Intelligent variable orderings and re-orderings in DAC-based solvers for WCSP *

Federico Heras and Javier Larrosa

Universitat Politecnica de Catalunya,
Barcelona, Spain
{fheras,larrosa}@lsi.upc.es,

Abstract. *Weighted constraint satisfaction problems* (WCSPs) is a well-known framework for combinatorial optimization problems with several domains of application. In the last few years, several local consistencies for WCSPs have been proposed. Their main use is to embed them into a systematic search, in order to detect and prune unfeasible values as well as to anticipate the detection of dead-ends. Some of these consistencies rely on an order among variables but nothing is known about which orders are best. Therefore, current implementations use the lexicographic order by default. In this paper we analyze the effect of heuristic orders at three levels of increasing overhead: i) compute the order prior to search and keep it fixed during the whole solving process (we call this a *static* order), ii) compute the order at every search node using current subproblem information (we call this a *dynamic* order) and iii) compute a sequence of different orders at every search node and sequentially enforce the local consistency for each one (we call this *dynamic re-ordering*). We performed experiments in three different problems: Max-SAT, Max-CSP and *warehouse location problems*. We did not find an alternative better than the rest for all the instances. However, we found that inverse degree (static order), sum of unary weights (dynamic order) and re-ordering with the sum of unary weights are good heuristics which are always better than a random order.

1 Introduction

Constraint satisfaction problems (CSP) provide a powerful framework to express a large spectrum of problems arising in computer science. In the last years, the CSP framework has been augmented with so-called *soft constraints* with which it is possible to express preferences among solutions [13, 2]. The *Weighted constraint satisfaction problem* (WCSP) is a well known soft-constraint framework. It provides a very general model with several applications in domains such as *resource allocation* [3], *combinatorial auctions* [12], *bioinformatics* and *probabilistic reasoning* [11].

Local consistency properties play a fundamental role in CSP and WCSP solving. Each property comes with a polynomial enforcing algorithm that, when applied to an arbitrary instance, transforms it into an equivalent one that satisfies the property. The resulting problem is more explicit and presumably simpler. From a practical point of

* This research is supported by the MEC through project TIC-2002-04470-C03.

view, an important class of local consistencies are those in which the enforcing algorithm prunes unfeasible values. These are often called *filtering algorithms* because they *filter out* redundant values. The main use of a filtering algorithm is to embed it into systematic search: Every search node is a subproblem and the filtering algorithm is executed at every visited node. Thus, such an algorithm *maintains* the corresponding local consistency at every subproblem [1, 8]. This approach has proven very efficient and is currently supported by most state-of-the-art solvers. A desirable property of filtering algorithms is *incrementality*. Namely, if consistency has been enforced at a given node, its child do not have to enforce it from scratch, but only consider the changes of their associated subproblem with respect to their parents'. Incrementality decreases the overhead of continuously enforcing the local consistency.

Two well-known CSP local consistencies are *arc consistency* (AC) and *directional arc consistency* (DAC) [7]. In the WCSP framework they can be combined producing an even stronger property called *full directional arc consistency* (FDAC*) [4, 9]. TOOLBAR [1] is a freely available WCSP solver. It performs a systematic depth-first search on the tree of assignments. It offers the possibility of maintaining different forms of local consistency during search using incremental algorithms. Extensive experiments of a large variety of domains show that FDAC* is usually a competitive option. In the sequel, we take FDAC* as the local consistency of reference.

FDAC* relies on a given variable order but nothing is known about how it affects the performance of the solving algorithm. Thus, the current implementation of TOOLBAR just takes the lexicographical order by default. In this paper we analyze the effect that the orderings may have in TOOLBAR performance. We observe that TOOLBAR is quite sensitive, since its performance can be affected by orders of magnitude. Then we study the effect of different orders at three different levels:

- *Static orders* are computed prior to search and kept fixed during the solving process. They have the advantage of being free of any additional overhead because they do not interfere with the incrementality of the filtering algorithm. Their disadvantage is that they are based on information from the original problem. However, the original problem is modified along the tree paths as variables are assigned. Therefore, the information may lose accuracy.
- *Dynamic orders* are computed at every search node using information on the current subproblem. They have the advantage of being much more accurate. One disadvantage is their overhead. Most importantly, the incrementality of the filtering algorithm is lost.
- We also explore *re-orderings*. Namely, FDAC* is enforced under a sequence of different orders at every search node.

We report experiments in three different domains: *warehouse location problems*, *Max-CSP* and *Max-SAT*. In our experiments we did not find a clear winner. However, we found that enforcing FDAC* subject to the inverse degree (static order), sum of unary weights (dynamic order) and re-ordering with the sum of unary weights usually improve the performance of systematic search algorithms.

[1] http://mulcyber.toulouse.inra.fr/projects/toolbar/

The structure of this paper is as follows: In Section 2 we provide the necessary background. In Section 3 we introduce the benchmark used in the experiments. In Section 4 we report and discuss the results of our experiments. In Section 5 we relate our work to the very recent EDAC* property. Finally, Section 6 gives some conclusions and points out some future work.

2 Preliminaries

2.1 WCSP

A *weighted constraint satisfaction problem* (WCSP) [10] is a tuple $P = (\top, \mathcal{X}, \mathcal{D}, \mathcal{C})$. $\mathcal{X} = \{1, \ldots, n\}$ is a set of *variables*. Each variable $i \in \mathcal{X}$ has a finite domain $D_i \in \mathcal{D}$ of values that can be assigned to it. (i, a) denotes the assignment of value $a \in D_i$ to variable i. A tuple t is an assignment to a set of variables. Actually, t is an ordered set of values assigned to the ordered set of variables $var(t) \subseteq \mathcal{X}$ (namely, the k-th element of t is the value assigned to the k-th element of $var(t)$). For a subset B of $var(t)$, the projection of t over B is noted $t \downarrow_B$. $\mathcal{C}$ is a set of *cost functions* (also called soft constraints). The set of variables affected by a C is called its *scope* and noted $var(C)$. The scope size is called *arity*. $\top$ is a natural number indicating the maximum cost. Formally, a cost function is $C : \prod_{i \in var(C)} D_i \rightarrow [0, \ldots, \top]$. When a constraint C assigns cost $\top$ to a tuple t, it means that C forbids t, otherwise t is permitted by C with the corresponding cost. The sum of costs $\oplus$ is defined as $a \oplus b = \min\{\top, a + b\}$. The *cost* of a tuple t, noted $\mathcal{V}(t)$, is the sum over all applicable costs,

$$\mathcal{V}(t) = \sum_{C \in \mathcal{C},\ var(C) \subseteq var(t)} C(t \downarrow_{var(C)})$$

Tuple t is *consistent* if $\mathcal{V}(t) < \top$. It is *globally consistent* if it can be extended to a complete consistent assignment. The usual task of interest is to *find a complete consistent assignment with minimum cost*, which is NP-hard. Observe that WCSP with $\top = 1$ reduces to classical CSP.

The arity of a problem is the largest arity among its constraints. A binary problem is a problem of arity 2. Given a WCSP instance, its *constraint graph* has one vertex associated to each variable and one edge associated to each pair of variables appearing in the scope of some cost function.

Example 1. Figure 1.a shows a WCSP. It has three variables $\mathcal{X} = \{x, y, z\}$ with values a, b. There are 2 binary constraints C_{xz}, C_{yz} and two non trivial unary constraints C_y and C_z. Unary costs are depicted inside their domain value. Binary costs are depicted as labelled edges connecting the corresponding pair of values (default cost of 1). Zero costs are not shown. In this problem all complete assignments are optimal, with cost 2.

Two WCSP defined over the same variables are said to be *equivalent* if they define the same cost distribution on complete assignments. For instance, The six problems in Figure 1 are equivalent.

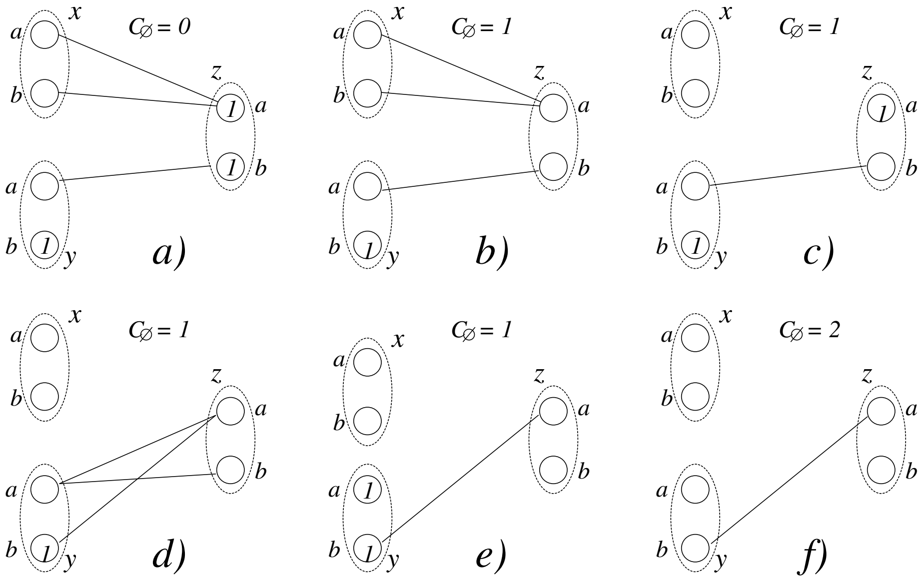

Fig. 1. Six equivalent WCSPs ($\top = 4$).

2.2 FDAC*

We review the *full directional arc consistency* (FDAC*) local consistency [4, 9]. We assume binary problems where every constraint has a different scope. In the following, C_{ij} is the binary constraint with scope $\{i, j\}$ (namely, $C_{ij} : D_i \times D_j \rightarrow [0, \ldots, \top]$). C_i is the unary constraint with scope $\{i\}$ (namely, $C_i : D_i \rightarrow [0, \ldots, \top]$). We assume the existence of a unary constraint C_i for every variable, and a *zero*-arity constraint, noted $C_\varnothing$ (if no such constraint is defined, we can always define *dummy* ones $C_i(a) = 0, \forall a \in D_i$ or $C_\varnothing = 0$). Finally, and most importantly in this paper, we assume that the set of variables $\mathcal{X}$ is totally ordered by $>$.

Definition 1 *Let $P = (\top, \mathcal{X}, \mathcal{D}, \mathcal{C})$ be a binary WCSP. P is FDAC* if it satisfies the following conditions:*

- *For every variable i, every value $a \in D_i$ satisfies that $C_\emptyset \oplus C_i(a) < \top$.*
- *For every variable i there is at least one value $a \in D_i$ such that $C_i(a) = 0$. Value a is called a support.*
- *Every cost function C_{ij} (with $i < j$) satisfies that:*
  - *Every value $b \in D_j$ has at least one value $a \in D_i$ such that $C_{ij}(a, b) = 0$. Value a is called a* support *of b.*
  - *Every value $a \in D_i$ has at least one value $b \in D_j$ such that $C_{ij}(a, b) \oplus C_j(b) = 0$. Value b is called a* full support *of a.*

It has been shown in [4, 10] that FDAC* can be enforced by applying basic transformations until the corresponding property is satisfied. They require a subtraction of costs. Let $a, b \in [0, \ldots, \top]$, be two costs such that $a \geq b$. $a \ominus b$ is the *subtraction* of b from a, defined as,

$$a \ominus b = \begin{cases} a - b & : \quad a \neq \top \\ \top & : \quad a = \top \end{cases}$$

Example 2. Consider the problem in Figure 1.a under the $x < y < z$ order. It is not FDAC* since z has no support. The support is enforced by *projecting* C_i costs to $C_\emptyset$. The projection consists of subtracting 1 from $C_i(\cdot)$ and adding 1 to $C_\emptyset$. The resulting problem is depicted in Figure 1.b. It still is not FDAC* because (z, a) has no support on x. The support is enforced by *projecting* C_{xz} costs to C_z. The projection consists on subtracting 1 from $C_{xz}(\cdot, a)$ and adding it to $C_z(a)$. The resulting problem is depicted in Figure 1.c. It is not FDAC* either since (y, a) has no full support on z. The full support is enforced in two steps. First $C_z(a)$ costs are *extended* to C_{yz}. The extension consists on subtracting 1 from $C_z(a)$ and adding it to $C_{yz}(\cdot, a)$. The resulting problem is depicted in Figure 1.d. Next, $C_{yz}(a, \cdot)$ is projected to $C_y(a)$. The resulting problem is depicted in 1.e. Finally, C_y is projected to $C_\emptyset$. The resulting problem, in Figure 1.f, is FDAC*.

In [9] an incremental enforcing algorithm for FDAC* with time complexity $O(end^3)$ was introduced (e,n and d are the number of constraints, variables and domain values, respectively). Note that a problem may satisfy the FDAC* property with respect to one variable order and not with respect to another. Therefore, if the order among variables changes, the filtering algorithm must be re-executed. Note as well that enforcing FDAC* is, in a way, assymetric with respect the order of variables. Consider an intermediate variable i. It requires a stronger property with respect to the larger variables than with respect to the smaller variables. The smallest variable 1 is the most demanding one (it requires full supports in every other variable). The largest variable n is the least demanding one (it does not require any full support).

2.3 Maintaining Local Consistency

TOOLBAR is a state-of-the-art WCSP solver. It follows a backtracking scheme which searches for the optimal solution by extending partial assignments. It searches depth-first the tree of assignments, where internal nodes stand for partial assignments and leaves stand for complete ones. At an internal node, the algorithm selects an unassigned variable (using some heuristic criterion) and iterates over its domain values. For each value, it extends the current assignment and applies recursively the same process.

The optimization nature of the WCSP problems is exploited by TOOLBAR as follows: On the one hand, the algorithm maintains the cost of the best solution found so far. Such a value is an upper bound (UB) for the optimal solution. On the other hand, at each node, the algorithm computes a lower bound (LB) of the best solution that can be possibly attained in the current line of search. Clearly, when the lower bound is greater than or equal to the upper bound the algorithm can backtrack, since the current line of search cannot improve over the best current solution.

For TOOLBAR, each visited node is a WCSP subproblem. Thus, the lower bound is the $C_\varnothing$ of the current subproblem and the upper bound is $\top$. Each time that a better solution is found, the value of $\top$ is updated. When a new assignment (i, a) is made, the current set of constraints $\mathcal{C}$ is updated by replacing each constraint C by the result of assigning it with (i, a). This process is usually called LookAhead. Assigning a variable i of a constraint C has the effect of removing i from its scope. As a result, several constraints with the same scope may appear. TOOLBAR combines them into one single constraint whose cost is the sum of all costs. Especially important is the combination of constraints with empty scope, since they can be added to the current lower bound. Another important feature of the LookAhead procedure is that it returns two streams Q and R. Q contains all variables whose domains of values have been pruned by the LookAhead. R contains all the variables having some value whose unary cost have been increased from 0. The streams are passed to the filtering algorithm. The filtering algorithm only needs to care about the variables connected with the variables in Q and R. This is how the incrementality of the filtering algorithm is obtained. Note that this is true only if the order under which FDAC* is enforced is the same in the parent and the child node.

The filtering algorithm considers the binary WCSP obtained by taking from the current subproblem all constraints of arity less than or equal to 2. If the consistency maintenance produces an empty domain, the algorithm backtracks. Otherwise, search continues recursively. In the following, MFDAC denotes the result of executing TOOLBAR maintaining FDAC*. An extensive experimentation on a wide range of test problems shows that MFDAC generally is the best option among a set of different local consistencies [9, 6].

In the current implementation, TOOLBAR maintains FDAC* under the lexicographical order of the variables. As we will show in Section 4, this arbitrary decision may affect the performance of the algorithm. Besides, it uses the *dom/deg* heuristics for branching which selects the variable with the smallest ratio of domain size divided by future degree. For value selection it considers values in increasing order of unary cost C_i.

3 The benchmark

All the experiments reported in this paper were conducted with a Pentium IV computer with 1 Gb of RAM and a speed of 3GHz. We experimented with three problems easily modeled as WCSP.

3.1 Max-SAT

Given a CNF formula, **Max-SAT** is the problem of finding a complete assignment with a maximum number of satisfied clauses. We generated and solved 80-variable 2-SAT and 40-variable 3-SAT instances varying the number of clauses using *Cnfgen*[2], a random k-SAT generator. Note that this generator prevents duplicate or opposite literals in clauses but not duplicate clauses. For each number of clauses we generated 10 instances and we always report mean values.

[2] A. van Gelder ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/UCSC/instances

3.2 Max-CSP

Given an over-constrained CSP, **Max-CSP** is the problem of finding a complete assignment with a maximum number of satisfied constraints. We generated 6 classes of binary random problems with different connectivity and tightness as proposed in [9]. The six classes are denoted: sparse loose, sparse tight, dense loose, dense tight, complete loose, and complete tight. Domain size was always set to 10. Samples have 20 instances and we always report mean values.

3.3 Warehouse Location

In the **uncapacitated warehouse location problem** (UWLP) a company considers opening warehouses at some candidate locations in order to supply its existing stores. The objective is to determine which warehouses to open, and which of these warehouses should supply the various stores, such that the sum of the maintenance and supply costs is minimized. Each store must be supplied by exactly one open warehouse. We simply model the problem by l boolean variables for the candidate locations, s integer variables for the stores with domain size set to l, $l + s$ soft unary constraints for the costs, and $l \times s$ hard binary constraints connecting stores and warehouses. The resulting constraint graph is a bipartite graph with locations on one side and stores on the other (Figure 2 shows the constraint graph of a $l = 2$,$s = 4$ instance). We took and solved some problem instances from the standard OR Library benchmarks for UWLP: *cap71-74* with 16 warehouses and 50 stores, *cap101-104* with 25 warehouses and 50 stores and *cap131-134* with 50 warehouses and 50 stores.

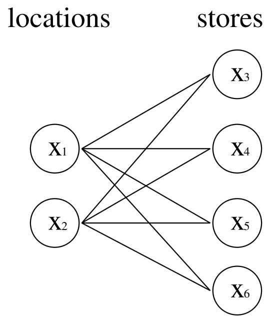

Fig. 2. Constraint graph of a UWLP with 2 warehouses and 4 stores.

4 Intelligent orderings for FDAC* maintenance

4.1 Static variable orderings

In this Section we analyze the effect in the performance of heuristically establishing a static ordering for FDAC* maintenance during search. The idea is to decide a variable ordering prior to search and keep it fixed along the whole solving process. A nice property of static orderings is that they are compatible with the incrementality of the filtering algorithms, so their use is free of any overhead whatsoever. Consequently, if we maintain FDAC* under a static variable ordering, the worst-case complexity per node is $O(end^3)$. However, incrementality makes the worst case unlikely. We consider the following alternatives (ties are always broken lexicographically):

– **Degree (dg)**. Given a WCSP instance, the degree of variable i is the number of adjacent nodes of i in its constraint graph. Under the degree ordering, we define $i < j$ if the degree of i is smaller than the degree of j.
– The **inverse degree (inv-dg)** is obtained by reversing the degree ordering.
– The **forward degree (fdg)** ordering is computed as follows: The smallest variable i is the variable with the smallest degree. The next smallest variable is obtained by removing i and all its adjacent edges from the constraint graph and repeating the same process. The recursive application of the process produces the ordering.
– The **inverse forward degree (inv-fdg)** is the order obtained by reversing *fdg*.

Problem	size	rnd	dg	inv-dg	fdg	dyn-inv-dg	suc
cap71	16×50	0 (0.01)	2 (0.11)	0 (0)	1 (0.09)	0 (0)	0 (0)
cap72	16×50	1 (0.1)	6 (0.38)	0 (0)	3 (0.24)	0 (0)	0 (0.01)
cap73	16×50	2 (0.18)	5 (0.35)	0 (0.01)	3 (0.28)	0 (0.01)	0 (0.02)
cap74	16×50	0 (0.05)	3 (0.2)	0 (0.01)	1 (0.13)	0 (0.01)	0 (0.01)
cap101	25×50	5 (0.71)	257 (21.68)	0 (0.02)	112 (15.32)	0 (0.04)	0 (0.03)
cap102	25×50	102 (13.04)	470 (46.49)	0 (0.08)	245(31.43)	0 (0.13)	0 (0.16)
cap103	25×50	20 (3.29)	239 (30.09)	2 (0.24)	138 (22.44)	2 (0.39)	1 (0.44)
cap104	25×50	8 (1.82)	62 (9.62)	1 (0.17)	34 (7.62)	1 (0.24)	1 (0.23)
cap131	50×50	-	130 (45.1)	130 (44.2)	-	130 (87.26)	114 (87.6)
cap132	50×50	-	237 (96.76)	237 (97.51)	-	237 (163.1)	240 (179.01)
cap133	50×50	-	253 (118.93)	253 (117.59)	-	253 (190.7)	248 (203.06)
cap134	50×50	-	47 (27.67)	47 (27.38)	457 (421.12)	47 (41.52)	46 (50.63)

Table 1. Visited nodes in thousands and time in seconds (within parenthesis) to prove optimality for uncapacitated warehouse location problems with different orderings. A "-" means the instance was not solved in less than 1000 seconds.

Figure 3 depicts the comparison of all the alternatives in Max-SAT and Max-CSP problems. We measure the different performances as the number of visited nodes (since static orders do not cause any overhead we do not report cpu time in this experiment). For comparison purposes, we also include the performance of enforcing FDAC* with a random order (noted *rnd*). We do not experiment with complete graph Max-CSP problems because heuristic orders based on degree are obviously useless in complete graphs.

In Max-2-SAT, we see that *inv-dg* and *fdg* are better than *rnd* and their reverses. Clearly, the best option is *inv-dg*, which visits almost a half of the number of nodes than *rnd*. In Max-3-SAT we observe that none of the heuristics have a significant effect. In Max-CSP we observe that *inv-dg* and *dg* are the best and worst options, respectively. *inv-dg* can visit up to three times less nodes than *rnd*. Surprisingly, the relative performance of *fdg* versus *inv-fdg* changes on loose problems. Thus *inv-dg* appears as the only

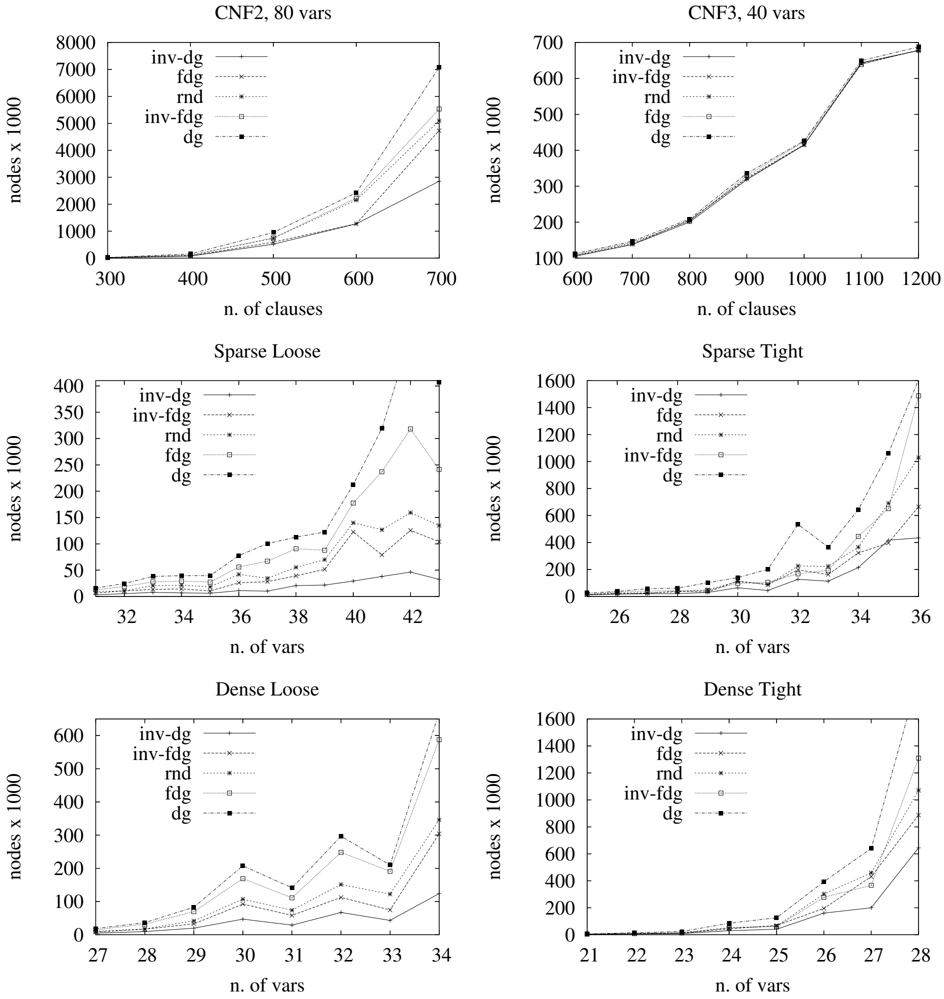

Fig. 3. Visited nodes for Max-SAT and Max-CSP problems with different static orders.

heuristic always giving good results (see fig. 3). Given the particular structure of UWLP and the fact that in our instances the number of locations is smaller than or equal to the number of stores, it is easy to reproduce the effect of static orders in this benchmark (in our format, variables associated to locations are lexicographically smaller than variables associated to stores). For instance, *inv-dg* and *inv-fdg* both yield the same order (hence we omit the later) which is the lexicographic order. *dg* makes stores to be smaller than locations (except in the *cap*131-134 where locations are smaller than stores as *inv-dg*). Both *rnd* and *fdg* produce an order in which locations and stores alternate. Results on UWLP are reported in table 1. The first and second columns indicate the identifier and the size of the instances. From the third to the sixth column we have the number of visited nodes (in thousands) and the cpu time (within parenthesis). We can see that *inv-dg*, as well as any other heuristic that makes locations smaller than stores, produces good results. Either making stores smaller or alternating stores and locations yield a bad performance of the solving algorithm.

4.2 Dynamic variable orderings

In this Section we analyze the effect in the performance of heuristically establishing a dynamic variable ordering. The idea is to decide the variable ordering at every search node, in accordance with the current subproblem. The advantage of dynamic orderings is that they do not lose accuracy along the search tree branches, where subproblems differ more and more from the original problem as new variables are assigned. The disadvantage of dynamic orderings is their overhead. On the one hand the order has to be computed every time. On the other hand, and most importantly, incrementality is lost. If the order of variables is arbitrarily changed in one node with respect to its parent's, FDAC* must be computed from scratch. Consequently, the cost per node of maintaining FDAC* under a dynamic variable ordering is $O(end^3)$ assuming that computing the order is less costly than that. Besides, the lack of incrementality makes the worst-case more likely to occur.

It seems reasonable to expect that those orders that were effective statically, should be even more effective dynamically. At least, this should be true when considering the number of visited nodes as performance measure. However, cpu time is a better comparison measure in this case because it also takes overhead into account. We consider the following alternatives:

- The **Dynamic inverse degree (dyn-inv-dg)** is like the inv-dg, but updated at every node.
- **Sum of unary costs (suc)**. Under this ordering, we define $i < j$ if $\sum_{a \in D_i} C_i(a) < \sum_{a \in D_j} C_j(a)$.
- **Dynamic domain size (dyn-dom)**. Under this ordering, we define $i < j$ if $|D_i| < |D_j|$, using the current domain size at each node.

Figure 4 reports the results of this experiments in terms of visited nodes. For comparison, we include *inv-dg*, the best static order, in all the plots. We do not include the results of *dyn-dom*, because it never improved over *inv-dg*. In Max-2-SAT we observe that, as expected, *dyn-inv-dg* clearly improves upon *inv-dg*. However, the best option

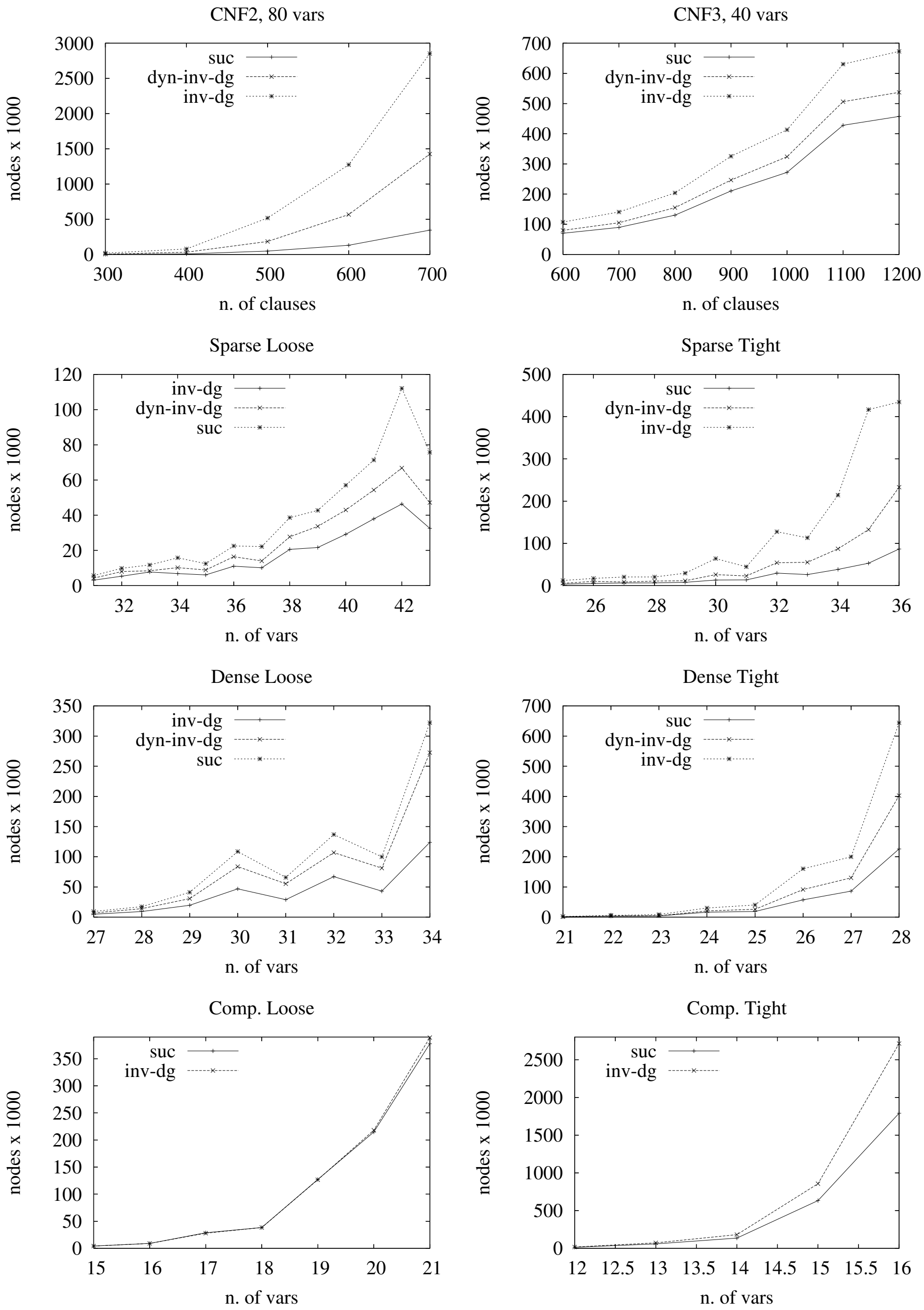

Fig. 4. Visited nodes in Max-SAT and Max-CSP problems with dynamic orderings.

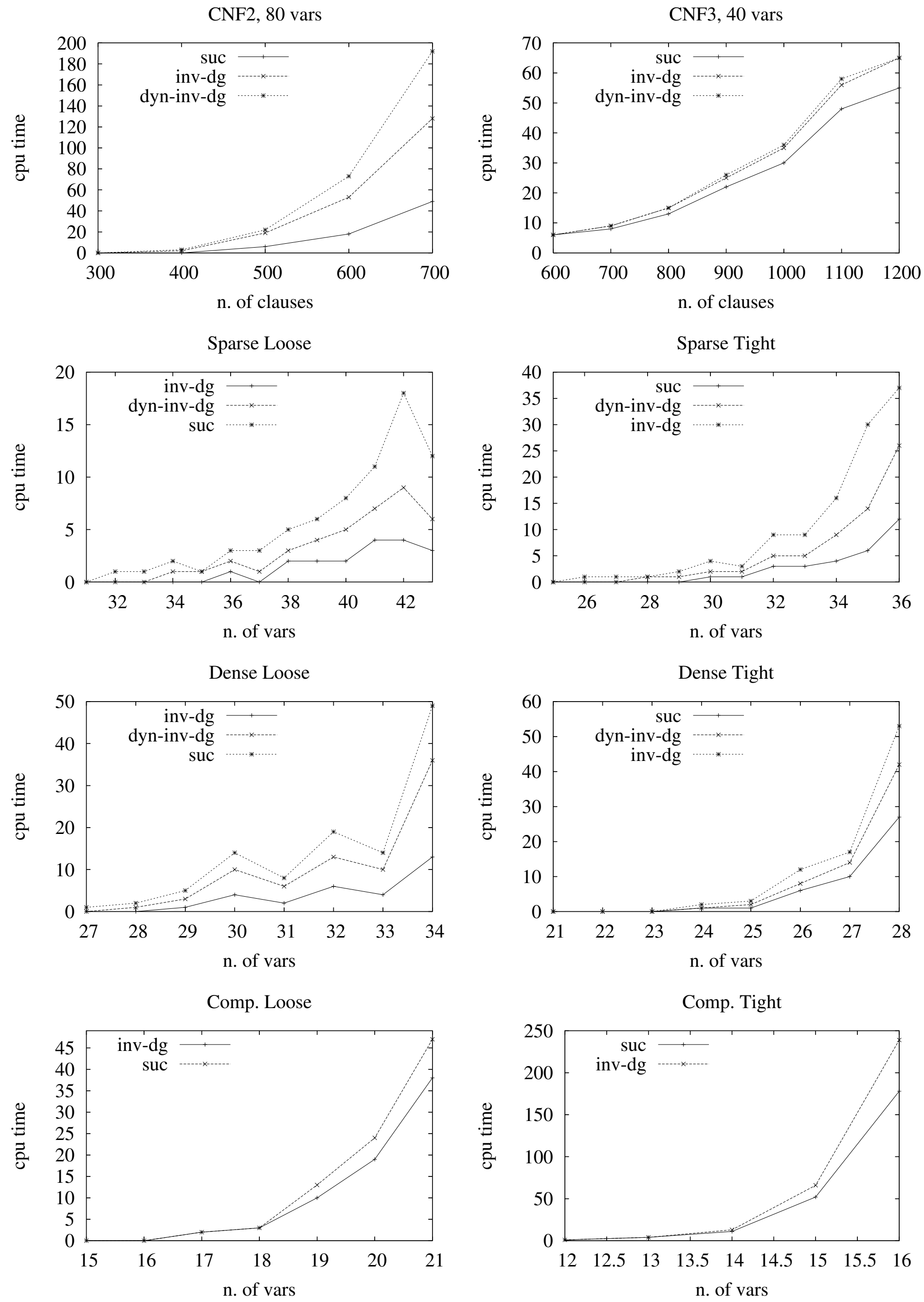

Fig. 5. CPU time in seconds in Max-SAT and Max-CSP problems with dynamic orderings.

is *suc* which visits 6 and 3 times less nodes than *inv-dg* and *dyn-inv-dg*, respectively. In Max-3-SAT we observe similar results, although gains are not so important. The same pattern is observed in tight Max-CSP problems (note that *dyn-inv-dg* is useless in complete graphs, hence omitted). As with static orders, sparse and dense loose problems present a different pattern. Surprisingly, the static *inv-dg* is the best order here. In dense-lose problems *inv-dg* visits up to 3 times less nodes than *suc*. Both *inv-dg* and *suc* obtain similar results for complete loose problems. Regarding cpu time (Figure 5) we usually observe the same relation of dominance, as per visited nodes (only in Max-2-SAT there is a swap between *inv-dg* and *dyn-inv-dg*). However, the higher overhead of dynamic orders makes performance ratios slightly more beneficial to static orders. Dynamic orderings of UWLP are reported in columns 7 and 8 of table 1. We observe that both *dyn-inv-dg* and *suc* obtain results similar to those of *inv-dg*. The reason is that the two of them tend to place locations before stores: in the *dyn-inv-dg* case it is clearly true in nodes near the root and very likely at deeper nodes (TOOLBAR will likely assign first location variables because they have smaller domains); in the *suc* case it is also likely because locations have much smaller domains and their sum of unary costs should be also small. Comparing static with dynamic orders, note that cpu times are little worsened due to the overhead.

4.3 Dynamic re-orderings

In this Subsection we analyze the use of more than one order at each node. The idea is to consider a sequence of orders and enforce FDAC* with respect to them one after the other. This approach raises two issues: the orders to consider and the length of the sequences. Regarding the first issue, we consider the following alternatives:

- **Sequence of *suc* (ssuc)**. In the previous Subsection we observed that *suc* is an effective dynamic order. Note that the filtering algorithm changes the problem, so if we recompute *suc* after enforcing FDAC*, we will probably obtain a different order. Therefore, we can enforce FDAC* again.
- **Sequence of alternating orders (alt)**. First we establish a static order *d*. The sequence of orders consists in alternating *d* and its reverse *inv-d*. The main potential advantage of this approach is that there is an incremental implementation, achieved by using an additional stream $R2$. As usual, R contains the set of variables, whose smaller neighbors need to be revised for full support (according to *d*). $R2$ contains equivalent information but under the reversed order *inv-d*. The current node receives stream R from its parent, which contains variables that need to be revised due to the enforcement of FDAC* under *inv-d* at the parent. Thus, the parent must pass R to each children and $R2$ must be reset to $\varnothing$. At the child node, the `LookAhead` procedure must add variables to both R and $R2$.When FDAC* is enforced under *d*, it pops variables from R and pushes variables to both R and $R2$. When FDAC* is enforced under *inv-d*, it pops variables from $R2$ and pushes variables to both R and $R2$. We experimented with different static orders and observed that it is the alternation and not the actual order *d* what makes this approach efficient. However, slightly best results were obtained with the inverse degree order.

Regarding the length of sequences we consider the following alternatives (in decreasing overhead):

- Enforce FDAC* with respect to the following order of the sequence as long as the filtering algorithm increases the lower bound or prunes domain values and there is no empty domain. We denote this criteria *prune*. The cost per node of this approach is $O(end^3 \max\{nd, \top\})$, since we may need to re-enforce FDAC* as many times as the number of possible lower bound increments and value prunings.
- Enforce FDAC* with respect to the following order of the sequence as long as the filtering algorithm increases the lower bound and there is no empty domain. We denote this criteria *lb*. The cost per node of this approach is $O(\top end^3)$, since we may need to re-enforce FDAC* as many times as the number of possible lower bound increments.
- Enforce FDAC* with respect to the following order of the sequence as long as the filtering algorithm increases the lower bound and there is no empty domain, but *never more than k times*, where k is a constant. We refer to this criteria with the value of k. The cost per node of this approach is $O(kend^3)$.

Combining the two ways to construct the sequence (*ssuc* and *alt*) and the three ways to define its lenght (*prune*, *lb* and k), we have 6 different alternatives. Each alternative is noted X-Y, where X is either *ssuc* or *alt* and Y is *prune*, *lb* or a constant value k. In our experiments we only show results with $k = 2$ because it provided the best performance.

Figures 6 and 7 report the performance of dynamic re-orderings based on *ssuc*. For comparison purposes, we include in the plot the best previous results in terms of cpu time. Regarding visited nodes, *ssuc-prune* is always the best alternative. While for tight Max-CSP problems the reduction of visited nodes is about 2 to 4 times, for loose Max-CSP differences are little. Better results with respect to dynamic orders are obtained in tight Max-CSP and Max-3-SAT problems. In Max-CSP loose problems the static *inv-dg* remains a very competitive order, improving over some re-orderings. Regarding cpu time, the high overhead of re-orderings makes them only useful in Max-3-SAT and tight Max-CSP problems, where *ssuc-prune*, the most costly one, is the best option. Note however, that improvements with respect to best dynamic orders is not very significant. Only in sparse tight Max-CSP problems the reduction reaches half of the time wrt *suc*.

Figures 8 and 9 report the performance of dynamic re-orderings based on *alt*. Regarding the visited nodes *alt-prune* produces better result than the other *alt* versions. Furthermore, it is the first time that we observe an important improvement in loose Max-CSP problems. However, in time it is always better to choose a simpler version such as *alt-2* ordering except for Max-3-SAT. In Max-2-SAT and tight Max-CSP *suc* and *ssuc-prune* respectively are the best option. In Max-3-SAT and complete loose Max-CSP, *alt-2* is little better than the static *inv-dg*, and for sparse and dense loose Max-CSP *alt-2* times are very close to *inv-dg* times.

Regarding UWLP problems, we observed that re-orderings did not bring more than minor gains in terms of visited nodes which were not cost effective in terms of cpu time (we omit these results).

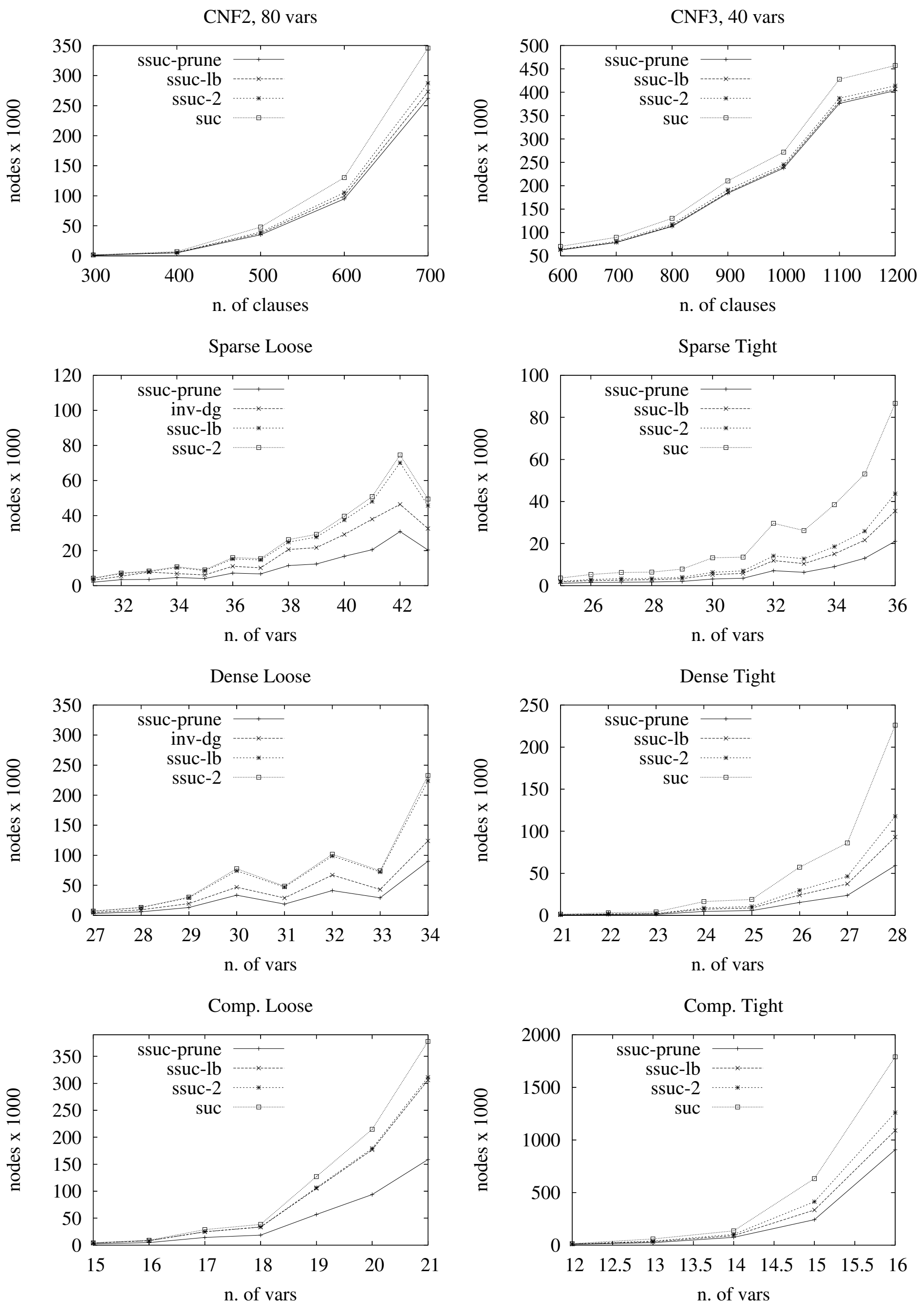

Fig. 6. Visited nodes in random Max-SAT and Max-CSP problems with different *ssuc* re-orderings.

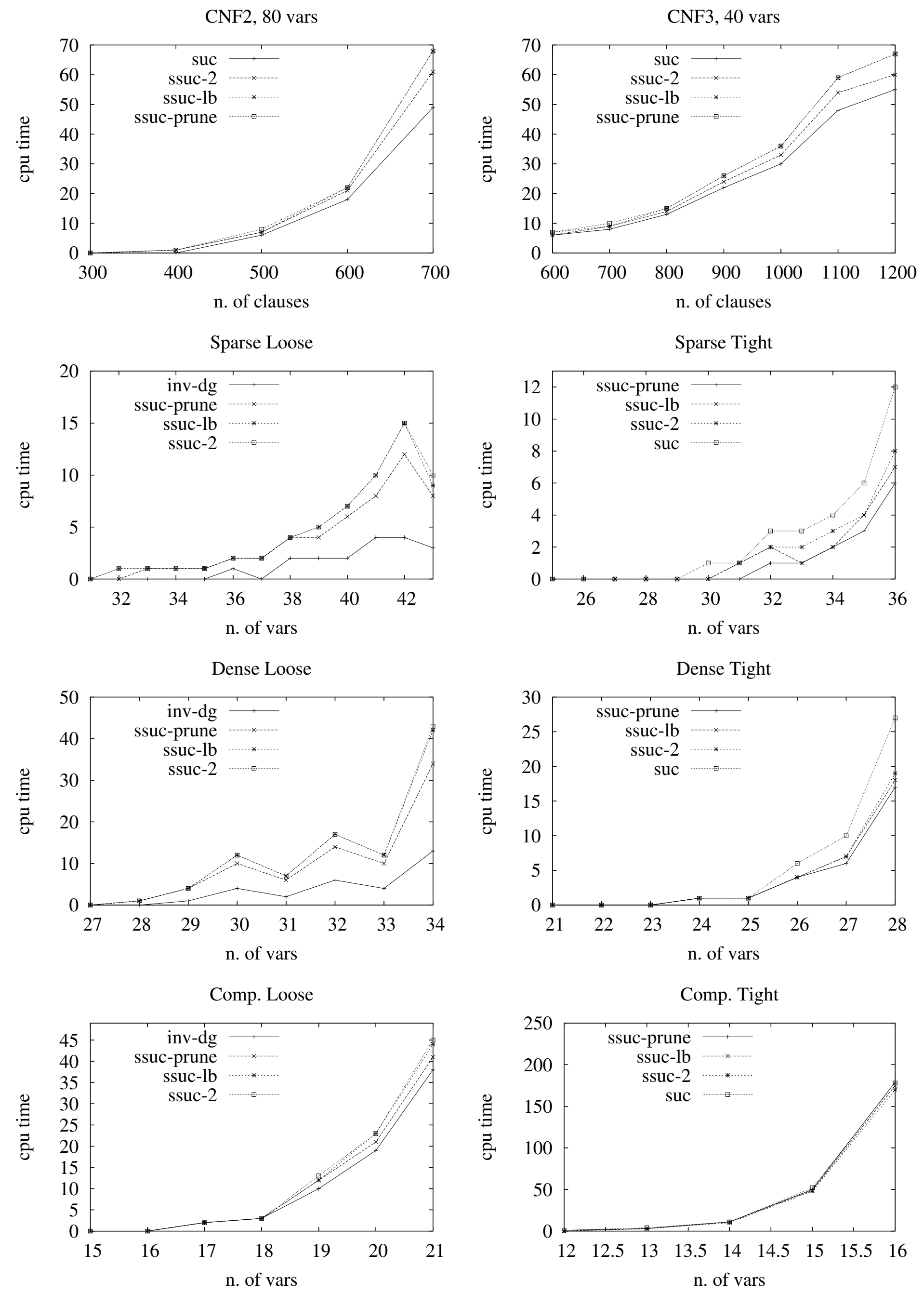

Fig. 7. CPU time in seconds in random Max-SAT and Max-CSP problems with different *ssuc* re-orderings.

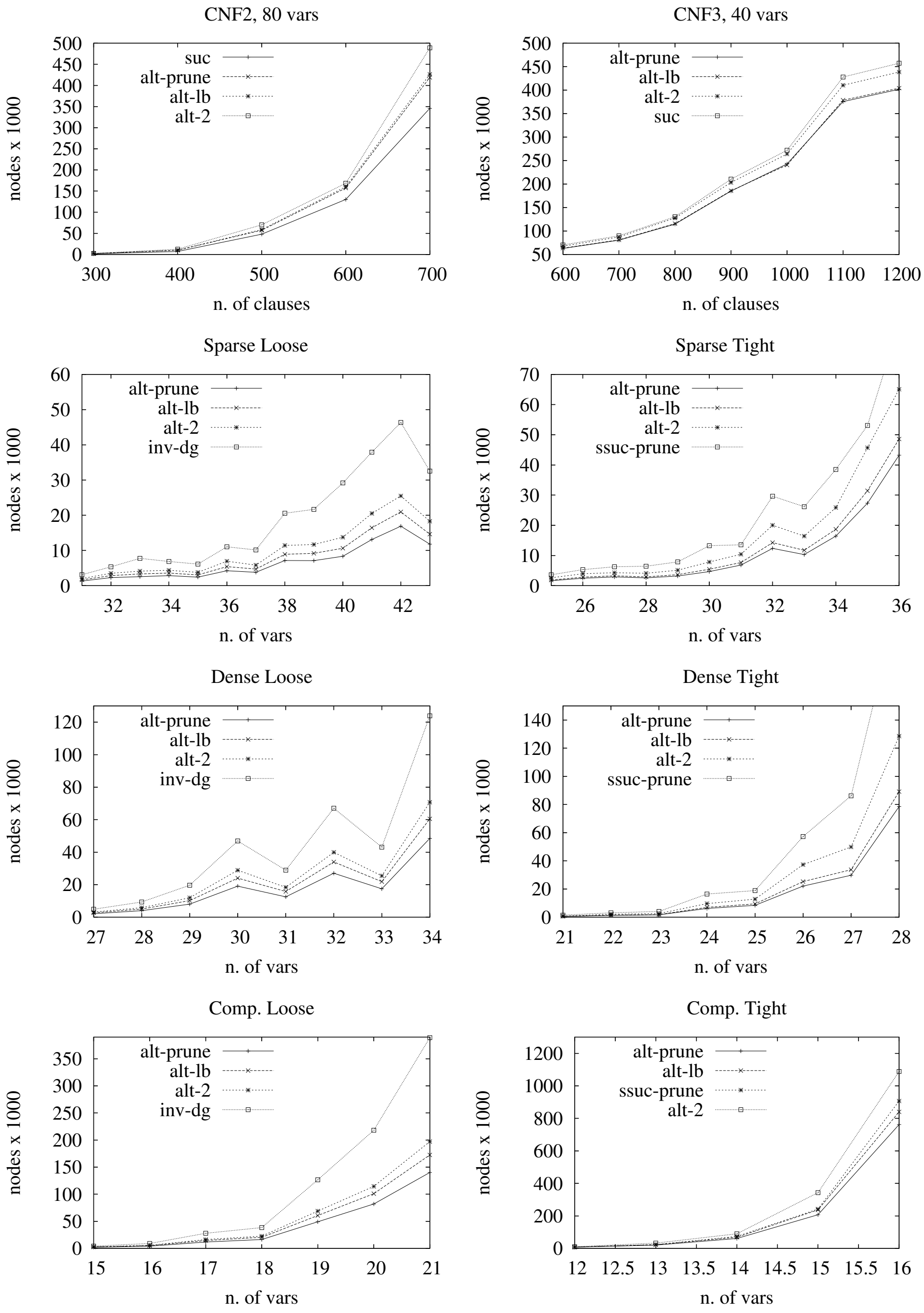

Fig. 8. Visited nodes in Max-SAT and Max-CSP problems with different *alt* re-orderings.

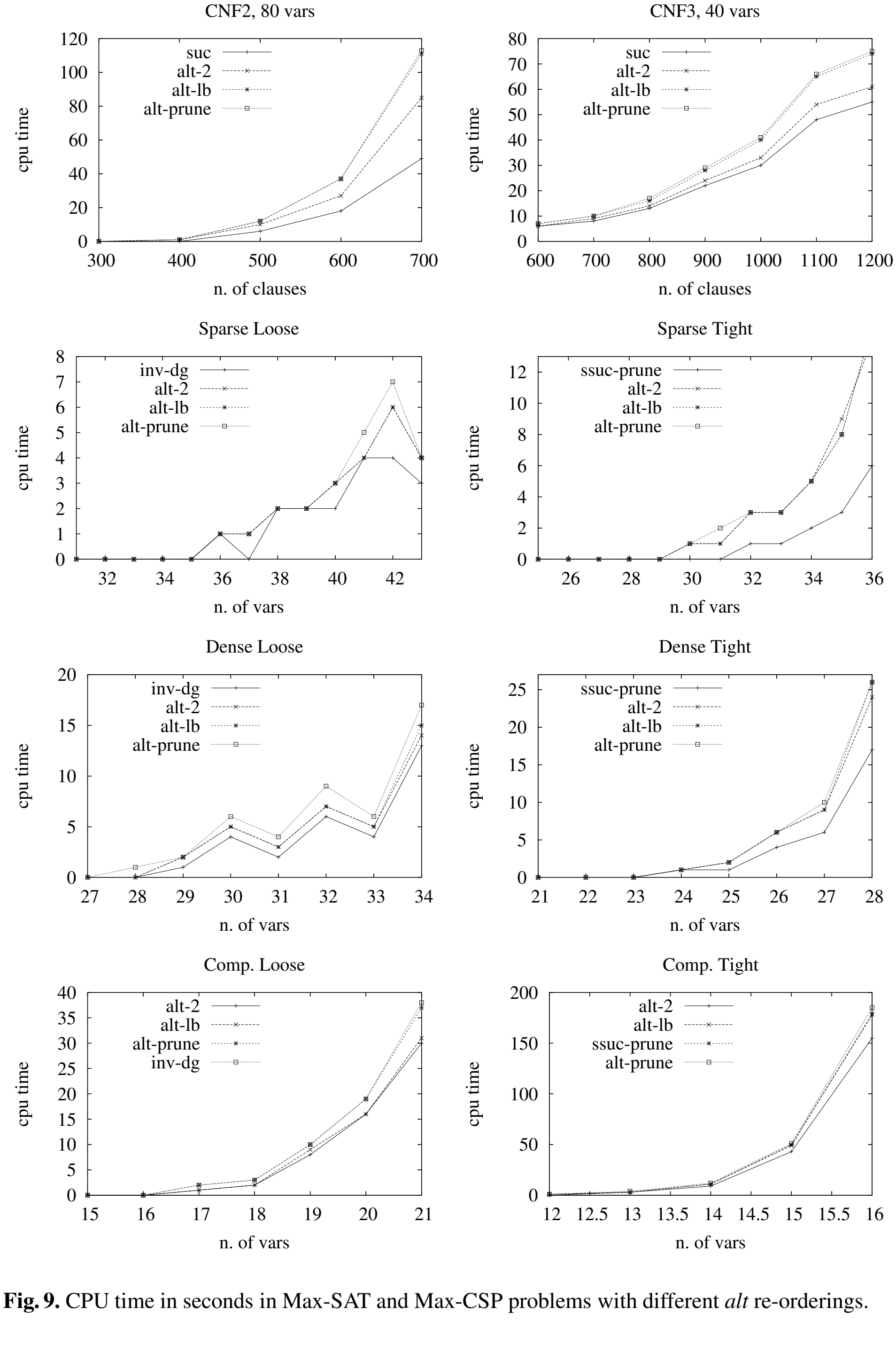

Fig. 9. CPU time in seconds in Max-SAT and Max-CSP problems with different *alt* re-orderings.

5 Related work

Our most recent work in cooperation with de Givry and Zytnicki [5] has lead to a new form of local consistency called *existential directional arc consistency* (EDAC*). EDAC* is stronger than FDAC*, since it requires the following additional property:

$$\forall i \exists a \in D_i, \texttt{s.t.} \forall j \neq i, \exists b \in D_j \texttt{s.t.} C_{ij}(a,b) \oplus C_i(a) \oplus C_j(b) = 0$$

In words, EDAC* requires every variable i to have at least one value $a \in D_i$ with unary cost 0 and fully supported in every other variable. EDAC* can be enforced in time $O(ed^2 \max\{nd, \top\})$ which may be more costly than enforcing FDAC* when $\top > nd$.

A natural question is whether *maintaining* EDAC* will also benefit from intelligent orderings. To answer that question we executed TOOLBAR with the EDAC* option and the best static and dynamic orderings and re-orderings (also with *rnd* for comparison). Figure 10 reports the results of this experiment in terms of visited nodes. It can be observed that *ssuc-prune* visits much less nodes than the other orderings, specially in tight MAX-CSP problems. Figure 11 reports the results in terms of cpu time. We can observe that overhead becomes much more noticeable in EDAC* than in FDAC*. Dynamic orderings and re-orderings do not seem to pay off, except for sparse tight Max-CSP problems. Static orders are the best option in the rest of problems. *rnd* and *inv-dg* always obtain similar results. However, in Max-CSP sparse loose and dense loose problems the best option is *inv-dg* which needs half of the time that *rnd*.

6 Conclusions and future work

In this paper we have shown that when TOOLBAR maintains FDAC*, its performance may depend on the variable order used by the local consistency algorithm. We have proposed several intelligent orders and have analyzed them at three levels of sophistication (and overhead): static, dynamic and dynamic re-ordering. As a result of our experiments we have identified some static orderings that always improve the performance being the inverse degree the most robust one. The best dynamic ordering seems to be the sum of unary costs, but it is not always worthwhile with respect to static orderings. Finally, dynamic re-orderings have been found useful only for certain problems. The same order heuristics have been applied in the newer EDAC* property and we didn't obtain such good results. It shows that FDAC* is more sensitive than EDAC* to the variable ordering. In the future, we want to study how to execute dynamic orders and re-ordering in a more efficient way and we will investigate new useful heuristics for the EDAC* property.

Acknowledgments

The authors are grateful the anonymous referees for their useful comments with which the paper could be improved.

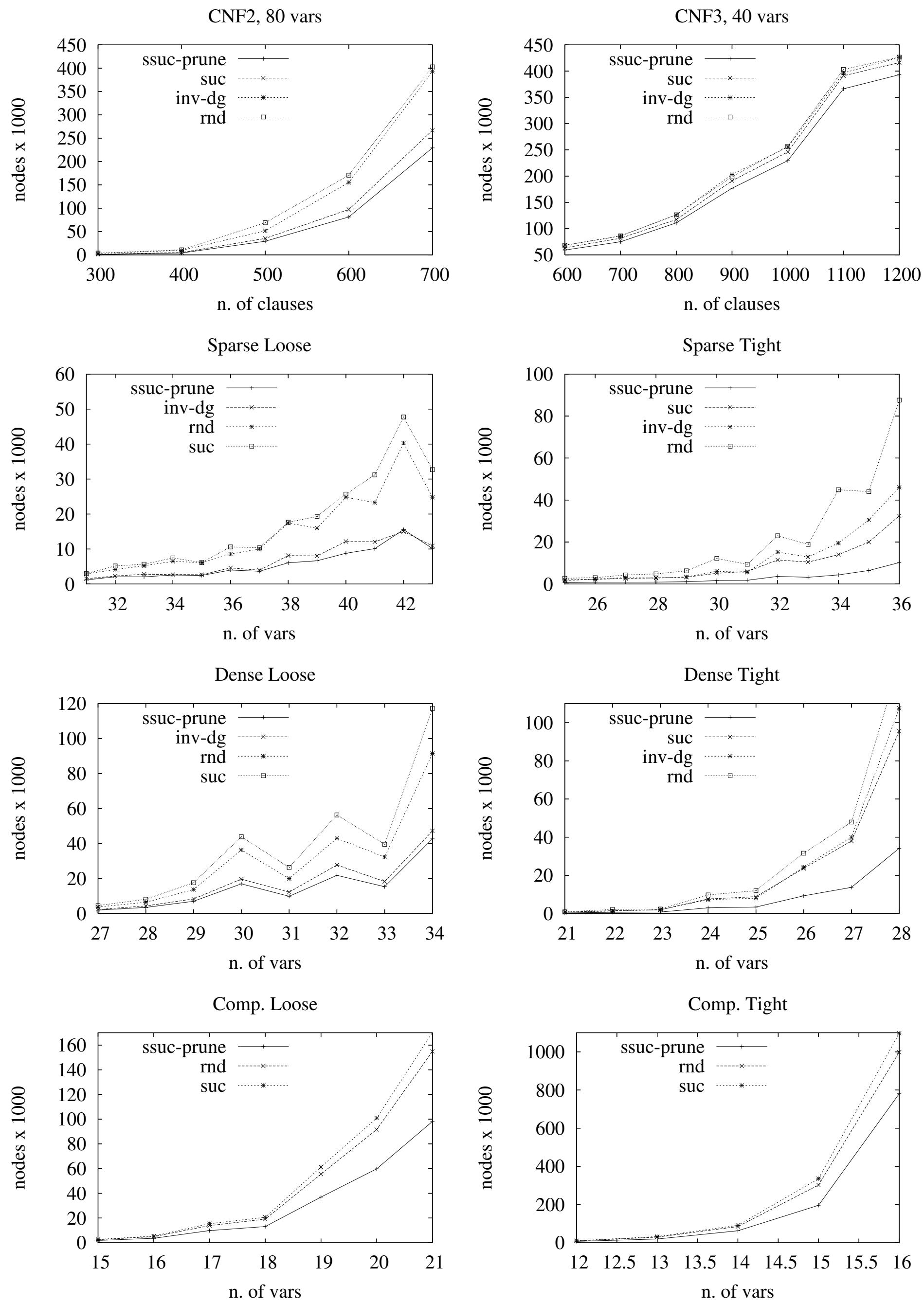

Fig. 10. Visited nodes in Max-SAT and Max-CSP problems with different orderings for EDAC.

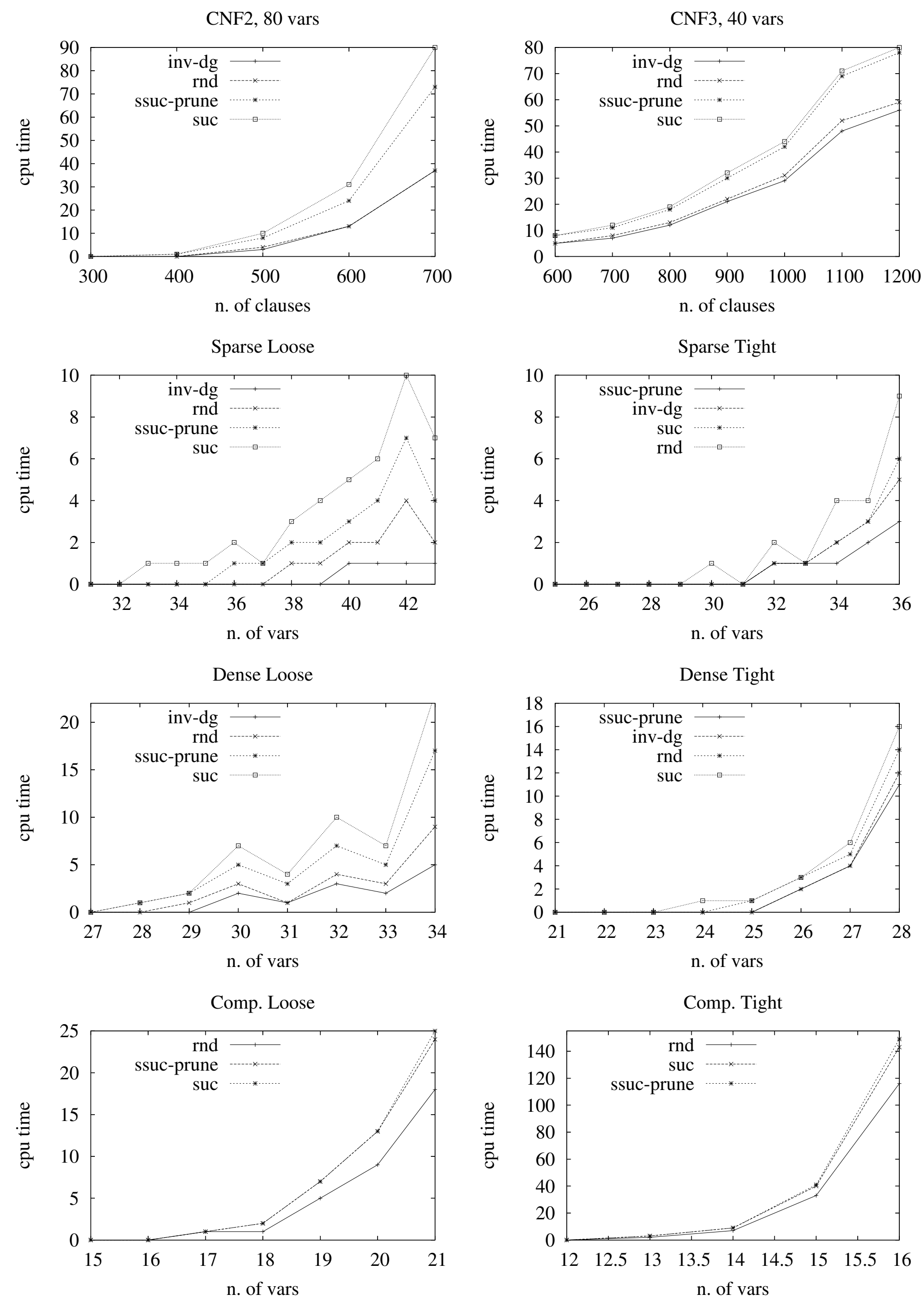

Fig. 11. CPU time in seconds in Max-SAT and Max-CSP problems with different orderings for EDAC.

References

1. C. Bessière and J.-C. Regin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. *Lecture Notes in Computer Science*, 1118:61–75, 1996.
2. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, March 1997.
3. B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Constraints*, 4:79–89, 1999.
4. M. Cooper. Reductions operations in fuzzy or valued constraint satisfaction. *Fuzzy Sets and Systems*, 134(3):311–342, 2003.
5. S. de Givry, F. Heras, J. Larrosa, and M. Zytnicki. Existential arc consistency: getting closer to full arc consistency in weighted csps. In *Proc. of the 19th IJCAI*, Edinburgh, U.K., August 2005.
6. S. de Givry, J. Larrosa, P. Meseguer, and T. Schiex. Solving max-sat as weighted csp. In *Proc. of the 9th CP*, pages 363–376, Kinsale, Ireland, 2003. LNCS 2833. Springer Verlag.
7. R. Dechter. *Constraint Processing*. Morgan Kaufmann, San Francisco, 2003.
8. J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible DAC for Max-CSP. *Artificial Intelligence*, 107(1):149–163, 1999.
9. J. Larrosa and T. Schiex. In the quest of the best form of local consistency for weighted csp. In *Proc. of the 18th IJCAI*, Acapulco, Mexico, August 2003.
10. Javier Larrosa. Node and arc consistency in weighted csp. In *Proceedings of the 18th AAAI*, pages 48–53, 2002.
11. J. Pearl. *Probabilistic Inference in Intelligent Systems. Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, CA, 1988.
12. Tuomas Sandholm. An algorithm for optimal winner determination in combinatorial auctions. In *IJCAI-99*, pages 542–547, 1999.
13. T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: hard and easy problems. In *IJCAI-95*, pages 631–637, Montréal, Canada, August 1995.