Propagate the Right Thing: How Preferences Can Speed-Up Constraint Solving

Christian Bessière **Anaïs Fabre***
LIRMM-CNRS (UMR 5506)
161, rue Ada
F-34392 Montpellier Cedex 5
{bessiere,fabre}@lirmm.fr

Ulrich Junker
ILOG S.A.
1681, route des Dolines
F-06560 Valbonne
ujunker@ilog.fr

Abstract

We present an algorithm Pref-AC that limits arc consistency (AC) to the preferred choices of a tree search procedure and that makes constraint solving more efficient without changing the pruning and shape of the search tree. Arc consistency thus becomes more scalable and usable for many real-world constraint satisfaction problems such as configuration and scheduling. Moreover, Pref-AC directly computes a preferred solution for tree-like constraint satisfaction problems.

1 Introduction

In the last two decades, considerable research effort in AI has been spent on studying the complexity of reasoning and problem solving, and the identification of tractable cases has become an important goal of this research. However, even if a problem can be solved by a polynomial algorithm, or if search effort can be reduced by polynomial algorithms, this might not be sufficient for real-world AI systems that face challenges such as interactivity and real-time. We give some examples from constraint satisfaction, where achieving arc consistency might be too costly, although it is tractable:

1. Web-based configuration systems must provide a response in a few seconds while serving multiple users. Although arc consistency (AC) is an appropriate technique to handle compatibility constraints, it will be too expensive in presence of large product catalogs.
2. Constraint-based approaches to problems such as scheduling, vehicle routing, and personnel planning often only maintain bound consistency in constraints involving time or other variables with large domains. As a consequence, complex rules on breaks, rest-times, and days-off provide only a poor propagation, which could be improved by arc consistency.

These examples illustrate that domain reduction does not develop its full power in many real-world applications of constraint programming. Pruning power and, indirectly, solution quality are traded against short response times. In certain circumstances, such a trade-off can be avoided if the system focuses on the right deductions and computations. In this paper, we argue that preferences are one possible way for achieving this. As stated by Jon Doyle [Doyle, 2002], preferences can play different roles and need not only represent desiderata or user preferences. Preferences can also control reasoning meaning that only preferred inferences and domain reductions are made. Non-interesting deductions are left apart, but may become preferred if additional information is added.

We can easily apply this idea to typical backtrack tree search algorithms that maintain arc consistency when solving a constraint satisfaction problem. The crucial question is whether such an algorithm needs to achieve full arc consistency and to remove all values that are not in the maximal arc-consistent domains. In [Schiex *et al.*, 1996], it has been shown that the fails of the algorithm can be preserved if it simply constructs some arc-consistent domains, thus reducing the computation effort in average. In this paper, we show that the search algorithm can preserve its search strategy and the overall pruning if 1. the search algorithm uses a variable and value ordering heuristics that can be described by (static) preference ordering between variables and values and 2. we construct some arc-consistent domains that contain the preferred values. The first condition is, for example, met by configuration problems [Junker and Mailharro, 2003]. The second condition needs some careful adaption of the definition of arc consistency, which will be elaborated in this paper.

We can thus cut down the propagation effort in each search node without changing the search tree, which leads to an overall gain if the search tree will not be explored completely. This happens if we are interested in one solution, the best solution in a given time frame, or preferred solutions as defined in [Junker, 2002]. Arc consistency thus becomes more scalable and less dependent on the size of domains, which opens exciting perspectives for large-scale constraint programming.

We first introduce preferences (Section 2), use them to define a directed constraint graph (Section 3), and discuss arc consistency for preferred values (Sections 4 and 5). After introducing preferred supports (Section 6), we present the algorithm Pref-AC (Section 7).

2 Preferred Solutions

In this paper, we consider constraint satisfaction problems of the following form. Let $\mathcal{X}$ be a set of variables and $\mathcal{D}$ be a set of values. A constraint c of arity k_c has a sequence of variables $X(c) = \{x_1(c), \ldots, x_{k_c}(c)\}$ from $\mathcal{X}$ and a relation R_c

*A. Fabre's work has been carried out during a stay at ILOG.

that is a set of tuples from $\mathcal{D}^{k_c}$. Let $\mathcal{C}$ be a set of constraints. The triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ is then called a *constraint network*.

A *solution* of a constraint network is a mapping ν of the variables $\mathcal{X}$ to the values in $\mathcal{D}$ such that each constraint is satisfied, i.e., $(\nu(x_1(c)), \ldots, \nu(x_{k_c}(c))) \in R_c$ for all constraints c. The *constraint satisfaction problem* (*CSP*) involves finding a solution of a constraint network.

Alternatively, we can represent the tuples by sets of assignments of the form $x = v$ between a variable x and a value v: if t is a tuple of the form $(v_1, \ldots, v_{k_c})$ in R_c then let $\hat{t}$ be the set of assignments $\{x_1 = v_1, \ldots, x_{k_c} = v_{k_c}\}$. Furthermore, let $\hat{R}_c$ be the set of all $\hat{t}$ such that t is in R_c. This representation will facilitate definitions and permits a straightforward logical characterization of a constraint c:

$$\bigvee_{t \in R_c} \bigwedge_{(x=v) \in \hat{t}} (x = v) \tag{1}$$

We further suppose that the following kinds of preferences are given. For each variable x, we introduce a strict partial order $\prec_x \subseteq \mathcal{D} \times \mathcal{D}$ among the possible values for x. This order can represent user preferences as occurring in configuration problems (cf. [Junker and Mailharro, 2003]). For example, the user might prefer a red car to a white car. Projecting preferences on criteria to decision variables [Junker, 2002] also produces such an order (e.g. if price is minimized then try cheaper choices first). Furthermore, we consider a strict partial order $\prec_{\mathcal{X}} \subseteq \mathcal{X} \times \mathcal{X}$ between variables. For example, we may state that choosing a color is more important than choosing a seat material and should therefore be done first. In the sequel, we suppose that the search procedure follows these preferences as described in [Junker and Mailharro, 2003] and always chooses a $\prec_{\mathcal{X}}$-best variable and a $\prec_x$-best value for x. These preferences are given statically and thus correspond to a static variable and value ordering heuristics, which are usually sufficient for configuration problems.

We now use the preferences to define a preferred solution. First of all we choose a linearization of the orders $\prec_x$ and $\prec_{\mathcal{X}}$ in form of total orders $<_x$ and $<_{\mathcal{X}}$ that are supersets of the strict partial orders. Then we consider a ranking $\chi(1), \ldots \chi(n)$ of the variables in $\mathcal{X}$ that corresponds to the order $<_{\mathcal{X}}$, i.e., $\chi(i) <_{\mathcal{X}} \chi(j)$ iff $i < j$. We then consider two solutions ν_1 and ν_2 and compare them lexicographically

$$\begin{gathered}(\nu_1(\chi(1)), \ldots, \nu_1(\chi(n))) <_{lex} (\nu_2(\chi(1)), \ldots, \nu_2(\chi(n))) \\ \text{iff} \\ \exists k : \nu_1(\chi(k)) <_{\chi(k)} \nu_2(\chi(k)) \text{ and} \\ \nu_1(\chi(i)) = \nu_2(\chi(i)) \text{ for all } i = 1, \ldots, k-1\end{gathered} \tag{2}$$

Definition 1 *A solution ν of a constraint network $\mathcal{P} := (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is a* preferred solution *of $\mathcal{P}$ iff there exist linearizations $<_{\mathcal{X}}$ of $\prec_{\mathcal{X}}$ and $<_x$ of $\prec_x$ for each $x \in \mathcal{X}$ s.t. ν is the best solution of $\mathcal{P}$ w.r.t. the lexicographical order defined by $<_{\mathcal{X}}$ and $<_x$.*

These preferred solutions correspond to the extreme solutions in [Junker, 2002] that are obtained if all variables are criteria. If the search procedure follows the preferences then its first solution is a preferred solution. Hence, the notion of a preferred solution helps us to forecast the first solution in certain cases.

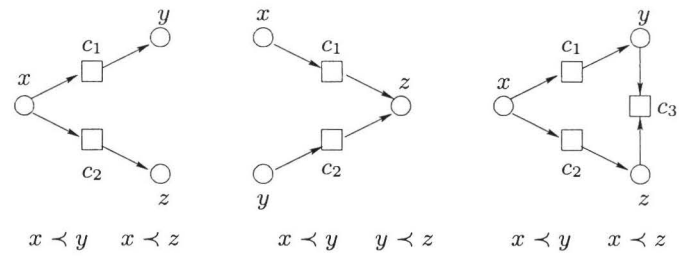

Figure 1: Directed constraint graphs.

3 Preference-based Constraint Graph

In order to find a preferred solution, the search procedure chooses values for more important variables first. We will see later that, in some cases, we can anticipate a preferred solution and construct it by a constraint propagation procedure if this procedure does not only follow the constraints, but also the order among the variables used by the search procedure.

In order to illustrate this, consider the bipartite constraint graph that connects the constraints with their variables. Suppose that a variable x is connected via a constraint c to a less important variable y. Search will then assign a value to x and cause a domain reduction of y. We can thus say that the preferences $\prec_{\mathcal{X}}$ among variables impose an order on the constraint graph. For example, we will say that the arc between x and c leads from x to c and the arc between y and c leads from c to y.

We generalize this idea for arbitrary constraints and partial orders. For each constraint c, we consider the $\prec_{\mathcal{X}}$-best variables in $X(c)$ and call them input variables. The other variables are called output variables. Arcs lead from input variables to a constraint and from there to the output variables.

Definition 2 *Let $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a constraint network and $\prec_{\mathcal{X}}$ be a strict partial order between the variables $\mathcal{X}$. A variable x of a constraint c is called* input variable *of c iff there is no other variable y of c s.t. $y \prec_{\mathcal{X}} x$. A variable x of c is called* output variable *of c iff it is not an input variable of c. The* directed constraint graph *of the CSP is a bipartite graph having the nodes $\mathcal{X} \cup \mathcal{C}$ and the set of arcs (x, c) s.t. x is an input variable of c and (c, y) s.t. y is an output variable of c.*

Each constraint can have several input and several output variables (cf. figure 1). If its variables are not comparable at all, then the constraint has no output variable, and it is a sink of the directed graph. However, each constraint has at least one input variable. As a consequence, no constraint can be a source of the directed graph. In general, the graph is not guaranteed to be acyclic, but it satisfies the following properties which are needed for our proofs: 1. All $\prec_{\mathcal{X}}$-best variables of the set $\mathcal{X}$ are sources of the constraint graph since they cannot be an output variable of any constraint. However there can be sources that are not $\prec_{\mathcal{X}}$-best variables. 2. If χ is a ranking of the variables $\mathcal{X}$ that is a linearization of $\prec_{\mathcal{X}}$ then each output variable of each constraint is preceded by at least one input variable of the same constraint in the ranking.

4 Arc Consistency

A typical systematic search procedure for solving CSPs proceeds as follows. In each search node, it picks an assignment $x = v$ for such a variable and creates two successor nodes, one for $x = v$ and one for $x \neq v$. Throughout the paper, we assume that a variable x is eliminated from the CSP of a successor node by a preprocessing step iff it has only a single possible value. Usually, the search procedure will not select an arbitrary assignment from the set $\mathcal{A}$ of all assignments, but tries to anticipate certain cases where the sub-tree of an assignment does not contain a solution. If such an assignment is detected it is eliminated from $\mathcal{A}$. Hence, the search procedure maintains a set $A \subseteq \mathcal{A}$ of possible assignments and considers only elements of A. Each solution can be characterized by a subset of A that assigns exactly one value to each variable and that satisfies all constraints. As a consequence, if a variable x has no assignment in A then no solution exists and the search will backtrack. In the sequel, we write A_x for the set $\{v \in \mathcal{D} \mid (x = v) \in A\}$ of possible values for x in A.

Assignments can be eliminated from A if they lack a support on a constraint:

Definition 3 *A set of assignments S is a* support *for $x = v$ in c iff 1. $S \cup \{x = v\}$ is an element of $\hat{R}_c$ and 2. $(x = v) \notin S$.*

Let $\mathcal{S}^c_{x=v}$ be the set of supports for $x = v$ in c. In this paper, we are interested in search procedures that maintain arc consistency, i.e. that work with a set A where all assignments have supports S that are subsets of A and this on all relevant constraints:

Definition 4 *Let $\mathcal{A}$ be a set of assignments. A subset A of $\mathcal{A}$ is an* arc-consistent set *of $\mathcal{A}$ iff 1. A_x is non empty for any $x \in \mathcal{X}$ and 2. for all $(x = v) \in A$ and all constraints c of x, there exists a support S for $(x = v)$ in c such that S is a subset of A.*

If an arc-consistent set A contains a single assignment for each variable then A corresponds to a single solution. If no arc-consistent set exists then the CSP has no solution. Otherwise, there is a unique maximal arc-consistent set, which is a superset of all arc-consistent sets, including the solutions.

Alternatively, we can consider a set Δ of some assignments for which we know that they do not belong to any solution:

$$\text{if all supports } S \text{ in } c \text{ for } (x = v) \text{ have an element } (y = w) \text{ in } \Delta \text{ then } (x = v) \text{ is in } \Delta. \tag{3}$$

This can easily be translated to a reflexive and monotonic operator that eliminates additional assignments:

$$\rho_{\mathcal{A}}(\Delta) := \Delta \cup \{(x = v) \in \mathcal{A} \mid \forall S \in \mathcal{S}^c_{x=v} : S \not\subseteq \mathcal{A} - \Delta\} \tag{4}$$

Fixed-point theory then tells us that the transitive closure of this operator is the unique minimal set Δ^* satisfying equation 3. As a consequence, the set A^* of all assignments that are not eliminated (i.e., $A^* := \mathcal{A} - \Delta^*$) is the unique maximal set satisfying item 2 of def. 4. If each variable has an assignment in A^* then A^* is the maximal arc-consistent assignment. If some variable has no assignment in A^* no arc-consistent assignment exists. In this case, no solution exists.

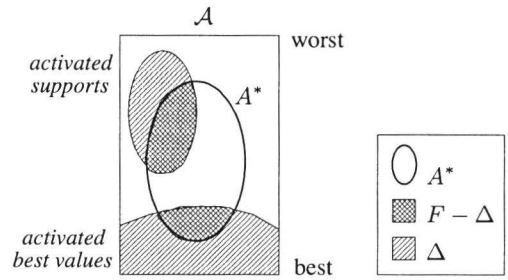

Figure 2: Activating best values and their supports.

5 Arc Consistency for Preferred Values

Maintaining arc consistency prunes the search tree as follows:

P1 if the current CSP is not arc-consistent then a fail occurs.

P2 if a value v is not arc-consistent for a variable x then the search procedure will not assign v to x inside the current subtree.

In general, it is desirable to eliminate as many values as possible and to determine A^* by applying an algorithm of the AC-family. The worst-case complexity is $O(|\mathcal{C}| \cdot |\mathcal{D}|^2)$ for binary constraints. If domains contain thousands or more elements as is the case for typical scheduling problems and for configuration problems with large catalogs, then maintaining arc consistency is no longer feasible in practice.

Interestingly, it is not necessary to eliminate all assignments of $\mathcal{A} - A^*$. [Schiex *et al.*, 1996] try to construct a small arc-consistent set A. If this succeeds then pruning rule P1 cannot be triggered. This can lead to saving time in search nodes that are arc-consistent. We now want to extend this policy to pruning rule P2 in the case that the original search procedure uses the given preferences between variables and values in order to select an assignment. In each search state, the search procedure will pick a $\prec_{\mathcal{X}}$-best variable and try out its possible values starting with the best value v^*. If $x = v^*$ is not in the maximal arc-consistent set A^* then we must ensure that it will be eliminated when constructing the subset A. However, if $x = v^*$ is in A^* then we must ensure that it will be in the subset A.

Instead of doing a pure elimination process, we interleave construction and elimination steps and determine a set F of *activated assignments*, called *focus*, and a set Δ of *eliminated assignments*. Both sets will grow during the process and our purpose is to obtain $F - \Delta$ as arc-consistent set. This set is sufficient for our search procedure if some properties are satisfied: 1. If x is a $\prec_{\mathcal{X}}$-best variable and v is a $\prec_x$-best among the non-eliminated values of x then the search procedure might pick $x = v$ as next assignment. Hence, we have to activate it. 2. If an activated assignment does not have a support on a constraint c then it has to be eliminated. 3. The supports for the activated, but non-eliminated assignments must only contain activated and non-eliminated elements. 4. We need to guarantee that $F - \Delta$ contains at least one value per variable if an arc-consistent set exists. Since all variables can be reached from the sources via constraints, we activate best values for all sources of the directed constraint graph.

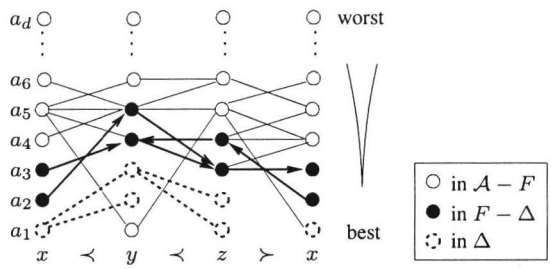

Figure 3: Example for a stable activation.

Definition 5 *Let $\mathcal{A}$ be a set of initial assignments and F, Δ be subsets of $\mathcal{A}$ s.t. $\Delta \subseteq F$. We say that (F, Δ) is an $\prec$-*activation *of $\mathcal{A}$ iff the following properties hold:*

1. *If x is a source and $(x = w) \in \Delta$ for all $w \in \mathcal{A}$ with $w \prec_x v$ then $x = v$ is in F.*
2. *If $(x = v) \in F$, $x \in X(c)$, and c has no support S for $x = v$ s.t. $S \subseteq \mathcal{A} - \Delta$ then $(x = v) \in \Delta$.*

An $\prec$-activation (F, Δ) is stable *iff all $(x = v) \in F - \Delta$ have a support $S \subseteq F - \Delta$ on all constraints c containing x.*

Proposition 1 *Let $\mathcal{A}$ be a set of initial assignments and $\Delta^* := \rho^*_{\mathcal{A}}(\emptyset)$. Then $(\mathcal{A}, \Delta^*)$ is a stable activation.*

Hence stable activations always exist and coincide with the maximal arc-consistent set in the worst case. Maintaining stable activations is sufficient to achieve pruning rules P1 and P2 for search procedures that always pick the best elements of $\mathcal{A} - \Delta$ (cf. figure 2):

Proposition 2 *Let (F, Δ) be a stable $\prec$-*activation *of $\mathcal{A}$. If there is no arc-consistent subset of $\mathcal{A}$ then there exists a variable x s.t. $F - \Delta$ does not contain an assignment for x. If there is an arc-consistent subset of $\mathcal{A}$ and A^* is the maximal arc-consistent subset of $\mathcal{A}$ then the following properties hold:*

1. *$F - \Delta$ is an arc-consistent subset of $\mathcal{A}$.*
2. *If x is a $\prec_{\mathcal{X}}$-best variable and $x = v$ is a $\prec_x$-best assignment for x in A^* then $x = v$ is in $F - \Delta$.*
3. *If x is a $\prec_{\mathcal{X}}$-best variable, $x = v$ is a $\prec_x$-best assignment for x in A^*, and $w \prec_x v$ then $x = w$ is in Δ.*

If the search procedure chooses a $\prec$-best assignment for a best variable in each step, then it will explore the same search tree independently of whether it maintains full arc consistency or whether it maintains stable activations. Since the first solution found by the standard search procedure is a preferred solution, we will still obtain a preferred solution first if we maintain stable activations.

We now discuss how to actually construct stable $\prec$-activations. Initially, we activate the best assignments in $\mathcal{A}$ for all relevant variables by a monotonic operator ϕ:

$$\begin{aligned}\phi_{\mathcal{A}}(\Delta) := \{(x = v) \in \mathcal{A} \mid x \text{ is a source,} \\ \forall w \in \mathcal{A} \text{ s.t. } w \prec_x v : (x = w) \in \Delta\}\end{aligned} \tag{5}$$

If an activated assignment is not supported by a constraint it will be eliminated and replaced by the next best assignments until we reach a set of assignments that have a support in each constraint. Unsupported assignments are eliminated by an operator δ, which restricts ρ to the focus F:

$$\begin{aligned}\delta_{\mathcal{A}}(F, \Delta) := \{(x = v) \in F \mid \exists c : \forall S \in \mathcal{S}^c_{x=v} : \\ S \not\subseteq \mathcal{A} - \Delta\}\end{aligned} \tag{6}$$

We can iterate both operators if we combine them into an operator σ that maps a pair of assignment sets to a pair of assignment sets s.t. $\sigma_{\mathcal{A}}((F, \Delta)) := (F \cup \phi_{\mathcal{A}}(\Delta), \Delta \cup \delta_{\mathcal{A}}(F, \Delta))$. We easily obtain a lattice on these pairs if we define union and intersection of two pairs by applying these operations to both elements. The operator $\sigma_{\mathcal{A}}$ is monotonic and reflexive in this lattice and its transitive closure $\sigma^*_{\mathcal{A}}((F, \Delta))$ is the smallest $\prec$-activation (F^*, Δ^*) that extends (F, Δ).

Next we discuss how an activation containing unsupported assignments can be extended to a stable one. We say that assignment $x = v$ is *unsupported* on c in (F, Δ) if c does not have a support S for $x = v$ s.t. $S \subseteq F - \Delta$. If no such assignment exists, we have found a stable activation. Otherwise, we pick an unsupported assignment, choose a support S for it, and activate the elements of this support. After this, we apply operator $\sigma_{\mathcal{A}}$ to produce a new activation:

Proposition 3 *Let (F, Δ) be an $\prec$-activation for $\mathcal{A}$. Suppose $x = v$ is unsupported on c in (F, Δ). Then c has a support S for $x = v$ s.t. $S \subseteq \mathcal{A} - \Delta$ and $S \not\subseteq F - \Delta$ and $\sigma^*_{\mathcal{A}}((F \cup S, \Delta))$ is an activation that supports $x = v$ on c.*

We can thus construct a stable activation as follows:

Definition 6 *Let $\mathcal{A}$ be a set of initial assignments. Let $S_1, \ldots, S_m$ be subsets of $\mathcal{A}$. We then define $(F_0, \Delta_0) := \sigma^*_{\mathcal{A}}((\emptyset, \emptyset))$ and $(F_i, \Delta_i) := \sigma^*_{\mathcal{A}}((F_{i-1} \cup S_i, \Delta_{i-1}))$ for $i = 1, \ldots, m$. The sequence $S_1, \ldots, S_m$ is called an* activation sequence *iff 1. $S_i \not\subseteq F_{i-1}$, 2. each S_i is a support on a constraint c for an assignment $(x = v)$ in $F_{i-1} - \Delta_{i-1}$, and 3. (F_m, Δ_m) is stable.*

Since F_i has more elements than F_{i-1} and $(\mathcal{A}, \rho^*_{\mathcal{A}}(\emptyset))$ is a stable activation, we can show the following result, which permits the construction of stable activations:

Proposition 4 *Let $\mathcal{A}$ be a set of initial assignments. There exists an activation sequence $S_1, \ldots, S_m$ for $\mathcal{A}$.*

Figure 3 gives an idea of the possible gains. Once the dashed assignments have been eliminated, we activate only two assignments per variable, i.e. six in total, whereas an AC-algorithm would activate each of the $3 \cdot |\mathcal{D}|$ assignments in $\mathcal{A}$. Hence, we obtain significant gains if domains are large and no elimination is required. The elimination of an assignment still costs $O(|\mathcal{D}|)$ checks.

6 Preferred Supports

Preferences between values and variables allow us to define preferred supports. A support $S \subseteq \mathcal{A} - \Delta$ for an assignment $x = v$ in constraint c can be considered a solution of the CSP $(X(c), \mathcal{D}, \{c, x = v\} \cup \{y \neq w \mid (y = w) \in \Delta\})$. We say that S is a *preferred support* for $x = v$ on c iff S is a preferred solution of this CSP. An assignment can have several preferred supports. In case of binary constraint networks, preferred supports are unique if all domain orders $\prec_x$ are total. In general, preferred supports are unique if additionally the order $\prec_{\mathcal{X}}$ is total as well.

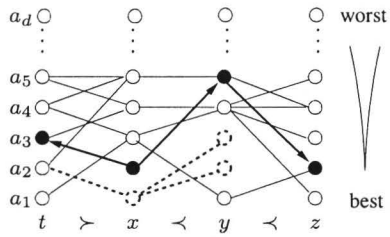

Figure 4: A preferred solution as stable activation.

Given a preferred solution, we cannot guarantee that each assignment has a preferred support in all constraints. However, if certain assignments in a solution S have preferred supports then S is preferred solution:

Proposition 5 *Let S be a solution of $\mathcal{P}$. Suppose that $\prec_{\mathcal{X}}$ and all $\prec_x$ are strict total orders. If each assignment $x = v$ to a source x of the direct constraint graph is a best assignment for x in the unique maximal arc-consistent set A^* and all activated assignments to input variables of a constraint c have a preferred support on c in S then S is a preferred solution of $\mathcal{P}$.*

When constructing a stable activation, we choose preferred supports since this may result in a preferred solution:

Proposition 6 *Suppose that $\prec_{\mathcal{X}}$ and all $\prec_x$ are strict total orders. Let $S_1, \ldots, S_m$ be an activation sequence and let (F_i, Δ_i) be defined as in def. 6. If 1. each S_i is a preferred support for some $x = v$ on some constraint c s.t. x is an input variable of c or $x = v$ is unsupported on c in (F_{i-1}, Δ_{i-1}) and 2. $F_m - \Delta_m$ is a solution of $\mathcal{P}$ then $F_m - \Delta_m$ is a preferred solution of $\mathcal{P}$.*

The first condition in proposition 6 can be satisfied by always activating preferred supports. The second condition can be met by CSPs of a special structure. For example, consider a binary constraint network such that its directed constraint graph forms a tree (cf. figure 4). In this case, we only activate supports for assignments to input variables, but not for assignments to output variables, since each variable can only be an output variable of a single constraint. This result is, for example, interesting for certain car configuration problems and permits us to efficiently take into account certain user preferences.

Preferred solutions are also obtained in trivial cases:

Proposition 7 *Suppose that $\prec_{\mathcal{X}}$ and all $\prec_x$ are strict total orders. If there is a preferred solution containing all best values of all variables in $\mathcal{A}$ then it is equal to all activations produced by preferred supports.*

7 Algorithm Pref-AC

In this Section, we present Pref-AC, an algorithm for computing an arc-consistent set A from a binary constraint network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$. This algorithm activates supports for unsupported elements and meets the requirements of definitions 5 and 6, and proposition 6. An algorithm for non-binary constraints can be derived from this one. Pref-AC follows the principles of lazy arc consistency [Schiex *et al.*, 1996] in order to build an arc-consistent set of assignments that is not necessarily maximal. We base Pref-AC on AC-6 [Bessière, 1994] for keeping the discussion simple, although we could exploit the bi-directionality of supports [Bessière *et al.*, 1999].

AC-6 assigns an ordering of the values in the domain of every variable x_i, checks one support (the *first* one or *smallest* one with respect to the ordering) for each assignment $(x_i = a)$ on each constraint $c(x_i, x_j)$ to prove that $(x_i = a)$ is currently viable. When $(x_j = b)$ is found as the smallest support of $(x_i = a)$ on $c(x_i, x_j)$, $(x_i = a)$ is added to $CS(x_j, b)$, the list of assignments currently having $(x_j = b)$ as smallest support. If b is removed from the domain of x_j then AC-6 looks for the next support in the domain of x_j for each assignment $x_i = a$ in $CS(x_j, b)$.

Pref-AC uses the domain order $<_{x_i}$, a chosen linearization of $\prec_{x_i}$, when looking for supports in the domain of a variable x_i. Furthermore, Pref-AC seeks supports only for elements of A, the current set of activated and non-eliminated values (i.e., $F - \Delta$). Pref-AC uses following data structure:

- Each variable x_i has a fixed initial domain $\mathcal{D}_{x_i}$, ordered by $<_{x_i}$, containing all values from $\mathcal{D}$ except those that violate unary constraints on x_i. The set $\overline{\Delta}(x_i) := \mathcal{D}_x - \Delta(x_i)$ contains the values of $\mathcal{D}_{x_i}$ that have not yet been removed by arc consistency.
- A is the arc-consistent set built by the algorithm.
- For each $(x_i = a) \in A$, a set $CS(x_i, a)$ contains the assignments that are currently supported by $(x_i = a)$.
- The set Pending contains all the 4-tuples (x_i, a, x_j, b) such that a support for $(x_i = a)$ has to be sought on $c(x_i, x_j)$. If $b \neq$ **nil**, this means that all values of x_j better than b have already been removed.

Pref-AC uses two subprocedures (Algorithm 1). Procedure `Activate`(x_i, a) adds an assignment $(x_i = a)$ to A, the set of current activated values. It initializes its data structure CS, and puts in Pending all the information needed to look for supports for $(x_i = a)$ on the constraints involving x_i. Function `BestSup`(x_i, a, x_j, b, B) looks for the best support for $(x_i = a)$ in B which is less preferred than b (we know there is no support in B better than or equal to b). B is $\overline{\Delta}(x_j)$ or A_{x_j} depending on the status of x_i (input variable or not).

Pref-AC works as follows. We start by initializing $\overline{\Delta}(x_i)$ to $\mathcal{D}_{x_i}$ for each x_i (line 1 of Algorithm 2). We activate the best value of the best variable (line 2). Then, 4-tuples (x_i, a, x_j, b) are taken from the Pending set (line 4), and if $(x_i = a)$ is still active (line 5), we must seek a (new) support c for it. If x_i is an input variable of c_{ij}, we seek the support in $\overline{\Delta}(x_j)$ because we have to ensure that it will be a preferred support (lines 6 to 7). Otherwise, we first seek a support c among the activated, but non-eliminated elements (line 8). If none exists, we seek a new best support c (line 10) following def. 6. If c exists, we activate it if not yet done, and store the fact that it supports $(x_i = a)$ (lines 11 to 13). If no support exists for $(x_i = a)$, we remove it from $\overline{\Delta}$ and A (line 14). If $\overline{\Delta}(x_i)$ is empty, a wipe out stops the procedure (line 15). If x_i is a source of the directed constraint graph and the best value of $\overline{\Delta}(x_i)$ is not in A_{x_i} ($(x_i = a)$ was the best of A_{x_i} and $\overline{\Delta}(x_i)$), the best value

Algorithm 1: Subprocedures

procedure `Activate` **(in** x_i: variable; a: value)

$A_{x_i} \leftarrow A_{x_i} \cup \{a\}$;
$CS(x_i, a) \leftarrow \emptyset$;
Pending $\leftarrow \{(x_i, a, x_j, \textbf{nil}) \mid c(x_i, x_j) \in C\}$;

function `BestSup` **(in** x_i; a; x_j; b; B**):** value

/* returns the best value in B supporting $(x_i = a)$, or **nil** if not found */ ;
if $b = \textit{nil}$ **then** $b \leftarrow best_{<_{x_j}}(B)$;
else
  if $b > max(B)$ **then return nil** ;
  $b \leftarrow next_{<_{x_j}}(b, B)$;
while $(b \neq \textit{nil})$ **do**
  if $(a, b) \in c(x_i, x_j)$ **then return** b;
  else $b \leftarrow next_{<_{x_j}}(b, B)$;
return nil;

of $\overline{\Delta}(x_i)$ has to be activated (lines 16 to 18). Finally, we have to put in Pending the information needed to propagate the deletion of $(x_i = a)$ (line 19).

Algorithm 2: `Pref-AC`

procedure Pref-AC()

1 $\overline{\Delta}(x_i) \leftarrow \mathcal{D}_{x_i}, \forall x_i \in \mathcal{X}$; Pending $\leftarrow \emptyset$;
2 **for each** *source* x **do** `Activate`$(x, best_{<_x}(\overline{\Delta}(x)))$;
3 **while** *Pending* $\neq \emptyset$ **do**
4 pick (x_i, a, x_j, b) from Pending ;
5 **if** $(x_i = a) \in A$ **then**
6 **if** x_i *is an input variable of* c_{ij} **then**
7 $c \leftarrow$ `BestSup`$(x_i, a, x_j, b, \overline{\Delta}(x_j))$;
  else
8 $c \leftarrow$ `BestSup`$(x_i, a, x_j, b, A_{x_j})$;
9 **if** $c = \textit{nil}$ **then**
10 $c \leftarrow$ `BestSup`$(x_i, a, x_j, b, \overline{\Delta}(x_j) - A_{x_j})$;
11 **if** $c \neq \textit{nil}$ **then**
12 **if** $c \notin A_{x_j}$ **then** `Activate`(x_j, c);
13 put $(x_i = a)$ in $CS(x_j, c)$;
  else
14 $\overline{\Delta}(x_i) \leftarrow \overline{\Delta}(x_i) - \{a\}$; $A \leftarrow A - \{(x_i = a)\}$;
15 **if** $\overline{\Delta}(x_i) = \emptyset$ **then return false** ;
16 **if** x_i *is a source* **then**
17 **if** $best_{<_{x_i}}(\overline{\Delta}(x_i)) \notin A_{x_i}$ **then**
18 `Activate`$(best_{<_{x_i}}(\overline{\Delta}(x_i)))$;
19 **for each** $(x_j, b) \in CS(x_i, a)$ **do**
  put (x_j, b, x_i, a) in Pending ;
return true ;

If Pref-AC terminates with an empty set of pending propagations (line 3), A only contains assignments for which a support has been activated on each constraint (because of line 12), and not deleted (because of line 19). In addition, this support is the preferred support for this assignment on this constraint if the variable was an input variable for the constraint (because of line 7 and the way `BestSup` works). We know A contains at least an assignment per variable because of the activation of supports. Therefore, A is an arc-consistent set. And because of lines 16-18, we know that for each x_i which is a source of $\mathcal{X}$, the $<_{x_i}$-*best* of $A^*_{x_i}$ is in A_{x_i}.

The space complexity of Pref-AC is the same as AC-6, namely $O(|\mathcal{C}| \cdot |\mathcal{D}|)$, since in the worst case, all the values are activated and have a support stored on each constraint. The time complexity is also bounded above by that of AC-6, namely $O(|\mathcal{C}| \cdot |\mathcal{D}|^2)$, since for each value in each domain, we perform at most $|\mathcal{D}|$ constraint checks on each constraint.

8 Conclusion

We have shown that it is sufficient to maintain an arc-consistent set for the preferred choices of a search procedure. This can significantly speed up constraint solving if the variable and value ordering heuristics of the search procedure can be described by preferences as is the case for typical configuration problems. We developed an algorithm called Pref-AC for achieving this reduction and are currently testing it for configuration problems with large domains.

Many real-world applications of constraint programming suffer from insufficient propagation, since most approaches support only bound consistency for crucial constraints such as precedences of activities, resources, and rest-time rules. Adapting algorithm Pref-AC to those constraints provides an interesting future perspective for improving constraint solving in areas such as scheduling, personnel planning, and other logistics applications. Future work will be devoted to improve Pref-AC such that it directly finds a preferred solution if the problem structure permits this (e.g. by activating preferred supports that are common to several variables).

Acknowledgements

We would like to thank Jean-Charles Régin and Olivier Lhomme for very helpful discussions.

References

[Bessière *et al.*, 1999] C. Bessière, E.C. Freuder, and J.C. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107:125–148, 1999.

[Bessière, 1994] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

[Doyle, 2002] Jon Doyle. Preferences: Some problems and prospects. In *AAAI-02 Workshop on Preferences in AI and CP: Symbolic Approaches*. AAAI Press, 2002.

[Junker and Mailharro, 2003] Ulrich Junker and Daniel Mailharro. Preference programming: Advanced problem solving for configuration. *AI-EDAM*, 17(1), 2003.

[Junker, 2002] Ulrich Junker. Preference-based search and multi-criteria optimization. In *AAAI-02*, pages 34–40, Menlo Park, CA, 2002. AAAI Press.

[Schiex *et al.*, 1996] T. Schiex, J.C. Régin, C. Gaspin, and G. Verfaillie. Lazy arc consistency. In *AAAI-96*, pages 216–221, 1996.