or equal to the current upper bound. In this situation, there are two independent sources of redundant consistency check computation:

   2.1. Value l is pruned before *look_ahead* (line 6, Figure 1). This implies that the upper bound has decreased from the last *look_ahead* call at the previous node (otherwise, l would have been pruned at that *look_ahead* call). PFC has computed in advance ic_{il}, checking every past variable against l when it is assigned. However, this may be more effort than required. If ic_{il} is not computed in advance, pruning can be detected at this point checking l against as few past variables as needed to satisfy the pruning condition, stopping as soon as the lower bound reaches the upper bound. Therefore, if l can be pruned checking it against a subset of past variables instead of the whole set, computing in advance inconsistency counts may cause redundant consistency checks.

   2.2. Value l is pruned after *look_ahead* (line 13, Figure 1). This function tries to update $\min_l(ic_{il})$ for every future variable. However, this may be more effort than required. It could be enough to update $\min_l(ic_{il})$ for a subset of future variables, such that the lower bound would satisfy the pruning condition. Therefore, performing the *look_ahead* function on the whole set of future variables without the possibility of early termination may cause redundant consistency checks.

4.2 Lazy Approach

From the previous analysis, we conclude that PFC may perform more consistency checks than needed due to four causes:

(a) early pruning may do unnecessary lookahead when detecting an empty domain, performing checks that are never used,

(b) early pruning does not always use the better bounds, which may cause to collect more inconsistencies than really needed and therefore, this may generate an extra number of unnecessary consistency checks,

(c) computing in advance inconsistency counts may cause more checks than really needed to prune a value of the current variable, and

(d) the *look_ahead* function is executed on the whole set of future variables, possibly doing more work than really needed to prune a value of the current variable.

Based on this analysis we take a lazy approach that essentially consists on *delaying as much as possible the IC computation, keeping updated the contribution of future variables to the lower bound.* This lazy strategy is further developed in the following points:

(α) *No early pruning*. To prevent redundant consistency checks due to causes (a), (b) and (c), values are not pruned when performing lookahead. For this reason,

pruning is performed on values of the current variable only.

(β) *Computing* $min_l\ (ic_{il}\ )$ *only*. For lower bound computation at the current node, it is not needed to compute exactly the inconsistency count ic_{il} of every value l of every future variable X_i. It is enough to have the $min_l\ (ic_{il}\ )$ for every future variable (see lines 13 and 15 of PFC, Figure 1).

(γ) *IC updating and lookahead can be stopped prior completion.* As a consequence of (α) a value is only checked for pruning when its variable has become the current one. Then, the following steps are made: (i) if needed, its inconsistency count is updated and, each time it is increased, the pruning condition for the current value is checked, and (ii) if pruning does not occur, lookahead is performed until the pruning condition is satisfied or every future variable is considered. Steps (i) and (ii) can be stopped prior completion, to prevent causes (c) and (d) respectively.

4.3 The PLFC and PLFC-DAC Algorithms

With the previous criteria, we have developed the *partial lazy forward checking* (PLFC) algorithm, and its code appears in Figure 2. PLFC keeps the same structure as PFC, substituting functions *feasible* and *look_ahead* by their lazy versions *lazy_feasible* and *lazy_look_ahead.*

```
procedure PLFC  (current_solution,next_variable,distance)
1  X_i := next_variable;
2   values := sort-values(X_i);
3  while values ≠ ∅ do
4     l := first(values);
5     values := values - l;
6     if (lazy_feasible(X_i,l)) then
7        new_distance := distance + ic_il;
8       if (i = n) then
9          best_distance := new_distance;
10         best_solution := current_solution + (X_i,l);
11      else
12        if (lazy_look_ahead(X_i,l)) then
13            PLFC(current_solution+(X_i,l),X_i+1,new_distance);
14        endif
15      endif
16    endif
17 endwhile
endprocedure
```

Figure 2. The PLFC algorithm.

```
function lazy_feasible (X_i,l)
18  feasible_so_far:= true;
19  stop:= false;
20 while (feasible_so_far  and not stop) do
21   if (distance+ic_il + Σ_{j>i} min_k{ic_jk} < best_distance) then
22      feasible_so_far:= false
23   else
24     if (ic_level_il = i-1) then stop:=true;
25     else
26        ic_level_il := ic_level_il + 1;
27        if (inconsistent(X_i,l,X ic_levelil,val( ic_level_il))) then
28          ic_il := ic_il + 1;
29        endif
30     endif
31   endif
32 endwhile
33 return  feasible_so_far;
endfunction

function lazy_look_ahead (X_i,l)
34 for j := i+1 to n do
35    update_value_with_min_IC(X_j)
36   if (new_distance + Σ_{j>i} min_k{ic_jk} < best_distance) return false
37   endif
38 endfor
39 return true;
endfunction

procedure update_value_min_IC (X_j)
40  k:= value_with_min_IC;
41  stop:= false;
42 while (not stop) do
43   if (ic_level_jk =i) then stop:= true
44   else
45     ic_level_jk := ic_level_jk + 1;
46     if (inconsistent(X_j,k,X ic_leveljk,val( ic_level_jk))) then
47          ic_jk := ic_jk + 1;
48     endif
49      k:=value_with_min_IC;
50   endif
51 endwhile
endprocedure
```

Figure 2 (cont.). The PLFC algorithm.

Given that PLFC does not keep updated the inconsistency counts of every future value, it needs an additional data structure *ic_level*, indexed by variables and values. ic_level_{il} indicates the update level of ic_{il} with respect to the assignment order of past variables. For example, $ic\_level_{il} = u$ and $ic_{il} = v$ mean that value l of future variable X_i is inconsistent with v assignments of u first past variables. We say that ic_{il} is updated if its update level is equal to the last assigned variable.

The *lazy_feasible* (X_i,l) function sequentially updates ic_{il} starting from the level $ic\_level_{il} + 1$ to the current level. The pruning condition for value l is checked at each step (line 21, Figure 2). Therefore, if the pruning condition holds during the updating, this process is stopped. If the pruning condition is not satisfied, the updating process continues until every past variable is considered.

The *lazy_look_ahead* (X_i,l) function updates $\sum_{j \in F} \min_k \{ic_{jk}\}$ after the assignment of l to the current variable X_i. If the function call terminates successfully, it is guaranteed that the minimum inconsistency count among the values of every future variable it updated. This function works as follows. For each future variable, it selects the current minimum inconsistency count and increases its update level until either (i) it becomes updated, or (ii) it is increased. If it becomes updated, this inconsistency count remains the minimum. If this inconsistency count is increased, the new minimum inconsistency count is selected, on which the same process is repeated. Observe that, with this lazy approach, the minimum inconsistency count of a future variable is updated to the current variable level, but the rest of inconsistency counts are only computed up to the level where they equal to or surpass by 1 the minimum inconsistency count of that variable. In addition, *lazy_look_ahead* checks the pruning conditions each time a variable has its minimum inconsistency count updated (line 36, Figure 2). It is worth noting that, in this process, the lower bound is repeatedly computed, adding at each iteration a new updated minimum inconsistency count of a future variable. As soon as the pruning condition is satisfied, the process stops.

PLFC can easily extended to include DACs, in the same way that PFC, producing the PLFC-DAC algorithm. This algorithm can be obtained by replacing lines 21, 35, 36, 40 and 49 by the following ones,

21 **if** (distance + ic_{il} + dac_{il} + $\sum_{j>i}$ $\min_k\{ic_{jk}+dac_{jk}\}$ < best_distance)

35 update_value_with_min_IC+DAC(X_j)

36 **if** (new_distance + $\sum_{j>i}$ $\min_k\{ic_{jk}+dac_{jk}\}$ < best_distance) **return false**

40 k:= value_with_min_IC+DAC;

49 k:= value_with_min_IC+DAC;

and replacing **procedure** update_value_min_IC (X_j) by **procedure** update_value_min_IC+DAC (X_j).

5 Empirical Results

We have compared the performance of PLFC-DAC versus PFC-DAC on random CSPs. A random problem is characterized by $<n,m,p_1,p_2>$ where n is the number of variables, m is the number of values for each variable, p_1 is the graph *connectivity* as the proportion of existing constraints (the number of constrained variable pairs is exactly $p_1 n(n\text{-}1)/2$), and p_2 is the constraint *tightness* as the proportion of forbidden value pairs between two constrained variables (the number of forbidden value pairs is exactly $p_2 m^{2)}$. The constrained variables and their nogoods are randomly selected [Prosser, 94]. We tested the following problem classes:

1. $<15,5,p_1,p_2>$, with p_1 taking values 25/105, 50/105, 75/105 and 105/105, and p_2 taking values from 12/25, 13/25, ..., 25/25.
2. $<10,10,p_1,p_2>$,with p_1 taking values 15/45, 25/45, 35/45 and 45/45, and p_2 taking values 50/100, 51/100, ..., 100/100.

generating for each parameter setting 50 problems, forming two sets of 2,600 and 10,200 instances respectively. All algorithms were implemented in C and run on a SUN Ultra 1.

Each problem was solved using PFC-DAC and PLFC-DAC. The static variable ordering used was decreasing forward degree, breaking ties with decreasing backward degree. This variable ordering gave a very good performance when used with PFC-DAC on previous experiments with the same set of problems [Larrosa and Meseguer, 96].

The average search effort as the number of consistency checks for the two tested classes appear in Figure 3 and Figure 4. For these classes, it is clear that PLFC-DAC outperforms PFC-DAC. It can be observed that the lazy approach saves the computation of up to 50% of consistency checks. The average CPU time of both algorithms on the two tested classes appears in Figure 5 and Figure 6. They clearly show that savings caused by PLFC-DAC, measured initially in consistency checks, is maintained when measuring execution time.

6 Conclusions

From this work we extract the following conclusions. First, as it was already detected for the FC algorithm, PFC may perform more consistency checks than really needed to detect dead-ends on MAX-CSP problems, although PFC presents more causes than FC for redundant check computation. Second, the lazy approach of performing the lowest amount of lookahead, just needed to keep updated $\sum_{j \in F} \min_k \{ic_{jk}\}$, is a correct strategy to remove the causes of PFC redundant check computation. And third, the lazy algorithm PLFC can still get advantage from DAC usage in the combined PLFC-DAC algorithm, which has shown a very good performance on random problems.

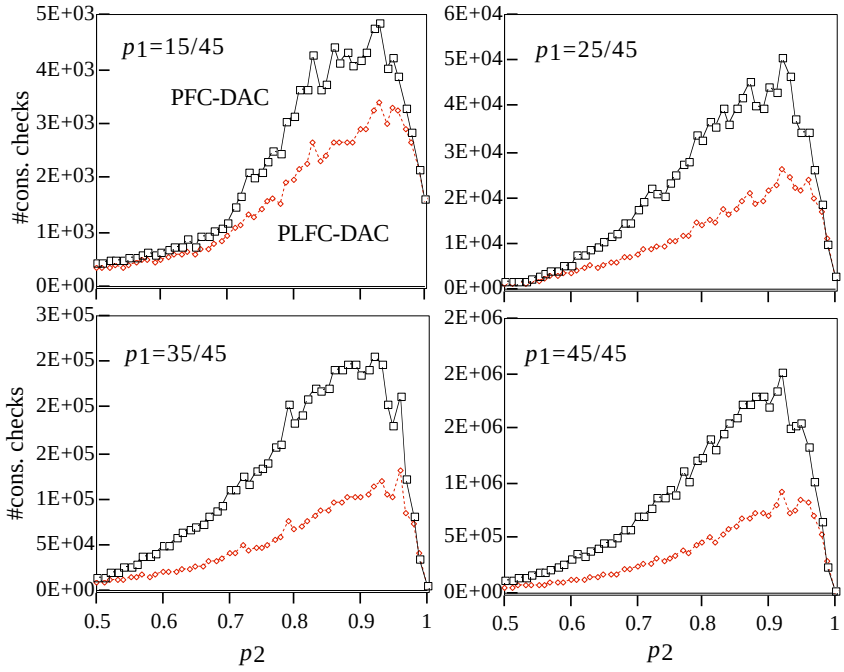

Figure 3. Consistency checks on the <10,10> random problem class.

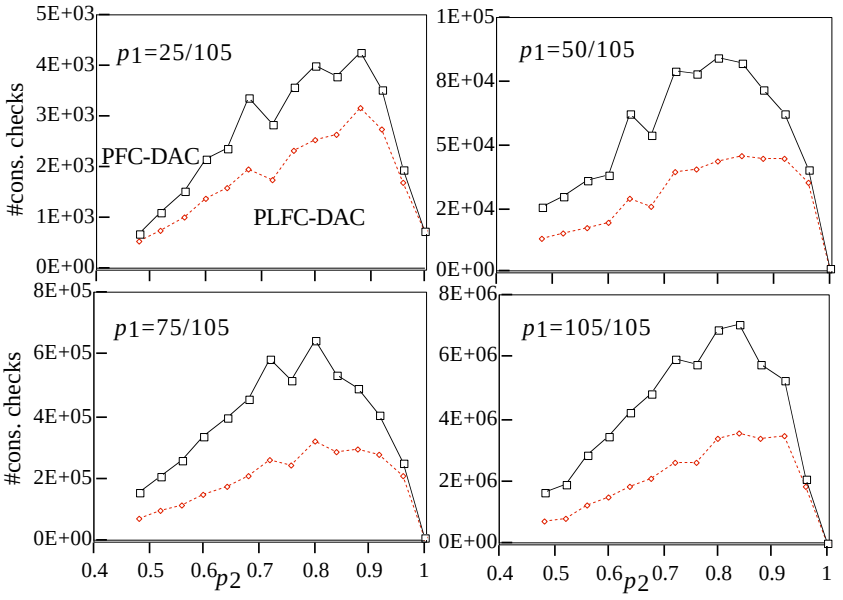

Figure 4. Consistency checks on the <15,5> random problem class.

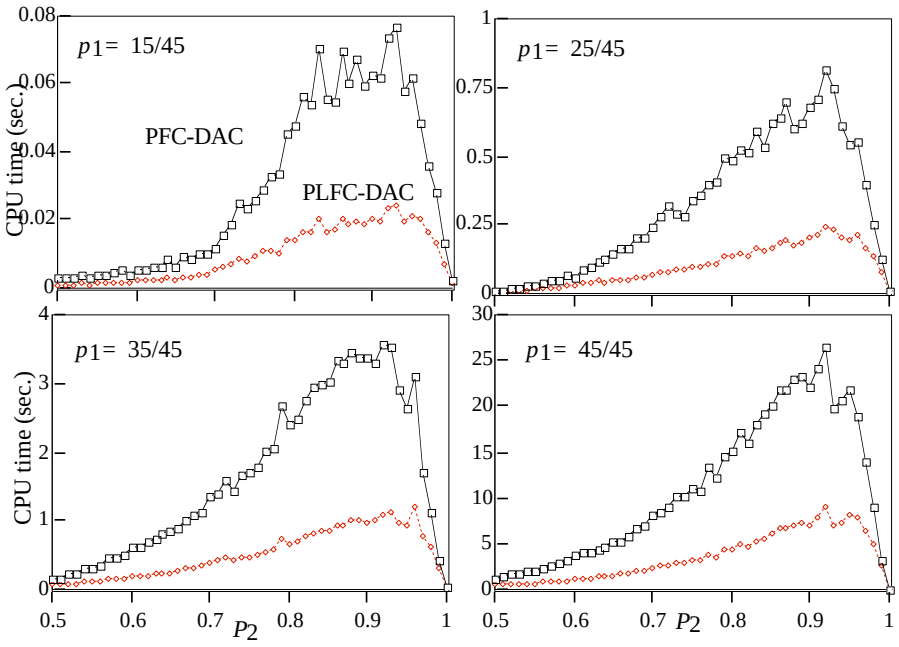

Figure 5. CPU time on the <10,10> random problem class.

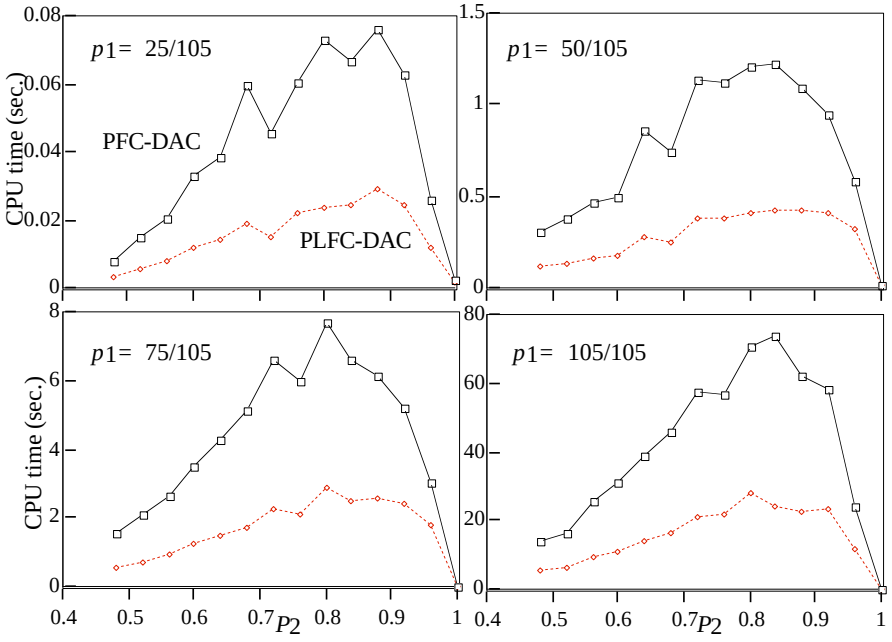

Figure 6. CPU time on the <15,5> random problem class.

References

Bacchus F. and Grove A. (1995). On the Forward Checking Algorithm, *Proceedings of CP-95*, 292-309.

Bakker R., Dikker F., Tempelman F. and Wognum P. (1993). Diagnosing and solving overdetermined constraint satisfaction problems, *Proceedings of IJCAI-93*, 276-281.

Cabon B., Verfaille G., Martinez D. and Bourret P. (1996) Using Mean Field Methods for Boosting Backtrack Search in Constraint Satisfaction Problems, *Proceedings of ECAI-96*, 165-169.

Dent M. and Mercer R. (1994). Minimal forward checking, *Proceedings of TAI-94*, 432-438.

Feldman R. and Golumbic M. C. (1990). Optimization algorithms for student scheduling via constraint satisfiability, *Computer Journal*, vol. 33, 356-364.

Fox M. (1987). *Constraint-directed Search: A Case Study on Jop-Shop Scheduling.* Morgan-Kauffman.

Freuder E. C. and Wallace R. J. (1992). Partial constraint satisfaction, *Artificial Intelligence*, 58: 21-70.

Larrosa J. and Meseguer P. (1995). Optimization-based Heuristics for Maximal Constraint Satisfaction, *Proceedings of CP-95*, 103-120.

Larrosa J. and Meseguer P. (1996). Exploiting the use of DAC in MAX-CSP. *Proceedings of CP-96*, 308-322.

Prosser P. (1994). Binary constraint satisfaction problems: some are harder than others, *Proceedings of ECAI-94*, 95-99.

Schiex T., Regin J.C., Gaspin C., and Verfaille G. (1996). Lazy Arc Consistency, *Proceedings of AAAI-96*, 216-221.

Wallace R. J. and Freuder E. C. (1993). Conjunctive width heuristics for maximal constraint satisfaction, *Proceedings of AAAI-93*, 762-778.

Wallace R. J. (1994). Directed Arc Consistency Preprocessing as a Strategy for Maximal Constraint Satisfaction, *ECAI94 Workshop on Constraint Processing*, M. Meyer editor, 69-77.

Wallace R. J. (1996). Enhancements of Branch and Bound Methods for the Maximal Constraint Satisfaction Problem, *Proceedings of AAAI-96*, 188-195.

Zweben M. and Eskey M. (1989). Constraint Satisfaction with Delayed Evaluation, *Proceedings of IJCAI-96*, 875-880.