M.R.C. van Dongen and Christophe Lecoutre (Ed.)

Proceedings of the Third International Workshop on

Constraint Propagation and Implementation

Nantes, France, September 2006

Held in conjunction with the
Twelfth International Conference on
Principles and Practice of
Constraint Programming (CP 2006)

Preface

Constraint Propagation is an essential part of many constraint programming systems. Sitting at the heart of a constraint solver, it consumes a significant portion of the time that is required for problem solving.

The *Third International Workshop on Constraint Propagation and Implementation* (CPAI'06) was convened to study the design and analysis of new propagation algorithms as well as any related practical issue. The implementation and evaluation of constraint propagation algorithms is studied in contexts ranging from special purpose solvers to programming language systems.

These proceedings are dedicated to the CPAI'06 workshop. They include an abstract of an invited talk, deliverd by Chris Jefferson, and seven contributed papers. In his talk Chris Jefferson presents his solver Minion. The contributed papers present filtering algorithms for precedence and dependency constraints, a new global meta-constraint called SLIDE, a study of residual supports in arc consistency, algorithms for maintaining singleton arc consistency, algorithms for probabilistic singleton arc consistency, improvements and simplifications of the SPREAD constraint, and filtering algorithms for the graph isomorphism problem.

We wish to thank the authors for submitting their work, the invited speaker, Chris Jefferson, the CP 2006 Workshop/Tutorial Chair, Barry O'Sullivan, the members of the CPAI'2006 Programme Committee, and the additional reviewers.

August 2006

Marc van Dongen
Christophe Lecoutre

CPAI'06 Organising Committee

Organisation

CPAI'06 Programme Committee

Christian Bessière	Université de Montpellier
Romuald Debruyne	Ecole des Mines de Nantes, France
Fred Hemery	Université d'Artois, France
Christophe Lecoutre	Université d'Artois, France
Peter van Beek	University of Waterloo, Canada
Marc van Dongen	University College Cork, Ireland
Willem-Jan van Hoeve	Cornell University, USA
Pascal Van Hentenryck	Brown University, USA
Rick Wallace	Cork Constraint Computation Centre, Ireland
Roland Yap	National University of Singapore, Singapore
Yuanlin Zhang	Texas Tech University, USA

CPAI'06 Additional Reviewers

Deepa Mehta	University College Cork, IReland
Mark Hennessy	Cork Constraint Computation Centre, Ireland

Table of Contents

Invited Talks

Regular Papers

Constants Matter: Implementing Minion, a fast Constraint Solver

Chris Jefferson

Uversity of Oxford, Oford, UK, Chris.Jefferson@comlab.ox.ac.uk

This talk will deal with many of the practical matters of implementing am efficient constraint solver using existing algorithms and methods. SAT solvers have historically been able to solve much larger problems than CSP solvers and search thousand of times more nodes per second. This talk will discuss the implementation of constraint solver Minion, which is one of the fastest constraint solvers available and has gone some way to reducing this gap. Most of Minion's speed come from better data structures and careful use templates in C++.

A recent extension to Minion uses the algorithms which make SAT solvers so successful. This talk discusses the difficulties of implementing these algorithms in a general CSP framework.

Chris Jefferson is a Research Fellow at the University of Oxford, where he collaborates with Pete Jeavons on using Gröbner Bases to solve CSPs. Chris wrote Minion with Ian Gent and Ian Miguel while a Research Assistant at St Andrews University, and continues to work on it. He is currently registered as a PhD student under the supervision of Alan Frish at the University of York. His PhD thesis is on the topic of representations in CP.

Incremental Filtering Algorithms for Precedence and Dependency Constraints

Roman Barták[1] and Ondřej Čepek[1 2]

[1] Charles University in Prague,
Faculty of Mathematics and Physics,
Malostranské nám. 25, 118 00 Praha 1, Czech Republic,
{roman.bartak,ondrej.cepek}@mff.cuni.cz
[2] Institute of Finance and Administration, Estonská 500, 101 00 Praha 10, Czech Republic

Abstract. Precedence constraints play a crucial role in planning and scheduling problems. Many real-life problems also include dependency constraints expressing logical relations between the activities – for example, an activity requires the presence of another activity in the plan. For such problems a typical objective is a maximization of the number of activities satisfying the precedence and dependency constraints. In the paper we propose new incremental filtering rules integrating propagation through both precedence and dependency constraints. We also propose a new filtering rule using the information about the requested number of activities in the plan. We demonstrate efficiency of the proposed rules on the log-based reconciliation problems and min-cutset problems.

1 Introduction

Planning and scheduling belong among the most successful application areas of constraint satisfaction. Solving these problems depends on efficient handling of temporal and resource constraints. Temporal networks play an important role in planning but they are not used as frequently in scheduling where resource restrictions traditionally play a stronger role. This is reflected in scheduling global constraints, where techniques like edge-finding or not-first/not-last combine restrictions on time windows with a limited capacity of the resource. Recently, a new category of propagation techniques combining information about relative position of activities with capacity of resources appeared [6, 7]. Also techniques combining information about precedence relations and time windows have been proposed [1, 6]. We believe that integration of temporal networks with reasoning on resources [7, 8] will play even more important role as planning and scheduling technologies are becoming closer. In addition to precedence relations, many problems include dependency constraints between the activities. This is typical for planning problems where an existence of an activity in the plan depends on the presence of other activities in the plan. Similar constraints appear in oversubscribed problems where the task is to schedule the maximal number of activities

and inclusion of an activity in the schedule may require presence of other activities in the schedule. Such problems can be modelled using optional activities; the system then decides about validity or invalidity of optional activities respecting all the constraints. This is similar to solving over-constrained problems with the goal to maximize the number of satisfied constraints. The important difference is that constraints are grouped in our problem and all constraints in the group must be satisfied together (for example, the group corresponds to constraints related to a single activity).

In this paper we focus on modelling precedence constraints using a precedence graph and on integrating reasoning on dependency constraints in this model. In particular, we propose a new constraint-based model of the precedence graph with optional activities and we design new filtering rules for incremental maintenance of transitive closure for such precedence graphs. In the filtering we also use information about dependency constraints. This is, we believe, the first time when filtering through precedence and dependency constraints is realised in an integrated way. We also propose new objective-based filtering for these problems. This filtering uses information about the requested number of valid activities in the final plan.

The paper is organized as follows. We will first introduce the problem more formally and survey the existing solving approaches. Then we will describe the filtering rules for maintaining a transitive closure of the precedence graph with optional activities. We will also show their theoretical time complexity and prove their soundness. After that, we will describe the propagation rule doing filtering based on requested number of valid activities. We will conclude the paper with experimental comparison of our approach with the existing model.

2 Problem Description and Related Works

In this paper we address the problem of modelling precedence constraints between the activities in over-subscribed problems. We do not assume activity duration or time windows here and the activities can run in parallel, if allowed by the precedence constraints. The *precedence constraint* $A \prec B$ specifies that activity A must be before activity B in the schedule. To model over-subscribed problems, we assume *optional activities*. An optional activity has one of the following three statuses. If the activity is not yet known to be or not to be included in the schedule then it is called *undecided*. If the activity is included in the schedule then it is called *valid*. If the activity is known not to be included in the schedule then it is called *invalid*. We also assume dependency constraints between the activities. The *dependency constraint* $A \Rightarrow B$ specifies that if activity A is valid then activity B must be valid as well. In other words, if activity A is included in the schedule then activity B must be included as well. This is one of the dependency constraints proposed in the general model for manufacturing scheduling [9]. The task is to decide about (in)validity of the undecided activities and to find a set of valid activities satisfying the precedence and dependency constraints. The precedence constraints are satisfied if there is no cycle between

valid activities. Usually, the problem is formulated as an optimization problem, where the task is to find a feasible solution in the above sense that maximizes the number of valid activities.

Though our motivation is mainly in the area of scheduling, the above problem is also known as a log-based reconciliation problem in databases. The straightforward constraint model for this problem has been proposed in [2]. The model uses n integer variables $p_1, \ldots, p_n$ which give the positions of activities in the schedule (n is the number of activities). The initial domain of these variables is $1, \ldots, n$. There are also n Boolean (0/1) variables $a_1, \ldots, a_n$ describing whether the activity is valid (1) or invalid (0). The precedence constraint between activities i and j is then described using the formula:

$$(a_i \wedge a_j) \Rightarrow (p_i < p_j) \text{ or equivalently } (a_i * a_j * p_i < p_j).$$

The dependency constraint between activities i and j can be formulated as:

$$a_i \Rightarrow a_j.$$

The solver uses standard constraint propagation over above constraints combined with enumeration of the Boolean variables a_i's. The paper [2] also proves that the log-based reconciliation problem is NP-hard – if there are no dependency constraints then the problem reduces to the problem of finding the smallest cutset in a directed graph (that is, the smallest set of vertices whose removal makes the input graph acyclic) [4].

In [3] an improvement of the above precedence constraint has been proposed using the reasoning on graph properties. Namely a global cutset constraint has been proposed that uses graph contraction techniques to infer some simple Boolean constraints. Still, this model assumes the dependency constraints separately; in particular the constraints are modelled in the above implication form.

The paper [5] also studies the log-based reconciliation problem, but rather than proposing a new filtering algorithm, a decomposition technique is used. The technique is again motivated by the minimal cutset problem and the dependency constraints are handled separately. Moreover, as opposed to the above described models, the technique from [5] is incomplete – meaning that it does not guarantee optimality.

Our approach is different from the above techniques by integrating reasoning on both precedence and dependency constraints. We cannot use the contraction techniques from [3], because our aim is to eventually use the designed filtering algorithm in a scheduler where the precedence graph is used by other constraints like the constraint that integrates reasoning on precedence relations and time windows. Such a constraint assumes activity durations and time windows so it can deduce new precedences using time windows and, vice versa, it can shrink time windows using information about precedences. For details see [1].

3 Filtering Rules for Precedence and Dependency Constraints

Precedence relations among activities define a precedence graph that is an acyclic directed graph where nodes correspond to activities and there is an arc from A to B if $A \prec B$. If access to all predecessors and successors of a given activity is frequently requested, like in [1, 6], then it is more efficient to keep a transitive closure of the graph where this information is available in $O(1)$ time, rather than to look for predecessors/successors on demand. We propose the following definition of transitive closure of the precedence graph with optional activities.

Definition 1. *We say that a precedence graph G with optional activities is transitively closed if for any two arcs A to B and B to C such that B is a valid activity and A and C are either valid or undecided activities there is also an arc A to C in G.*

It is easy to prove that if there is a path from A to B such that A and B are either valid or undecided and all inner nodes in the path are valid then there is also an arc from A to B in a transitively closed graph (by induction on the path length). Hence, if no optional activity is used (all activities are valid) then Definition 1 corresponds to a standard definition of the transitive closure.

We propose to realise reasoning on precedence relations using constraint satisfaction technology. This allows integration of our model with other constraint reasoning techniques, namely the one proposed in [1]. This integration requires the model to provide full information about precedence relations to all other constraints. We index each activity by a unique number from the set $1, \ldots, n$, where n is the number of activities. For each activity we use a 0/1 variable Valid indicating whether the activity is valid (1) or invalid (0). If the activity is undecided (not yet known to be valid or invalid) then the domain of Valid is $\{0, 1\}$. The precedence graph is encoded in two sets attached to each activity. CanBeBefore(A) is a set of indices of activities that can be before activity A. CanBeAfter(A) is a set of indices of activities that can be after activity A. For simplicity reasons we will write A instead of the index of A. To simplify description of the propagation rules we also define for every activity A the following derived sets:

$$\text{MustBeAfter}(A) = \text{CanBeAfter}(A) \setminus \text{CanBeBefore}(A)$$
$$\text{MustBeBefore}(A) = \text{CanBeBefore}(A) \setminus \text{CanBeAfter}(A)$$
$$\text{Unknown}(A) = \text{CanBeBefore}(A) \cap \text{CanBeAfter}(A).$$

MustBeAfter(A) and MustBeBefore(A) are sets of those activities that must be after and before the given activity A respectively. Unknown(A) is a set of activities that are not yet known to be before or after activity A (Figure 1).

Note on representation. The main reason for using sets to model the precedence graph is their possible representation as domains of variables in constraint satisfaction packages. Recall that domains of variables can only shrink as problem solving proceeds. The sets in our model are also shrinking as new

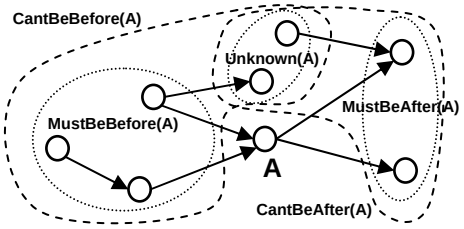


Fig. 1. Representation of the precedence graph

arcs $\prec$ are added to the precedence graph. Hence a special data structure is not necessary to describe the precedence graph in constraint satisfaction packages. Moreover, these packages usually provide tools to manipulate the domains, for example membership and deletion operations. In the subsequent complexity analysis, we will assume that these operations require time O(1), which can be realised for example by using a bitmap representation of the sets. Note finally, that empty domain implies inconsistency in constraint satisfaction that may be a problem for the very first and very last activity which has no predecessors and successors respectively. To solve the problem we can simply leave activity A in both sets CanBeAfter(A) and CanBeBefore(A). Then no domain of CanBeBefore and CanBeAfter will ever be empty but we can detect inconsistency via the empty domain of Valid variables.

The goal of propagation rules is to remove inconsistent values from the above described sets – this is called domain filtering in constraint satisfaction. In the first stage, we will focus on making a transitive closure of the precedence graph according to Definition 1. Note that the transitive closure of the precedence graph also simplifies detection of inconsistency of the graph. The precedence graph is inconsistent if there is a cycle of valid activities. In a transitively closed graph, each such cycle can be detected by finding two valid activities A and B such that $A \prec B$ and $B \prec A$. Our propagation rules prevent cycles by making invalid the last undecided activity in each cycle. This propagation is realised by using an exclusion constraint. As soon as there is a cycle $A \prec B$ and $B \prec A$ detected, the following exclusion constraint can be posted:

$$\text{Valid}(A) = 0 \vee \text{Valid}(B) = 0.$$

This constraint ensures that each cycle is broken by making at least one activity in the cycle invalid. Instead of posting the constraint directly to the constraint solver, we propose keeping the set Ex of exclusions. The above exclusion constraint is modelled as a set $\{A, B\} \in Ex$. Now, the propagation of exclusions is realised explicitly – if activity A becomes valid then all activities C such that $\{A, C\} \in Ex$ are made invalid (see rule /1/ below).

In addition to precedence constraints, there are also dependency constraints in the problem. The dependency $A \Rightarrow B$ can be easily described using the constraint:

$$(\text{Valid}(A) = 1) \Rightarrow (\text{Valid}(B) = 1).$$

Similarly to exclusions, we propose to keep the set Dep of dependencies instead of posting the above constraints, and to realise the propagation of dependencies explicitly. In particular, if activity A becomes valid then all activities C such that $(A \Rightarrow C) \in Dep$ are made valid. Reversely, if activity A becomes invalid then all activities C such that $(C \Rightarrow A) \in Dep$ are made invalid (see rule /1/ below).

Keeping the exclusions and dependencies explicitly has the advantage of stronger filtering (Table 1). In particular, if exclusion $\{A, B\}$ is to be added to Ex and there is a dependency $(A \Rightarrow B) \in Dep$ then we can make activity A invalid (and the exclusion is resolved so it does not need to be kept in Ex). Note that A must be invalid in any solution satisfying the above exclusion and dependency constraints which justifies the proposed filtering. Moreover, if $\{A, B\}$ is added to Ex and there is an activity C such that $(C \Rightarrow A) \in Dep$ and $(C \Rightarrow B) \in Dep$ then we can make activity C invalid. Again, C must be invalid in any solution satisfying the above exclusion and dependency constraints which justifies the proposed filtering. This reasoning is used in both filtering rules /1/ and /2/ below. Keeping explicit dependencies and exclusions will also help us later to deduce a better estimate of the number of valid/invalid activities that is used in cost-based filtering (see rule /3/ below).

Table 1. Reasoning on exclusions and dependencies.

Condition	**Effect**
$\{A, B\} \in Ex \wedge (A \Rightarrow B) \in Dep$	$\text{Valid}(A) = 0$
$\{A, B\} \in Ex \wedge (C \Rightarrow A), (C \Rightarrow B) \in Dep$	$\text{Valid}(C) = 0$

The above described reasoning is realised by the following propagation rule that is invoked when the validity status of the activity becomes known. "Valid(A) is instantiated" is its trigger. The part after $\longrightarrow$ is a propagator describing pruning of domains. "exit" means that the constraint represented by the propagation rule is entailed so the propagator is not further invoked (its invocation does not cause further domain pruning). We will use the same notation in all rules.

```
Valid(A) is instantiated -->                                    /1/
if Valid(A) = 0 then
    for each C s.t. (C=>A) in Dep do Valid(C):= 0
    Ex := Ex \ {{A,X} | X is an activity}
else                                              // Valid(A)=1
    for each C s.t. (A=>C) in Dep do Valid(C):= 1
    for each C s.t. {A,C} in Ex do Valid(C):= 0
    for each B in MustBeBefore(A) s.t. Valid(B) <> 0 do
      for each C in MustBeAfter(A) \ MustBeAfter(B)
          s.t. Valid(C) <> 0 do
```

```
        CanBeAfter(C):= CanBeAfter(C) \ {B}  //add arc from B to C
        CanBeBefore(B):= CanBeBefore(B) \ {C}
        if C not in CanBeAfter(B) then          // break the cycle
          if (C=>B) in Dep then Valid(C):= 0
          else if (B=>C) in Dep then Valid(B):= 0
               else Ex:= Ex + {{B,C}}         // add {B,C} into Ex
                    for each X s.t. (X=>B) in Dep
                        and (X=>C) in Dep do Valid(X):= 0
exit
```

Note that rule /1/ maintains symmetry of sets modelling the precedence graph for all valid and undecided activities because the domains are pruned symmetrically in pairs. We shall show now, that if the entire precedence graph is known in advance (no arcs are added during the solving procedure), then rule /1/ is sufficient for keeping the transitive closure according to Definition 1.

Proposition 1. *Let $A_0, A_1, \ldots, A_m$ be a path in the precedence graph such that Valid(A_j) = 1 for all $1 \leq j \leq m-1$ and Valid(A_0) $\neq$ 0 and Valid(A_m) $\neq$ 0 (that is, the endpoints of the path are not invalid, and all inner points of the path are valid). Then $A_0 \prec A_m$, that is, $A_0 \notin$ CanBeAfter(A_m) and $A_m \notin$ CanBeBefore(A_0).*

Proof. We shall proceed by induction on m. The base case $m = 1$ is trivially true after initialisation (we assume that for every arc (X, Y) in the precedence graph X is removed from CanBeBefore(Y) and Y is removed from CanBeAfter(X) in the initialisation phase). For the induction step let us assume that the statement of the lemma holds for all paths (satisfying the assumptions of the lemma) of length at most $m-1$. Let $1 \leq j \leq m-1$ be an index such that Valid(A_j):= 1 was set last among all inner points $A_1, \ldots, A_{m-1}$ on the path. By the induction hypothesis we get

- $A_0 \notin$ CanBeAfter(A_j) and $A_j \notin$ CanBeBefore(A_0) using the path $A_0, \ldots, A_j$
- $A_j \notin$ CanBeAfter(A_m) and $A_m \notin$ CanBeBefore(A_j) using $A_j, \ldots, A_m$

We shall distinguish two cases. If $A_m \in$ MustBeAfter(A_0) (and by symmetry also $A_0 \in$ MustBeBefore(A_m)) then by the definition (of the MustBeBefore sets) we get $A_m \notin$ CanBeBefore(A_0) and $A_0 \notin$ CanBeAfter(A_m) and so the claim is true trivially. Thus, let us in the remainder of the proof assume that $A_m \notin$ MustBeAfter(A_0).

Now let us show that $A_0 \in$ CanBeBefore(A_j) must hold, which in turn (together with $A_0 \notin$ CanBeAfter(A_j)) implies $A_0 \in$ MustBeBefore(A_j). Let us assume by contradiction that $A_0 \notin$ CanBeBefore(A_j). However, at the time when both $A_0 \notin$ CanBeAfter(A_j) and $A_0 \notin$ CanBeBefore(A_j) became true, that is, when the second of these conditions was made satisfied by rule /1/, rule /1/ must have done one the following things

- in case of a dependency constraint between A_0 and A_j, make one of these activities invalid
- in case of no dependency between A_0 and A_j, add the pair (A_0, A_j) into the set Ex of exclusions.

The latter case moreover implies that at the moment when A_j is made valid A_0 is made invalid and hence both cases contradict the assumptions of the lemma.

By a symmetric argument we can prove that $A_m \in \text{MustBeAfter}(A_j)$. Thus when rule /1/ is triggered by setting $\text{Valid}(A_j) := 1$ both $A_0 \in \text{MustBeBefore}(A_j)$ and $A_m \in \text{MustBeAfter}(A_j)$ hold (and $A_m \notin \text{MustBeAfter}(A_0)$ is assumed), and therefore rule /1/ removes A_m from the set $\text{CanBeBefore}(A_0)$ as well as A_0 from the set $\text{CanBeAfter}(A_m)$, which finishes the proof.

Proposition 2. *If implemented properly, the worst-case time complexity of the propagation rule /1/ including all possible recursive calls is $O(n^3)$, where n is the number of activities.*

Proof. If an activity A is made invalid then it is necessary to find all the activities it is dependent on. This can be done in $O(n)$ if the dependency graph as well as its transposed graph (where edges are reversed) is represented by adjacency lists, or if it is represented by an adjacency matrix (one matrix is then sufficient as it is easy to read out both predecessors and successors of A). Also the removal of all exclusion pairs that include A can be done in $O(n)$ if the exclusion pairs are kept in memory as a symmetric $n \times n$ binary matrix. The recursive calls that make other activities invalid thus take $O(n)$ per activity and at most n activities can be made invalid, so the total time for all the recursive calls is $O(n^2)$.

If activity A becomes valid then the detection of dependencies and exclusions (not counting the recursive calls) can be handled in $O(n)$ as above. The recursive calls that make activities invalid take $O(n)$ per activity (as proved above), which gives a total $O(n^2)$ for all such activities. The recursive calls that make activities valid take $O(n^2)$ per activity (as will be proved below), which gives a total $O(n^3)$ for all such activities.

In the two nested loops where new arcs may be added to the graph up to $\Theta(n^2)$ pairs B, C may be inspected for activity A, so this inspection (deciding for which pairs B, C an arc should be added) can take up to $\Theta(n^2)$ for each activity A. This gives the $O(n^2)$ bound used above for each recursive call that makes an activity valid.

It is important to note, that only $O(n^2)$ arcs can be added to the graph during all recursive calls, so the part of the code inside the two nested loops is executed $O(n^2)$ times over all recursive calls (using this bound individually for each activity A which is made valid would yield an overall $O(n^4)$ time bound). The part of the code inside the two nested loops (excluding the recursive calls) takes $O(n)$ time (because of the for loop, all other tests can be performed in $O(1)$ time). Thus we get a total $O(n^3)$ bound for all executions of the code inside the two nested loops (excluding the recursive calls) and a total $O(n^2)$ bound for all recursive calls that make activities invalid.

In some situations arcs may be added to the precedence graph during the solving procedure, either by the user, by the scheduler/planner, or by other filtering algorithms like in [1]. The following rule /2/ updates the precedence graph to keep transitive closure when an arc is added to the precedence graph. We can also use the same rule for the initialisation of precedence graph – the

known arcs are added using this rule rather than added by explicit changes of sets CanBeBefore and CanBeAfter.

```
arc (A,B) is added into G -->                                    /2/
  if A in MustBeBefore(B) then exit  // the arc is already present
  CanBeAfter(B):= CanBeAfter(B) \ {A}
  CanBeBefore(A):= CanBeBefore(A) \ {B}
  if A not in CanBeBefore(B) then              // break the cycle
    if (A=>B) in Dep then Valid(A):= 0
    else if (B=>A) in Dep then Valid(B):= 0
         else Ex:= Ex + {{A,B}}               // add {A,B} into Ex
              for each X s.t. (X=>A) in Dep and (X=>B) in Dep do
                  Valid(X):= 0
  else                                       // transitive closure
    for each C in MustBeBefore(A) \ MustBeBefore(B) do
      if Valid(A)=1 or (C=>A) in Dep or (B=>A) in Dep then
        add arc (C,B) into G
    for each C in MustBeAfter(B) \ MustBeAfter(A)  do
      if Valid(B)=1 or (C=>B) in Dep or (A=>B) in Dep then
        add arc (A,C) into G
  exit
```

Rule /2/ does the following. If a new arc $A \prec B$ is added then we first check whether the arc is not already present in the graph. If it is a new arc then the corresponding sets are updated and a possible cycle is detected (we use the same reasoning as in rule /1/). Finally, if any end point of the arcs is valid, then necessary arcs are added to update the transitive closure according to Definition 1. Moreover, we can add more arcs using information about dependencies – this is useful for earlier detection of possible cycles. Assume that arc $A \prec B$ has been added. If $(B \Rightarrow A) \in Dep$ then all predecessors of A can be connected to B like in the case when A is valid. This is sound because if B becomes valid then A must be valid as well and such arcs will be added anyway and if B becomes invalid then any arc related to B is irrelevant. For the same reason, if there is any predecessor C of A such that $(C \Rightarrow A) \in Dep$ then C can be connected to B. The same reasoning can be applied to successors of B. Note that the propagators for new arcs are evoked after the propagator of the current rule finishes. The following proposition shows that all necessary arcs are added by rule /2/.

Proposition 3. *If the precedence graph G is transitively closed (in the sense specified by Definition 1) and arc $A \prec B$ is added to G then rule /2/ updates the precedence graph G to be transitively closed again.*

Proof. Assume that arc $A \prec B$ is added into G at a moment when arc $B \prec C$ is already present in G. Moreover assume that Valid(A) $\neq 0$, Valid(B)=1, and Valid(C) $\neq 0$. We want to show that $A \prec C$ is in G after rule /2/ is fired by the addition of $A \prec B$. The presence of arc $B \prec C$ implies that $C \in$ MustBeAfter(B) (and by symmetry also $B \in$ MustBeBefore(C)). Now there are two possibilities. Either $C \notin$ MustBeAfter(A) in which case rule /2/ adds the arc $A \prec C$ into G, or $C \in$ MustBeAfter(A) (and by symmetry also $A \in$ MustBeBefore(C)) which means that arc $A \prec C$ was already present in G when arc $A \prec B$ was added.

The case when arc $A \prec B$ is added into G at a moment when arc $C \prec A$ is already present in G and Valid(C) $\neq 0$, Valid(A)=1, Valid(B) $\neq 0$ holds can be handled similarly.

Thus when an arc is added into G, all paths of length two with a valid midpoint which include this new arc are either already spanned by a transitive arc, or the transitive arc is added by rule /2/. In the latter case this may invoke adding more and more arcs. However, this process is obviously finite (cannot cycle) as an arc is added into G only if it is not present in G, and no arc is ever removed from G. More on the time complexity of arc additions follows in Proposition 4.

Therefore, it is easy to see, that when the process of recursive arc additions terminates, the graph G is transitively closed. Indeed, for every path of length two in G with a valid midpoint one of the arcs on the path is added later than the other, and we have already seen that at a moment of such an addition the transitive arc is either already in G or is added by rule /2/ in the next step.

Proposition 4. *The worst-case time complexity of the propagation rule /2/ (adding a new arc) including all recursive calls to rules /1/ and /2/ is $O(n^3)$, where n is the number of activities.*

Proof. Every recursive call to rule /1/ is making some activity invalid, so following the arguments from the proof of Proposition 2, we get that the total time needed to process all such calls is $O(n^2)$. The rest of the code, excluding the recursive calls to itself (to rule /2/), can be executed in $O(n)$ time. To see this it is enough to realize that each test for dependency or exclusion can be handled in $O(1)$ time (if the dependency graph and exclusion pairs are stored using a matrix representation as in the proof of Proposition 2) and therefore each of the three "for each" loops can be handled in $O(n)$ time. Because only $O(n^2)$ arcs can be added over all recursive calls the total $O(n^3)$ time bound follows.

4 Objective-Based Filtering Rule

As we mentioned in the introduction, a typical objective in problems with optional activities is a maximization of the number of valid activities. In constraint solvers, an objective function is usually converted into a constraint with a new variable Obj:

$$Obj = \sum_{A} \text{Valid}(A)$$

where the task is to maximize the value of variable Obj. Then, computing bounds of the objective function and propagating the bounds to problem variables is realised as propagation through this constraint. The above constraint can be realized as it stands, that is, as the sum of variables Valid. In this section, we will present a filtering rule realizing stronger propagation through this constraint. Namely, the rule can deduce better bounds for variable Obj and the rule can also deduce values of some not-yet decided Valid variables.

The proposed filtering rule is based on ideas of constructive disjunction. If activity A is still undecided, we will explore both alternatives, namely Valid(A) = 1 and Valid(A) = 0, to find out their influence on variable Obj and vice versa.

Recall, that variables Valid participate in dependency and exclusion constraints and these constraints are explicitly available via sets Dep and Ex. We will use these constraints to estimate bounds of variable Obj. In particular, if activity A becomes valid (Valid(A)=1) then all undecided activities B such that $(A \Rightarrow B) \in Dep$ must also become valid and, similarly, all undecided activities C such that $\{A, C\} \in Ex$ must become invalid. Symmetrically, if activity A becomes invalid (Valid(A) = 0) then all undecided activities B such that $(B \Rightarrow A) \in Dep$ must also become invalid. Using this deduction and taking into account the numbers of known valid and invalid activities we can estimate bounds for variable Obj. These computed bounds are then used to define better bounds for Obj and vice versa, by comparing the computed bounds with the current bounds of Obj, we can deduce that one of the alternatives is not viable and hence the remaining alternative is forced (unless, both alternatives are not viable and then a failure is detected). For example, if the computed lower bound of Obj for Valid(A) = 1 is greater than the current upper bound of Obj then it is not possible to assign value 1 to Valid(A).

The following filtering rule /3/ realises the above described reasoning. Note, that the filtering rule is not idempotent, that is, the rule is expected to be called again if it proposes a change to any Valid variable or a change to Obj variable. An idempotent version of the rule would be possible but then the rule should integrate propagation rule /1/ and the code would become more complicated (while the pruning power would be the same).

```
bounds of Obj changed or any Valid(X) instantiated -->         /3/
  NumValid := |{X : Valid(X)=1}|
  NumInvalid := |{X : Valid(X)=0}|
  MinObj := lb(Obj)                  // current lower bound of Obj
  MaxObj := ub(Obj)                  // current upper bound of Obj
  LB := max( MinObj, NumValid)
  UB := min( MaxObj, N - NumInvalid)//N = the number of activities
  for each A s.t. Valid(A)={0,1} do
    ValidLB := 1 + NumValid +
               + |{C : Valid(C)={0,1} and (A=>C) in Dep }|
    ValidUB := N - NumInvalid -
               - |{C : Valid(C)={0,1} and {A,C} in Ex }|
    InvalidLB := NumValid
    InvalidUB := N - 1 - NumInvalid - |{C : (C=>A) in Dep }|
    if (ValidLB <= MaxObj) and (ValidUB >= MinObj) then
      if (InvalidLB <= MaxObj) and (InvalidUB >= MinObj) then
        LB := max( LB, min(ValidLB,InvalidLB) )
        UB := min( UB, max(ValidUB,InvalidUB) )
      else
        Valid(A) := 1
        LB := max( LB, ValidLB )
        UB := min( UB, ValidUB) )
    else  if (InvalidLB <= MaxObj) and (InvalidUB >= MinObj) then
            Valid(A) := 0
            LB := max( LB, InvalidLB )
            UB := min( UB, InvalidUB) )
          else fail
  end for
  lb(Obj) := LB
  ub(Obj) := UB
  if NumValid + NumInvalid = N then exit
```

It may seem that the filtering power of rule /3/ can be further strengthen by the following deduction. Irrespectively of assigning 0 or 1 to Valid(A), the

activities from the set $\{C : Valid(C) = \{0,1\} \wedge \{A,C\} \in Ex \wedge (C \Rightarrow A) \in Dep\}$ must become invalid and hence their Valid variables can be set to 0. This is surely true but notice that exclusion $\{X,Y\}$ is added to set Ex by rules /1/ and /2/ only if neither $(X \Rightarrow Y)$ nor $(Y \Rightarrow X)$ are elements of Dep. If this is ensured for any exclusion $\{X,Y\}$ then the above mentioned set will always be empty and hence the deduction based on this set is useless.

5 Experimental Results

To evaluate the practical applicability of the proposed filtering rules, we did some preliminary experiments with log-based reconciliation problems and min-cutset problems. The proposed filtering rules were implemented in SICStus Prolog 3.12.3 using the standard interface for the definition of global constraints. The experiments run under Windows XP Professional on 1.1 GHz Pentium-M processor with 1280 MB RAM.

5.1 Log-based Reconciliation Problems

Though our original motivation to introduce dependency constraints into a precedence graph is in scheduling, log-based reconciliation problems fit perfectly our problem specification where precedence and dependency constraints are combined. We took the problem set from [3] and we compared our approach with the constraint model proposed in [2]. Unfortunately implementation of the cut-set global constraint proposed in [3] was not available to us so we have no direct comparison of runtimes yet. Nevertheless, for two problems, where neither approach found (proved) an optimal solution, our technique improved significantly the lower bound of the objective function. Table 2 presents the results for the CLP model (Original) from [2] and our approach (Precedence). We compare both the runtime (measured in milliseconds) and the number of backtracks to find and prove an optimal solution. We used a limit of 50 minutes to cut the search and we report the best solution found within this time limit (recall that the task is to maximize the number of valid activities).

We have found most of the problems quite easy; frequently the first found solution was the optimal solution. The runtime of our approach for these problems is slightly longer than in the original model; this is due to overhead for building more complex data structures. Nevertheless, the table clearly demonstrates that our approach requires significantly less backtracks to find the solution so the filtering power of the proposed propagation rules pays off there. The table also demonstrates that as soon as the problems are becoming harder, the difference between our approach and the original model is more significant (see problems r200v2 and r800v2). For two problems, r800v1 and r1000v2, neither approach was able to find/prove an optimal solution within the fifty minutes limit. Nevertheless, our propagation rules lead to much better lower bound within a given time limit. The lower bounds for these problems reported in [3] are 771 for r800v1 and 943 for r1000v2, so we also improved the best lower bounds reported there.

Table 2. Computation results on log-based reconciliation benchmarks from [3].

	Original			Precedence		
Bench	**Best**	**Runtime**	**Backtracks**	**Best**	**Runtime**	**Backtracks**
r100v1	98	141	16	98	438	1
r100v2	77	250	85	77	125	3
r100v3	95	156	49	95	313	7
r100v4	100	31	1	100	360	1
r100v5	52	16	3	52	62	5
r200v1	65	63	13	65	78	5
r200v2	191	74657	8015	191	3313	42
r500v1	198	219	3	198	407	5
r500v2	498	1265	32	498	2547	2
r800v1	770	-	-	780	-	-
r800v2	318	3828	327	318	984	10
r1000v1	389	672	3	389	1266	5
r1000v2	935	-	-	957	-	-

To support the above claim that our approach is prevailing over the original model for harder problems, we did a second set of experiments using pseudo-real log-based reconciliation problems proposed in [5]. These problems have a structure typical for real-life problems so the results are more interesting from the practical point than using completely random problems. Table 3 shows the specification of problems used in our experiment – this specification is identical to problems used in [5], though we generated own problems because the problems from [5] were not available. The table also shows the best solutions obtained in our experiments.

Table 3. Pseudo-real log-based reconciliation problems.

Bench	**Activities**	**Precedences**	**Dependencies**	**Original best**	**Precedence best**
p50-3	150	162	175	146	146
p50-4	200	229	211	193	193
p50-5	250	290	346	244	244
p50-6	300	375	377	288	290
p50-7	350	451	468	333	333
p50-8	400	527	593	376	378
p50-9	450	630	680	404	406

We again compared the CLP model proposed in [2] with our filtering rules. We used the time limit of four hours (14 400 000 milliseconds) to cut search, Table 3 reports the best solution found within this time limit. Starting with p50-6, the original model was not able to find/prove the optimal solution within

the time limit while our technique found and proved optimal solutions for all the problems. Figure 2 shows the comparison of runtimes and the number of backtracks for both approaches (we use a logarithmic scale). Our approach requires more than an order of magnitude less backtracks to find the solution and it also requires much less time.

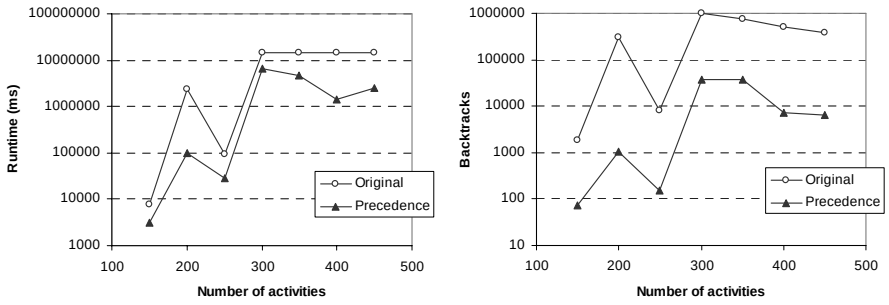


Fig. 2. Computation results on pseudo-real log-based reconciliation problems

5.2 Min-Cutset Problems

We believe that using a precedence graph is better than using absolute positioning in a sequence for modelling problems with precedence relations. Though our approach is proposed for problems with both precedence and dependency constraints, we decided to demonstrate superiority of the precedence graph over absolute positioning on a well known min-cutset problem. The min-cutset problem consists of precedence relations only and the task is to find the largest set of vertices such that the sub-graph induced by these vertices does not contain any cycle (or equivalently to find the smallest set of vertices such that all cycles are broken if these vertices are removed from the graph). This problem is known to be NP-hard [4].

We use the data set from [10] to compare our approach based on the precedence graph with the CLP model from [2] based on absolute positioning in the sequence of activities. All the problems in the data set consist of 50 activities while the number of precedence constraints varies. Table 4 shows the specification of problems used in our experiment and the best solutions obtained. Note that the solutions obtained by our approach (Precedence) are optimal.

Figure 3 shows the comparison of runtimes and the number of backtracks for both approaches (we use a logarithmic scale). Again our approach requires more than an order of magnitude less backtracks and less runtime to find and prove the optimal solution. In fact, with the exception of problems with 50 and 100 precedence constraints, the original CLP model was not able to find the

Table 4. Min-cutset problems.

Bench	Activities	Precedences	Original best	Precedence best
P50-100	50	100	47	47
P50-150	50	150	41	41
P50-200	50	200	35	37
P50-250	50	250	31	33
P50-300	50	300	28	31
P50-500	50	500	21	22
P50-600	50	600	17	19
P50-700	50	700	16	17
P50-800	50	800	16	16
P50-900	50	900	14	14

optimal solution (or to prove optimality) within the time limit of 50 minutes. Note finally, that concerning the runtime we cannot compete with the GRASP heuristic proposed in [10], but this was not our original ambition as we tackle different problems. Moreover, opposite to the GRASP approach our technique is complete and, indeed, for some problems we have found better solutions than reported in [10].

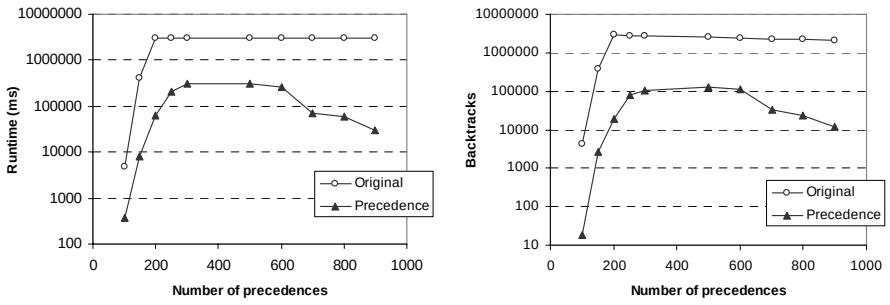


Fig. 3. Computation results on min-cutset problems

6 Conclusions

In the paper we proposed new incremental filtering rules for precedence and dependency constraints. These rules were based on maintaining a transitive closure of the precedence graph with optional activities. Opposite to existing approaches, we proposed to use information about dependency constraints within the filtering rules for the precedence constraints rather than propagating dependencies

separately. We also proposed a filtering rule that uses information about the requested number of valid activities in the precedence graph. This rule belongs to the developing area of cost-based filtering. We experimentally demonstrated that our approach is prevailing over the existing model of precedence and dependency constraints on log-based reconciliation problems and min-cutset problems.

Though we focused on a particular form of dependencies, we believe that our approach is extendable to other dependency constraints, for example, those in [9] where existence of some activity forces removal of another activity. Moreover, with the exception of cost-based filtering, our model can be extended to open precedence graphs where the number of activities is not known in advance.

7 Acknowledgements

The research is supported by the Czech Science Foundation under the contract no. 201/04/1102. We would also like to thank anonymous reviewers for useful comments.

References

1. Barták, R.: Incremental Propagation of Time Windows on Disjunctive Resources. In Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference (FLAIRS 2006), pp. 25–30, 2006.
2. Fages, F.: CLP versus LS on log-based reconciliation problems for nomadic applications. In Proceedings of ERCIM/CompulogNet Workshop on Constraints, Praha, 2001.
3. Fages, F.; Lal, A.: A Constraint Programming Approach to Cutset Problems. Journal Computers and Operations Research. Volume 33, Issue 10, pp. 2852–2865, 2006.
4. Garey, M. R., and Johnson, D. S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, San Francisco, 1979.
5. Hamadi, Y.: Cycle-cut decomposition and log-based reconciliation. In ICAPS Workshop on Connecting Planning Theory with Practice, pp. 30-35, Whistler, 2004.
6. Laborie, P.: Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. Artificial Intelligence 143: pp. 151–188, 2003.
7. Laborie, P.: Resource temporal networks: Definition and complexity. In Proceedings of the 18th International Joint Conference on Artificial Intelligence, pp. 948–953, 2003.
8. Moffitt, M. D.; Peintner, B.; and Pollack, M. E.: Augmenting Disjunctive Temporal Problems with Finite-Domain Constraints. In Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-2005). pp. 1187–1192. AAAI Press, 2005.
9. Nuijten, W.; Bousonville, T.; Focacci, F. Godard, D. Le Pape, C.: MaScLib: Problem description and test bed design. 2003. http://www2.ilog.com/masclib
10. Pardalos, P.M.; Qian, T.; Resende, M.G.: A greedy randomized adaptive search procedure for the feedback vertex set problem. Journal of Combinatorial Optimization, 2: pp. 399–412, 1999.

The Slide Meta-Constraint

Christian Bessiere[1], Emmanuel Hebrard[2], Brahim Hnich[3], Zeynep Kiziltan[4], and Toby Walsh[5]

[1] LIRMM, CNRS/University of Montpellier, France, bessiere@lirmm.fr
[2] 4C and UCC, Cork, Ireland, e.hebrard@4c.ucc.ie
[3] Izmir University of Economics, Izmir, Turkey, brahim.hnich@ieu.edu.tr
[4] University of Bologna, Italy, zkiziltan@deis.unibo.it
[5] NICTA and UNSW, Sydney, Australia, tw@cse.unsw.edu.au

Abstract. We study the Slide meta-constraint. This slides a constraint down one or more sequences of variables. We show that Slide can be used to encode and propagate a wide range of global constraints. We consider a number of extensions including sliding down sequences of set variables, and combining Slide with a global cardinality constraint. We also show how to propagate Slide. Our experiments demonstrate that using Slide to encode constraints can be just as efficient and effective as using specialized propagators.

1 Introduction

In scheduling, rostering and related problems, we often have a sequence of decision variables and a constraint which applies down the sequence. For example, in the car sequencing problem (prob001 in CSPLib), we need to decide the sequence of cars on the production line. We might have a constraint on how often a particular option is met along each sequence (e.g. only 1 out of 3 cars can have the sun-roof option). As a second example, in a nurse rostering problems, we need to decide the sequence of shifts worked by each nurse. We might have a constraint on how many consecutive night shifts any nurse can work. To model such problems, we consider a *meta*-constraint, Slide which ensures that a constraint repeatedly holds down a sequence of variables. This is a special case of the previously introduced CardPath constraint [1]. Although Slide is very simple, we demonstrate that it is surprisingly powerful. In addition, we describe methods to propagate such constraints, which unlike the previous methods proposed for CardPath, can prune all possible values.

The rest of the paper is organised as follows. After presenting the necessary formal background, we introduce the simplest form of the Slide meta-constraint. In later sections, we consider a number of generalizations and give examples of global constraints that can be encoded using these various forms of Slide. These encodings therefore provide a simple and easy way to implement these global constraints. In most cases, propagating our encoding is as efficient and as effective as a specialized propagator.

2 Background

A constraint satisfaction problem consists of a set of variables, each with a finite domain of values, and a set of constraints specifying allowed combinations of values for some subset of variables. We use capital letters for variables (e.g. X, Y and S), and lower case for values (e.g. d and d_i). We consider both finite domain and set variables. A set variable can be represented by its lower bound which contains the definite elements in the set and an upper bound which contains the definite and potential elements. Constraint solvers typically explore partial assignments enforcing a local consistency property. A constraint is *generalized arc consistent* (GAC) iff when a variable is assigned any of the values in its domain, there exist compatible values in the domains of all the other variables. For binary constraints, generalized arc consistency is often called simply arc consistency (AC).

3 Slide constraint

We start with the simplest form of the Slide meta-constraint. If C is a constraint of arity k then we consider the meta-constraint:

$$\textsc{Slide}(C, [X_1, \ldots, X_n])$$

This holds iff $C(X_i, \ldots, X_{i+k-1})$ itself holds for $1 \leq i \leq n-k+1$. That is, we slide the constraint C down the sequence of variables, X_1 to X_n. This simple form of Slide is a special case of the $\textsc{CardPath}(N, [X_1, \ldots, X_n], C)$ meta-constraint, which holds iff C holds N times on the sequence $[X_1, \ldots, X_n]$ [1]. As we shall see, its simple structure will permit us to enforce GAC. Also, we will consider more complex forms of Slide that, for instance, slide over multiple sequences or over set variables.

We illustrate this simple form of Slide with an example of a global constraint used in car sequencing problems. In Section 11, we discuss how to propagate such encodings into Slide. The AmongSeq constraint ensures that values occur with some given frequency. For instance, we might want that no more than 3 out of every sequence of 7 shift variables be a "night shift". More precisely, $\textsc{AmongSeq}(l, u, k, [X_1, \ldots, X_n], v)$ holds iff between l and u values from the ground set v occur in every k sequence of variables [2]. We can decompose this using a Slide; $\textsc{AmongSeq}(l, u, k, [X_1, \ldots, X_n], v)$ can be encoded as $\textsc{Slide}(D^{k,v}_{l,u}, [X_1, \ldots, X_n])$ where $D^{k,v}_{l,u}$ is an instance of the Among constraint [2]. That is, $D^{k,v}_{l,u}(X_i, \ldots, X_{i+k-1})$ holds iff $l \leq \sum_{j=i}^{i+k-1}(X_j \in v) \leq u$.

For example, suppose 2 of every 3 variables along a sequence $X_1 \ldots X_5$ should take the value a, where $X_1 = a$ and $X_2, \ldots, X_5 \in \{a, b\}$. Then we can encode this as $\textsc{Slide}(E, [X_1, X_2, X_3, X_4, X_5])$ where $E(X_i, X_{i+1}, X_{i+2})$ is an instance of the Among constraint that ensures two of its three variables take a. This Slide constraint ensures that the following three constraints hold: $E(X_1, X_2, X_3)$, $E(X_2, X_3, X_4)$ and $E(X_3, X_4, X_5)$. Note that each ternary constraint is GAC. However, enforcing GAC on the Slide constraint will set $X_4 = a$ as there are only two satisfying assignments for X_1 to X_5 and neither of them have $X_4 = b$.

4 Slide over multiple sequences

We often wish to slide a constraint down two or more sequences of variables at once. We therefore consider a more complex form of Slide. If F is a constraint of arity $2k$ then:

$$\text{Slide}(F, [X_1, \ldots, X_n], [Y_1, \ldots, Y_n])$$

holds iff $F(X_i, \ldots, X_{i+k-1}, Y_i, \ldots, Y_{i+k-1})$ itself holds for $1 \leq i \leq n-k+1$. We can slide down three or more sequences of variables in a similar way. Note we could view these as syntactic sugar for a Slide down a single sequence of variables where the different sequences are interleaved (e.g., $[X_1, Y_1, X_2 \ldots, Y_{n-1}, X_n, Y_n]$) , and the constraint is loosened so it trivially holds if it is applied with the wrong offset. This loosening is direct if all X_i and Y_i have distinct domains (the constraint is satisfied for all tuples starting by a value from Y_i). Otherwise an extra sequence of marking variables S_i with a dummy value can be added, and the constraint sliding on $[S_1, X_1, Y_1, S_2, \ldots, Y_n]$ enforces E on the X_i and the Y_i only when its first argument takes the dummy value.

As an example of sliding down multiple sequences of variables, consider the constraint $\text{Regular}(\mathcal{A}, [X_1, \ldots, X_n])$. This ensures that the values taken by a sequence of variables form a string accepted by a deterministic finite automaton [14]. This global constraint is useful in scheduling, rostering and sequencing problems to ensure certain patterns do (or do not) occur over time. It can be used to encode a wide range of other global constraints including: Among [2], Contiguity [13], Lex and Precedence [12].

To encode the Regular constraint with Slide, we introduce finite domain variables, Q_i to record the state of the automaton. We then post the constraint $\text{Slide}(F, [X_1, \ldots, X_{n+1}], [Q_1, \ldots, Q_{n+1}])$ where X_{n+1} is a "dummy" variable, Q_1 is assigned to the starting state of the automaton, Q_{n+1} is restricted to any of the accepting states, and $F(X_i, X_{i+1}, Q_i, Q_{i+1})$ holds iff $Q_{i+1} = \delta(X_i, Q_i)$ where δ is the transition function of the finite automaton. Note that F is independent of its second argument so is effectively ternary. Since the automaton is deterministic, F is also functional on X_i and Q_i. Enforcing GAC on this encoding takes $O(ndQ)$ time where d is the number of values for the X_i and Q is the number of states of the automaton. This is identical to the specialized propagator for Regular proposed in [14].

One advantage of our encoding of the Regular constraint it that it gives us explicit access to the states of the automaton. Consider, for example, a rostering problem where workers are allowed to work for up to three consecutive shifts and then must take a break. This can be specified with a simple Regular language constraint. Suppose now we want to minimize the number of times a worker has to work for three consecutive shifts. To model this, we can post an Among constraint on the state variables to count the number of times we visit the state representing three consecutive shifts, and minimize the value taken by this variable.

5 Slide with counters

We often wish to slide a constraint down one or more sequences of variables computing some count. We can use Slide to encode such constraints by incrementally computing the count in an additional sequence of variables. As an example, consider the meta-constraint CardPath$(C, [X_1, \ldots, X_n], N)$ where C is any constraint of arity k [1]. This holds iff $C(X_i, \ldots, X_{i+k-1})$ holds N times down the sequence of variables. As we observed earlier, Slide is a special case of CardPath where $N = n - k + 1$. However, as we show here, CardPath can itself be encoded into a Slide constraint using a sequence of counters.

The CardPath constraint is useful in rostering problems. For example, we can count the number of changes in the type of shift given to a single worker using CardPath$(\neq, [X_1, \ldots, X_n], N)$. CardPath can also be used to model a range of Boolean connectives since $N \geq 1$ gives disjunction, $N = 1$ gives exclusive or, and $N = 0$ gives negation. For notational simplicity, we will consider the case when $k = 2$ and C is a binary constraint. The generalization to other k is straightforward. We introduce a sequence of integer variables M_i in which to accumulate the count. More precisely we decompose a CardPath constraint on a binary constraint C into Slide$(G, [X_1, \ldots, X_{n+1}], [M_1, \ldots, M_{n+1}])$ where X_{n+1} is a "dummy" variable, $M_1 = 0$, $M_{n+1} = N$, and $G(X_i, X_{i+1}, M_i, M_{i+1})$ holds iff $C(X_i, X_{i+1})$ implies $M_{i+1} = M_i + 1$ else $M_{i+1} = M_i$.

6 Slide over sets

In some cases, we want to slide a constraint down one or more sequences of set variables. We therefore consider Slide meta-constraints which involve set variables. We give an example useful for breaking symmetry in problems like the social golfer's problem (prob010 in CSPLib).

Law and Lee have introduced the idea of value precedence for breaking the symmetry of indistinguishable values [12]. They proposed a global constraint to deal with set variables containing indistinguishable values. More precisely, Precedence$([v_1, \ldots, v_m], [S_1, \ldots, S_n])$ holds iff $\min\{i \mid (v_j \in S_i \wedge v_k \notin S_i) \vee i = n+1\} \leq \min\{i \mid (v_k \in S_i \wedge v_j \notin S_i) \vee i = n+2\}$ for all $1 \leq j < k \leq m$, where $[v_1, \ldots, v_m]$ are the indistinguishable values. That is, the first time we distinguish v_j and v_k (because both don't occur in a given set variable), we have v_j occurring and not v_k. For example, the following sequence of sets satisfies value precedence: $\{1, 2, 3\}$, $\{4, 5, 6\}$, $\{1, 4, 5\}$. The first two sets distinguish apart 1, 2, and 3 from 4, 5 and 6, whilst the third set distinguishes apart 1 from 2 and 3, and 4 and 5 from 6. However, this next sequence of sets does not satisfy value precedence as we distinguish apart 3 before 2: $\{1, 2, 3\}$, $\{1, 3, 4\}$.

We can encode such a symmetry breaking constraint using a Slide. For simplicity, we consider just two indistinguishable values, v_j and v_k. However, we can deal with multiple values using Slide but it is notationally more messy. We introduce 0/1 variables, B_i to record whether the two values have been distinguished apart so far. We then post Slide$(H, [S_1, \ldots, S_{n+1}], [B_1, \ldots, B_{n+1}])$

where S_{n+1} is a "dummy" set variable, $B_1 = 0$ and $H(S_i, S_{i+1}, B_i, B_{i+1})$ holds iff $B_i = B_{i+1} = 1$, or $B_i = B_{i+1} = 0$ and $v_j \in S_i$ and $v_k \in S_i$, or $B_i = B_{i+1} = 0$ and $v_j \notin S_i$ and $v_k \notin S_i$, or $B_i = 0$ and $B_{i+1} = 1$ and $v_j \in S_i$ and $v_k \notin S_i$. Note that H is again independent of its second argument.

7 Other examples of Slide

There are many other examples of global constraints which can be encoded using Slide. For example, we can encode the lexicographical ordering constraint Lex using Slide. Lex holds iff a vector of variables $[X_1..X_n]$ is lexicographically smaller than a vector $[Y_1..Y_n]$. We introduce a sequence of $n+1$ Boolean variables B_i to indicate if the vectors have been ordered yet at position i (B_0 is set to 0). We slide the constraint $U(X_i, Y_i, B_{i-1}, B_i)$ which holds iff $(B_{i-1} = B_i = 0 \wedge X_i = Y_i)$ or $(B_{i-1} = 0 \wedge B_i = 1 \wedge X_i < Y_i)$ or $(B_{i-1} = B_i = 1)$. This gives us a linear time propagator as efficient and incremental as the specialized algorithm in [8]. As a second example, we can encode many types of channeling constraints using Slide like the Domain constraint [15], the LinkSet2Booleans constraint [4] and the Element constraint [10]. As a final example, we can encode "optimization" constraints like the soft form of the Regular constraint which measures the Hamming or edit distance to a regular string [19].

There are, however, global constraints that can be encoded using Slide which do not give us as efficient and effective propagators as specialized algorithms. As an example, the AllDifferent constraint can easily be specified using Slide (we just need a Slide which accumulates in a sequence of set variables the values used so far and ensure the final set variable has cardinality n). However, this is not as effective as a specialized flow-based propagator [17]. There are also global constraints like the inter-distance constraint [18] which Slide provides neither a good propagator nor it seems even a simple encoding.

8 Slide with Gcc

We often have a constraint on the values which should occur across the whole sequence. For example, in car sequencing problems, to ensure we build the correct orders, we have a constraint on the total number of occurrences of each value along the sequence. The global sequencing constraint (Gsc) [16] augments an AmongSeq constraint with a global cardinality constraint (Gcc) on the total number of occurrence of different values. More precisely, $\textsc{Gsc}([X_1, \ldots, X_n], a, b, q, v, [l_1, \ldots, l_m], [u_1, \ldots, u_m])$ is satisfied iff for each $i \in [1..m]$, $l_i \leq |\{j \mid X_j = i\}| \leq u_i$ (that is, the value i occurs between l_i and u_i times in total), and for each $k \in [1..n]$, $a \leq |\{j \mid X_j \in v \;\&\; k \leq j \leq k+q-1\}| \leq b$ (that is, values in v occur between a and b times in each sequence of q consecutive variables). In [16], an algorithm that partially propagates Gsc is proposed. Another way to propagate Gsc is to decompose it into a separate Slide and Gcc. Enforcing GAC on such

a decomposition hinders propagation and is incomparable to the pruning of the algorithm in [16].

We prove here that adding cardinality constraints to a Slide makes propagation intractable. In fact, we shall prove that enforcing GAC on the Gsc constraint is intractable. As Gsc can easily be encoded into a Slide and a Gcc, the result follows immediately. This also settles the open question of the complexity of propagating Gsc.

Theorem 1 *Enforcing GAC on* Gsc *is NP-hard.*

Proof: We reduce the 1in3-SAT problem on positive clauses to finding support for a particular Gsc. Consider a 1in3-SAT problem in N variables and M positive clauses in which the Boolean variables are numbered from 1 to N. We let $n = 2NM$. The basic idea of the reduction is that each consecutive block of $2N$ CSP variables represents a given truth assignment. The even numbered CSP variables will represent the truth assignment. The odd numbered CSP variables will essentially be "junk" and serve only to ensure we have exactly N non-zero values in each $2N$ block. That is, X_{2jN+2i} will be non-zero iff the ith Boolean variable is true in the given truth assignment. To achieve this, we set $a = b = N$, $q = 2N$ and $v = \{1, \ldots, M+1\}$. Each $2N$ block thus contains the same pattern of N zeroes and N non-zeroes.

The jth block of $2N$ CSP variables will ensure that the jth clause is satisfied by the truth assignment. That is, just one of its positive literals is true. Suppose the jth clause is $r \vee s \vee t$. Then we let X_{2jN+2r}, X_{2jN+2s} and X_{2jN+2t} have the domain $\{0, j+1\}$. All other CSP variables in the block have 0/1 domains. We set $l_{j+1} = u_{j+1} = 1$ to ensure only one of X_{2jN+2r}, X_{2jN+2s} and X_{2jN+2t} is set to $j+1$. Finally, we let $l_0 = u_0 = NM$, and $l_1 = u_1 = NM - M$. An assignment for X_i then corresponds to a satisfying assignment for the original 1in3-SAT problem. Deciding if the Gsc has support is thus NP-hard. □

The proof can be generalized to show that enforcing bounds consistency on such a constraint is NP-hard, as well as to the case where $u_i = 1$ (in other words, when we have an AmongSeq with an AllDifferent constraint).

9 Circular Slide

Another generalization of Slide is when we wish to ensure that a constraint applies at any point round a cycle of variables. Such a meta-constraint is useful in scheduling and rostering problems where we need to ensure the schedule can be repeated, say, every two weeks. If C is a constraint of arity k then we consider the meta-constraint:

$$\textsc{Slide}_{\mathrm{O}}(C, [X_1, \ldots, X_n])$$

This holds iff $C(X_i, \ldots, X_{1+(i+k-1 \bmod n)})$ itself holds for $1 \leq i \leq n$.

As an example, we encode the circular form of the Stretch constraint [9]. First, let us consider the non-cyclic Stretch constraint. In a Stretch constraint, we are given a sequence of shift variables X_1 to X_n, each having domain

a set of shift types τ, a set $\pi \subset \tau \times \tau$ of ordered pairs (called patterns), and the function $shortest(t)$ (resp. $longest(t)$) denoting the minimum (resp. maximum) length of any stretch of type t. Stretch$([X_1, \ldots, X_n])$ holds iff (1) each stretch (i.e., a sequence of variables having the same type) of type t is feasible, i.e., each stretch has length between $shortest(t)$ and $longest(t)$; and (2) each pair of consecutive types of stretches is in π. We can encode Stretch as Slide$(C, [X_1, \ldots, X_n], [Q_1, \ldots, Q_n])$ where $Q_1 = 1$ and $C[X_i, X_{i+1}, Q_i, Q_{i+1})$ holds iff (1) $X_i = X_{i+1}$, $Q_{i+1} = 1 + Q_i$, and $Q_{i+1} \leq longest(X_i)$; or (2) $X_i \neq X_{i+1}$, $\langle X_i, X_{i+1} \rangle \in \pi$, $Q_i \geq shortest(X_i)$ and $Q_{i+1} = 1$. Circular Stretch is simply Slide$_{\mathrm{O}}(C, [X_1, \ldots, X_n], [Q_1, \ldots, Q_n])$ in which we do not force $Q_1 = 1$.

10 A Slide algebra

When we negate a Slide, we get a disjunctive sequence of constraints. We therefore propose the SlideOr meta-constraint. More precisely, if C is a constraint of arity k then:

$$\text{SlideOr}(C, [X_1, \ldots, X_n])$$

holds iff one or more of $C(X_i, \ldots, X_{i+k-1})$ holds. We can also slide down multiple sequences simultaneously as with Slide. SlideOr can itself be encoded using Slide since SlideOr$(C, [X_1, \ldots, X_n])$ is equivalent to CardPath$(C, [X_1, \ldots, X_n], N)$ where $1 \leq N \leq n$, and CardPath can itself be encoded into Slide. One application of the SlideOr meta-constraint is to encode the global not all equals constraint, NotAllEqual$([X_1, \ldots, X_n])$. This holds iff $X_i \neq X_j$ for some $1 < j \leq n$.

In fact, we can build up more complex sliding constraints using Boolean operators. We can simplify such complex constraint expressions by exploiting associativity, commutativity and De Morgan's identities. For example:

$$\begin{aligned}
\neg\text{Slide}(C_1, [X_1, ., X_n]) &\leftrightarrow \text{SlideOr}(\neg C_1, [X_1, ., X_n]) \\
\neg\text{SlideOr}(C_1, [X_1, ., X_n]) &\leftrightarrow \text{Slide}(\neg C_1, [X_1, ., X_n]) \\
\text{Slide}(C_1, [X_1, ., X_n]) &\wedge \\
\wedge\text{Slide}(C_2, [X_1, ., X_n]) &\leftrightarrow \text{Slide}(C_1 \wedge C_2, [X_1, ., X_n])
\end{aligned}$$

11 Propagating Slide

A meta-constraint like Slide is only really useful if we can propagate it easily. The simplest possible way to propagate Slide$(C, [X_1, \ldots, X_n])$ is to decompose it into the sequence of constraints, $C(X_i, \ldots, X_{i+k-1})$ for $1 \leq i \leq n-k+1$ and let the constraint solver propagate the decomposition. Surprisingly, this is enough to achieve GAC in many cases. For example, we can achieve GAC this way using our Slide encoding of the Regular constraint. In such a case, propagating the decomposition is also an efficient means to achieve GAC. Only those constraints in the decomposition which have variables whose domains change need wake up. It thus provides an efficient incremental propagator for Slide.

Theorem 2 *Enforcing GAC on the decomposition of* Slide *achieves GAC on* Slide *if the slid constraints overlap on just one variable.*

Proof: The constraint graph of the decomposition is Berge-acyclic [7]. □

Similarly, enforcing GAC on the decomposition achieves GAC on Slide if the constraints being slid are monotone. A constraint C is monotone iff there exists a total ordering $\prec$ of the domain values such that for any two values v, w, if $v \prec w$ then v is substitutable to w in any support for C. For instance, the constraints Among and Sum are monotone if either no upper bound, or no lower bound is given.

Theorem 3 *Enforcing GAC on the decomposition of* Slide *achieves GAC on* Slide *if the slid constraints are monotone.*

Proof: For an arbitrary value $v \in D(X)$, we show that if every constraint is GAC, then we can build a support for (X, v) on Slide. For any variable other than X, we choose the first value in the total order, that is, the value which can be substituted to any other value in the same domain. The tuple built this way satisfies all the constraints being slid since we know that there exists a support for each (they are GAC), and the values we chose can be substituted to this support. □

On the other hand, in the general case, if constraints overlap on more than one variable (e.g. in the Slide encoding of AmongSeq). we need to do more work to achieve GAC. For reasons of space, we only have room here to outline how to propagate Slide in these circumstances. We consider two cases. If the arity of the constraint being slid is fixed, then we show that propagation is polynomial. On the other hand, if the arity of the constraint is not fixed, then propagation is intractable even if the constraint being slid is itself polynomial to propagate. In other words, enforcing GAC on Slide is fixed parameter tractable.

When the arity of the constraint being slid is fixed, we can use dynamic programming to compute support along the Slide. This is similar to the propagators for the Regular and Stretch constraints [14, 9]. Alternatively, we can use a dual encoding [7]. We sketch how such a dual encoding works. We introduce dual variables to contain the supports for each constraint in the decomposition of the Slide. Between consecutive dual variables, we have binary compatability constraints to ensure the supports agree on overlapping dual variables. As the constraint graph of the dual variables is Berge-acyclic, enforcing AC on these dual variables, achieves GAC on the original Slide constraint. Using such a dual encoding, Slide can be easily added to any constraint solver. In general, enforcing GAC on a Slide constraint takes in $O(nd^{k+1})$ time and $O(nd^k)$ space where k is the overlap between successive constraints in the decomposition of Slide and d is the maximum domain size.

When the arity of the constraint being slid is not fixed, enforcing GAC is NP-hard.

Theorem 4 *Enforcing GAC on* Slide$(C, [X_1, \ldots, X_n])$ *is NP-hard when the arity of C is not fixed even if enforcing GAC on C is itself polynomial.*

Proof: A simple reduction from 3-SAT in N variables and M clauses. We let $n = (N+1)M$. Each block of $N+1$ variables represents a clause and a truth assignment. We will have $X_{j(N+1)+i+1} = 1$ iff the Boolean variable x_i is true $(1 \leq i \leq N)$. If the $k+1$th clause is $x_a \vee \neg x_b \vee x_c$ then $X_{k(N+1)} \in \{x_a, \neg x_b, x_c\}$. Finally $C(X_i, \ldots, X_{i+N+1})$ holds iff $X_i \notin \{0,1\}$ and $X_{i+N+1} = X_i$, or $X_i = x_d$ and $X_{i+d} = 1$, or $X_i = \neg x_d$ and $X_{i+d} = 0$. An assignment for X_i then corresponds to a satisfying assignment for the original 3-SAT problem. □

12 Experiments

We wish to show that encoding problems using the Slide meta constraint can be just as efficient and effective as using specialized propagators. Experiments are done using ILOG Solver 5.3.

12.1 Balanced Incomplete Block Design Generation

Balanced Incomplete Block Design (BIBD) generation is a standard combinatorial problem from design theory with applications in cryptography and experimental design. A BIBD is specified by a binary matrix of b columns and v rows, with exactly r ones per row, k ones per column, and a scalar product of λ between any pair of distinct rows. Our model consists of sum constraints on each row and each column as well as the scalar product constraint between every pair of rows. Any pair of rows and any pair of columns of a solution can be exchanged to obtain another symmetrical solution. We therefore impose lexicographic ordering constraints on rows and columns and look for a solution, following the details in [11]. We propagate the Lex constraints either using the specialised algorithm GACLex given in [8] or the Slide encoding described in Section 7.

Table 1 shows the results on some large instances described as v, b, r, k, λ. As both propagators maintain GAC, we report the runtime results. We observe that the Slide encoding of Lex is as efficient (and sometimes even slightly more efficient than) the specialised algorithm.

12.2 Nurses Scheduling Problem

We consider a variant of the Nurse Scheduling Problem [6] that consists of generating a schedule for each nurse of shifts duties and days off within a short-term planning period. There are three types of shifts (day, evening, and night). We ensure that (1) each nurse should take a day off or be assigned to an available shift; (2) each shift has a minimum required number of nurses; (3) each nurse work load should be between specific lower and upper bounds; (4) each nurse can work at most 5 consecutive days; (5) each nurse must have at least 12 hours of break between two shifts; (6) each nurse should have at least two consecutive days on any shift. We wrote two models to solve this problem. In both models, we introduce one variable for each nurse and each day, indicating to what type of shift, if any, this nurse is affected on this day. The constraints (1)-(3) are

	GACLex algorithm	Slide encoding
Instance	time (s)	time (s)
7,91,39,3,13	0.59	0.55
9,72,24,3,6	0.57	0.53
9,96,32,3,8	2.20	2.16
9,108,36,3,9	2.13	2.11
10,90,27,3,6	1.26	1.28
10,120,36,3,8	3.38	3.50
11,110,30,3,6	2.55	2.65
12,88,22,3,4	1.28	1.25
13,78,18,3,3	0.98	1.00
13,104,24,3,4	2.15	2.13
15,21,7,5,2	26.78	26.60
15,70,14,3,2	0.97	0.91
16,32,12,6,4	452.25	450.96
16,80,15,3,2	1.49	1.39
19,57,9,3,1	2.70	2.63
22,22,7,7,2	73.97	71.81

Table 1. BIBD generation.

	Slide encoding		No Slide encoding	
Instance	time (s)	backtracks	time (s)	backtracks
10×14	82.32	271,348	133.44	776,019
12×14	4.52	13,484	11.57	58,709
14×14	0.37	1,356	0.29	1,877
10×16	0.83	4,116	1.35	10,017

Table 2. Nurse Schedule generation.

enforced using a set of global cardinality constraints. Constraints (4), (5) and (6) form sequences of respectively 6-ary, binary and ternary constraints. Notice that (4) is monotone, hence we simply posted these constraints in both models. However, the conjunction of constraints (4) and (5) is slid in the first model whilst it is decomposed in the second.

To test the two models, we generated by hand four instances respecting common sense criteria, such as lower demand during evening and night shifts. We observe that in the four instances the slide model outperforms the other model in terms either of backtracks or both cpu time and backtracks. It manages to solve the four instances in less time (cpu time ratio of 1.37 in average) and with fewer backtracks (backtrack ratio of 2.75 in average). This shows the effectiveness of Slide both as a modelling construct and as well as a specialised propagator.

13 Related work

Beldiceanu and Carlsson introduced the CardPath meta-constraint [1]. They showed that it can be used to encode a wide range of constraints like Change, Smooth, AmongSeq and SlidingSum. They provided a propagator for CardPath that greedily constructs upper and lower bounds on the number of (un)satisfied constraints by posting and retracting (the negation of) each of the constraints. This propagator does not achieve GAC.

Pesant introduced the Regular constraint, and gave a propagation algorithm based on dynamic programming that enforces GAC [14]. As we saw, the Regular constraint can be encoded using a simple Slide. There are, however, a number of important differences between the two. First, Regular only slides a ternary constraint down a sequence of variables. Slide, however, can slide a constraint of *any* arity. This permits us to deal with constraints like AmongSeq. Second, Pesant proposed a specialized propagator for Regular based on dynamic programming. This is unnecessary as we can achieve GAC by simply decomposing the Slide constraint into a sequence of ternary constraints. Third, as we described earlier, our encoding introduces variables for representing the states and access to these state variables can be useful (e.g. for expressing objective functions).

Beldiceanu, Carlsson and Petit have also proposed specifying global constraints by means of deterministic finite automata augmented with counters [3]. Propagators for such automata are constructed automatically by decomposing the specification into a sequence of signature and transition constraints. If the automaton uses counters, this decomposition hinders propagation so pairwise consistency is needed in general to guarantee GAC. We can encode such automata using a Slide where we introduce an additional sequence of variables for each counter. Our methods thus provide a GAC propagator for such automata.

Hellsten, Pesant and van Beek proposed a propagator for the Stretch constraint that achieves GAC based on dynamic programming similar to that for the Regular constraint [9]. We can again encode the Stretch constraint using a simple Slide.

14 Conclusions

We have studied the Slide meta-constraint. This slides a constraint down one or more sequences of variables. We have shown that Slide can be used to encode and propagate a wide range of global constraints including AmongSeq, CardPath, Precedence, and Regular. We have also considered a number of extensions including sliding down sequences of set variables, and combining Slide with a global cardinality constraint. When the constraint being slid overlap on just one variable, we argued that decomposition does not hinder propagation and Slide can be propagated simply by posting the sequence of constraints. Our experiments demonstrated that using Slide to encode constraints can be just as efficient and effective as specialized propagators. There are many directions for future work. One promising direction is to use binary decision diagrams

to store the supports for the constraints being slid when they have many satisfying tuples. We believe this could improve the efficiency of our propagator in many cases.

References

1. N. Beldiceanu and M. Carlsson. Revisiting the cardinality operator and introducing cardinality-path constraint family. In *Proc. of ICLP'01*, pp. 59–73. Springer-Verlag, 2001.
2. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20:97–123, no. 12 1994.
3. N. Beldiceanu, M. Carlsson, and T. Petit. Deriving filtering algorithms from constraint checkers. In *Proc. of CP'04*, pp. 107–122. Springer, 2004.
4. N. Beldiceanu, M. Carlsson, and J-X. Rampon. Global constraints catalog. SICS Technical Report T2005/08, 2005.
5. C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. The range and roots constraints: Specifying counting and occurrence problems. In *Proc. of IJCAI'05*, pp. 60–65. Professional Book Center, 2005.
6. Burke, E. K.; Causmaecker, P. D.; Berghe, G. V.; and Landeghem, H. V. 2004. The state of the art of nurse rostering. *J. of Scheduling* 7(6):441–499.
7. R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.
8. A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In *Proc. of CP'02*, pp. 93–108. Springer, 2002.
9. L. Hellsten, G. Pesant, and P. van Beek. A domain consistency algorithm for the stratch constraint. In *Proc. of CP'04*, pp. 290–304. Springer, 2004.
10. P. Van Hentenryck and J.-P. Carillon. Generality versus specificity: An experience with AI and OR techniques. In *Proc. of AAAI'88*, pp. 660–664. AAAI Press/The MIT Press, 1988.
11. Z. Kiziltan. Symmetry Breaking Ordering Constraints. PhD Thesis, Uppsala University, 2004.
12. Y.C. Law and J.H.M. Lee. Global constraints for integer and set value precedence. In *Proc. of CP'04*, pp. 362–376. Springer, 2004.
13. M. Maher. Analysis of a global contiguity constraint. In *Proc. of the CP'02 Workshop on Rule Based Constraint Reasoning and Programming*, 2002.
14. G. Pesant. A regular language membership constraint for finite sequences of variables. In *Proc. of CP'04*, pp. 482–295. Springer, 2004.
15. P. Refalo. Linear formulation of constraint programming models and hybrid solvers. In *Proc. of CP'00*, pp. 369–383. Springer, 2000.
16. J-C. Régin and J-F. Puget. A filtering algorithm for global sequencing constraints. In *Proc. of CP'97*, pp. 32–46. Springer, 1997.
17. J-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proc. of AAAI'94*, pp. 362–367. AAAI Press, 1994.
18. J-C. Régin. The global minimum distance constraint. Technical report, ILOG Inc, 1997.
19. W-J. van Hoeve, G. Pesant, and L-M. Rousseau. On global warming : Flow-based soft global constaints. To appear in *Journal of Heuristics*.

A Study of Residual Supports in Arc Consistency

Christophe Lecoutre and Fred Hemery

CRIL (Centre de Recherche en Informatique de Lens)
CNRS FRE 2499
rue de l'université, SP 16
62307 Lens cedex, France
{lecoutre,hemery}@cril.univ-artois.fr

Abstract. In an Arc Consistency (AC) algorithm, a residual support, or residue, is a support that has been stored during a previous execution of the procedure which determines if a value is supported by a constraint. The point is that a residue is not guaranteed to represent a lower bound of the smallest current support of a value. In this paper, we study the theoretical impact of exploiting residues with respect to the basic algorithm AC3. First, we prove that AC3r (AC3 with uni-directional residues) and AC3rm (AC3 with multi-directional residues) are optimal for low and high constraint tightness. Second, we show that when AC has to be maintained during a backtracking search (the well-known MAC algorithm), MAC2001 presents, with respect to MAC3r and MAC3rm, an overhead in O(μed) per branch of the binary tree built by MAC, where μ denotes the number of refutations of the branch, e the number of constraints and d the greatest domain size of the constraint network. One consequence is that, MAC3r and MAC3rm admit a better worst-case time complexity than MAC2001 for a branch involving μ refutations when either $\mu > d^2$ or $\mu > d$ and the tightness of any constraint is either low or high. Our experimental results clearly show that exploiting residues allows enhancing MAC and SAC algorithms.

1 Introduction

It is well-known that Arc Consistency (AC) plays a central role in solving instances of the Constraint Satisfaction Problem (CSP). Indeed, the MAC algorithm [17], i.e., the algorithm which maintains arc consistency during the search of a solution, is still considered as the most efficient generic approach to cope with large and hard problem instances. Furthermore, AC is at the heart of a stronger consistency called Singleton Arc Consistency (SAC) which has recently attracted a lot of attention (e.g., [2, 9]).

For more that two decades, many algorithms have been proposed to establish arc consistency. Today, the most referenced algorithms are AC3 [13] because of its simplicity and AC2001/3.1 [3] because of its optimality (while being not too complex). The worst-case time complexities of AC3 and AC2001 are respectively O(ed^3) and O(ed^2) where e denotes the number of constraints and d the greatest domain size. The interest of an optimal algorithm such as AC2001 resides in its robustness. It means that, on some instances, AC2001 can largely be faster than an algorithm such as AC3 whereas the reverse is not true. This situation occurs when the tightness of the constraints is high, as it is the case for the equality constraint (i.e. constraint of the form $X = Y$).

Indeed, as naturally expected and demonstrated later, AC3 admits then a practical behaviour which is close to the worst-case, and the difference by a factor d between the two theoretical worst-case complexities becomes a reality.

In this paper, we are interested in residues for AC algorithms. A residue is a support that has been stored during a previous execution of the procedure which determines if a value is supported by a constraint. The point is that a residue is not guaranteed to represent a lower bound of the smallest current support of a value. The basic algorithm AC3 can be refined by exploiting residues as follows: before searching a support for a value from scratch, the validity of the residue associated with this value is checked. We then obtain an algorithm denoted AC3r, and when multi-directionality is exploited, an algorithm denoted AC3rm.

In fact, AC3r is an algorithm which can be advantageously replaced by AC2001 when AC must be established stand-alone on a given constraint network. However, when AC has to be maintained during search, MAC3r which corresponds to mac3.1-residue [11] becomes quite competitive. On the other hand, AC3rm is interesting of its own as it exploits *multi-directional* residues just like AC3.2 [8]. But, let us see the interest of exploiting residues.

First, we prove in this paper that AC3r and AC3rm, contrary to AC3, admits an optimal behaviour when the tightness of the constraints is high. To illustrate this, let us consider the Domino problem introduced in [3]. All but one constraints of this problem correspond to equality constraints. The results that we obtain when running AC3, AC2001, AC3.2 and the new algorithm AC3rm on some instances of this problem are depicted in Table 1. We do not consider AC3r as it is always outperformed by AC2001 (even if, on these very special instances, they have the same behaviour). The time in seconds (cpu) and the number of constraint checks (ccks) is given for each instance of the form $domino$-n-d where n corresponds to the number of variables and d the number of values in each domain. Clearly, AC3rm largely compensates the weakness of the basic AC3.

Next, we analyse the cost of managing data structures with respect to backtracking. On the one hand, it is easy to embed AC3, AC3r or AC3rm in MAC and SAC algorithms as these algorithms do not not require any maintenance of data structures during MAC search and SAC inference. On the other hand, embedding an optimal algorithm such as AC2001 entails an extra development effort, with, in addition, an overhead at the execution. For MAC2001, this overhead is O(μed) per branch of the binary tree built by MAC as we have to take into account the reinitialization of a structure (called $last$)

$Instances$		$AC3$	$AC3rm$	$AC2001$	$AC3.2$
domino-100-100	cpu	1.81	0.16	0.23	0.18
	$ccks$	$18M$	$990K$	$1485K$	$990K$
domino-300-300	cpu	134	3.40	6.01	3.59
	$ccks$	$1377M$	$27M$	$40M$	$27M$
domino-500-500	cpu	951	15.0	21.4	15.2
	$ccks$	$10542M$	$125M$	$187M$	$125M$
domino-800-800	cpu	6144	60	87	59
	$ccks$	$68778M$	$511M$	$767M$	$511M$

Table 1. Establishing Arc Consistency on Domino instances

which contains smallest found supports. Here, μ denotes the number of refutations of the branch, e denotes the number of constraints and d the greatest domain size of the constraint network.

The paper is organized as follows. First, we introduce constraint networks and present a new algorithm denoted AC3rm. Then, we show the theoretical interest of using AC3/AC3rm in MAC and SAC algorithms. After presenting the results of an experimentation that we have conducted, we conclude.

2 Constraint Networks

A (finite) Constraint Network (CN) P is a pair $(\mathscr{X}, \mathscr{C})$ where $\mathscr{X}$ is a finite set of n variables and $\mathscr{C}$ a finite set of e constraints. Each variable $X \in \mathscr{X}$ has an associated domain, denoted $dom(X)$, which contains the set of values allowed for X. Each constraint $C \in \mathscr{C}$ involves a subset of variables of $\mathscr{X}$, called scope and denoted $vars(C)$, and has an associated relation, denoted $rel(C)$, which contains the set of tuples allowed for the variables of its scope. The initial (resp. current) domain of a variable X is denoted $dom^{init}(X)$ (resp. $dom(X)$). For each r-ary constraint C such that $vars(C) = \{X_1, \ldots, X_r\}$, we have: $rel(C) \subseteq \prod_{i=1}^{r} dom^{init}(X_i)$ where $\prod$ denotes the Cartesian product. Also, for any element $t = (a_1, \ldots, a_r)$, called tuple, of $\prod_{i=1}^{r} dom^{init}(X_i)$, $t[X_i]$ denotes the value a_i. It is also important to note that, assuming a total order on domains, tuples can be ordered using a lexicographic order $\prec$. To simplify the presentation of some algorithms, we will use two special values $\bot$ and $\top$ such that any tuple t is such that $\bot \prec t \prec \top$.

Definition 1. *Let C be a r-ary constraint such that $vars(C) = \{X_1, \ldots, X_r\}$, a r-tuple t of $\prod_{i=1}^{r} dom^{init}(X_i)$ is said to be:*

- *allowed by C iff $t \in rel(C)$,*
- *valid iff $\forall X_i \in vars(C)$, $t[X_i] \in dom(X_i)$,*
- *a support in C iff it is allowed by C and valid.*

A tuple t will be said to be a support of (X_i, a) in C when t is a support in C such that $t[X_i] = a$. Determining if a tuple is allowed is called a constraint check. A solution to a constraint network is an assignment of values to all the variables such that all the constraints are satisfied. A constraint network is said to be satisfiable iff it admits at least one solution. The Constraint Satisfaction Problem (CSP) is the NP-complete task of determining whether a given constraint network is satisfiable. A CSP instance is then defined by a constraint network, and solving it involves either finding one (or more) solution or determining its unsatisfiability. Arc Consistency (AC) remains the central property of constraint networks and establishing AC on a given network P involves removing all values that are not arc consistent.

Definition 2. *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN. A pair (X, a), with $X \in \mathscr{X}$ and $a \in dom(X)$, is arc consistent (AC) iff $\forall C \in \mathscr{C} \mid X \in vars(C)$, there exists a support of (X, a) in C. P is AC iff $\forall X \in \mathscr{X}$, $dom(X) \neq \emptyset$ and $\forall a \in dom(X)$, (X, a) is AC.*

The following definitions will be useful later to analyze the worst-case time complexity of some algorithms.

Definition 3. *A cn-value is a triplet of the form (*C*,*X*,*a*) where* $C \in \mathscr{C}$, $X \in vars(C)$ *and* $a \in dom(X)$.

Definition 4. *Let (*C*,*X*,*a*) be a cn-value such that* $vars(C) = \{X, Y\}$.

- *The number of supports of* (X, a) *in* C*, denoted* $s_{(C,X,a)}$*, corresponds to the size of the set* $\{b \in dom(Y) \mid (a, b) \in rel(C)\}$.
- *The number of conflicts of* (X, a) *in* C*, denoted* $c_{(C,X,a)}$*, corresponds to the size of the set* $\{b \in dom(Y) \mid (a, b) \notin rel(C)\}$.

Note that the number of cn-values that can be built from a binary constraint network is O(ed). To sum up all evaluations of an expression $Expr(C, X, a)$ wrt all the cn-values of a given CN, we will write: $\sum_{C,X,a} Expr(C, X, a)$.

3 Coarse-grained AC algorithms

In this section, we introduce AC3 and AC3rm, and we propose a detailed analysis of their complexities. It is important to remark that our algorithms are given in the general case (i.e. they can be applied to instances involving constraints of any arity). Hence, strictly speaking, their descriptions correspond to GAC3 and GAC3rm since for non binary constraints, one usually talks about Generalized Arc Consistency (GAC). However, to simplify, theoretical complexities will be given for binary instances. More precisely, for all theoretical results, we will consider given a binary constraint network $P = (\mathscr{X}, \mathscr{C})$ such that, to simplify and without any loss of generality, each domain exactly contains d values.

To establish (generalized) arc consistency on a given CN, $doAC$ (Algorithm 1) can be called. It returns $true$ when the given constraint network can be made arc-consistent and it is described in the context of a coarse-grained algorithm. Initially, all pairs (C, X), called arcs, are put in a set Q. Once Q has been initialized, each arc is revised in turn (line 4), and when a revision is effective (at least one value has been removed), the set Q has to be updated (line 6). A revision is performed by a call to the function $revise$ specific to the chosen coarse-grained arc consistency algorithm, and entails removing values that have become inconsistent with respect to C. This function returns $true$ when the revision is effective. The algorithm is stopped when a domain wipe-out occurs (line 5) or the set Q becomes empty.

Algorithm 1 doAC (P = $(\mathscr{X}, \mathscr{C})$: Constraint Network): Boolean

```
1: Q ← {(C, X) | C ∈ 𝒞 ∧ X ∈ vars(C)}
2: while Q ≠ ∅ do
3:    pick and delete (C, X) from Q
4:    if revise(C,X) then
5:       if dom(X) = ∅ then return false
6:       Q ← Q ∪ {(C', Y) | C' ∈ 𝒞, C' ≠ C, Y ≠ X, {X, Y} ⊆ vars(C')}
7: return true
```

Algorithm 2 revise3(C : Constraint, X : Variable) : Boolean

1: nbElements $\leftarrow$ | $dom(X)$ |
2: **for** each $a \in dom(X)$ **do**
3: **if** seekSupport3$(C, X, a) = \top$ **then**
4: remove a from $dom(X)$
5: return nbElements $\neq$ |$dom(X)$|

Algorithm 3 seekSupport3(C, X, a) : Tuple

1: $t \leftarrow \bot$
2: **while** $t \neq \top$ **do**
3: **if** $C(t)$ **then** return t
4: $t \leftarrow setNextValid(C, X, a, t)$
5: return $\top$

3.1 AC3

For AC3 [13], each revision is performed by a call to the function $revise3(C, X)$, depicted in Algorithm 2. This function iteratively calls the function $seekSupport3$ which determines from scratch whether or not there exists a support of (X, a) in C. It uses $setNextValid$ which returns either the smallest valid tuple t' built from C such that $t \prec t'$ and $t'[X] = a$, or $\top$ if it does not exist. For binary constraints, we assume that any call to $setNextValid$ is performed in constant time (e.g., see [10]). Note that $C(t)$ must be understood as a constraint check and that $C(\bot)$ returns false.

AC3 has a non-optimal worst-case time complexity of $O(ed^3)$ [14]. However, it is possible to refine this result by focusing on the cumulated cost of seeking successive supports of a value (X,a) in a constraint C. We have the following results[1]:

Proposition 1. *In AC3, the worst-case cumulated time complexity of $seekSupport3$ for a given cn-value (C,X,a) is O($cd + s$) with $c = c_{(C,X,a)}$ and $s = s_{(C,X,a)}$.*

Proof. Let us evaluate the maximum number of constraint checks that can be performed (during the full process of propagation) when iteratively calling $seekSupport3(C, X, a)$. Let us consider vars(C) = $\{X, Y\}$, $c = c_{(C,X,a)}$ and $s = s_{(C,X,a)}$. Note that $d = c + s$. The worst-case is when:
1) only one value is removed from dom^{init}(Y) between two calls to $revise3(C, X)$,
2) values of dom^{init}(Y) are ordered in such a way that the c first values correspond to values which do not support a and the s last values correspond to values which support a,
3) the first $s - 1$ values removed from dom^{init}(Y) correspond to values which support a and the c next values removed from dom^{init}(Y) correspond to values which do not support a.
Hence, for the first s calls to $seekSupport3(C, X, a)$, we obtain $s * (c + 1)$ constraint checks. For the next c calls, we obtain $c + (c - 1) + \cdots + 1$ checks. Then, we have a worst-case cumulated complexity in O($sc + s + c^2$) = O($c(s + c) + s$) = O($cd + s$). □

[1] Note that s cannot be ignored since c can be equal to 0.

It is interesting to note that c represents a tightness parameter (see also [6], page 61). When the constraint tightness is low (more precisely, when c is O(1)), the worst-case cumulated time complexity becomes $O(d)$, which is optimal. However, when the constraint tightness is high (when c is O(d)), it becomes $O(d^2)$. Unsurprisingly, this result indicates that AC3 is adapted to instances involving constraints of low tightness. We can now deduce the following result.

Proposition 2. *The worst-case time complexity of AC3 is:* $O(d * \sum_{C,X,a} c_{(C,X,a)} + \sum_{C,X,a} s_{(C,X,a)})$.

3.2 AC3rm

Following the principle used in AC3.2 [8], we propose a mechanism to partially benefit from (positive) multi-directionality. The idea is that, when a support t is found, it can be recorded for all values occurring in t. For example, let us consider a binary constraint C such that *vars*(C) = $\{X, Y\}$. If (a, b) is found in C when looking for a support of either (X, a) or (Y, b), in both cases, it can be recorded as being the last found support of (X, a) in C and the last found support of (Y, b) in C. In fact, one can simply record for any cn-value (C,X,a) the last found support of (X,a) in C. However, here, unlike AC2001, by exploiting multi-directionality, we cannot benefit anymore from uni-directionality. It means that, when the last found support is no more valid, one has to search for a new support from scratch. Indeed, by using multi-directionality, we have no guarantee that the last found support corresponds to the last smallest support. This new algorithm requires the introduction of a a three-dimensional array, denoted *supp*. This data structure is used to store for any cn-value (C,X,a) the last found support of (X,a) in C. Initially, any element of the structure *supp* must be set to $\perp$. Each revision (see Algorithm 4) involves testing for any value the validity of the last found support (line 3) and if, it fails, a search for a new support is started from scratch (note the call to seekSupport3). If this search succeeds, structures corresponding to last found supports are updated (line 6).

To summarize, the structure *supp* allows to record what we call *multi-directional* residues. Of course, it is possible to exploit simpler residues [11], called here *uni-directional* residues, by not exploiting multi-directionality. We can derive a new algorithm, denoted AC3r, by replacing line 8 of Algorithm 4 with: $supp[C, X, a] \leftarrow t$

Algorithm 4 revise3rm(C : Constraint, X : Variable) : Boolean

```
1: nbElements ← | dom(X) |
2: for each a ∈ dom(X) do
3:    if supp[C, X, a] is valid then continue
4:    t ← seekSupport3(C, X, a)
5:    if t = ⊤ then
6:       remove a from dom(X)
7:    else
8:       for each Y ∈ vars(C) do supp[C, Y, t[Y]] ← t
9: return nbElements ≠ |dom(X)|
```

However, with AC3r, when AC must be established stand-alone, rather than searching a new support from scratch when the residue is no more valid, it is more natural and more efficient to perform the search using the value of the residue as a resumption point. This is exactly what is done by AC2001. It means that, in practice, AC3r is interesting (as we will see) only when it is embedded in MAC [11] or a SAC algorithm.

AC3rm has a space complexity of $O(ed)$ and a non-optimal worst-case time complexity of $O(ed^3)$. However, it is possible to refine this result as follows:

Proposition 3. *In AC3rm (and AC3r), the worst-case cumulated time complexity of* $seekSupport3$ *for a cn-value (*C*,*X*,*a*) is O(*$cs + d$*) with* $c = c_{(C,X,a)}$ *and* $s = s_{(C,X,a)}$.

Proof. As for AC3, the worst-case in terms of constraint checks is when:
1) only one value is removed from dom$^{init}(Y)$ between two calls to revise3rm(C,X),
2) values of dom$^{init}(Y)$ are ordered in such a way that the c first values correspond to values which do not support a and the s last values correspond to values which support a.
But, unlike AC3, the worst-case in terms of constraint checks is when the first s values removed from dom$^{init}(Y)$ systematically correspond to the last found supports recorded by AC3rm (until a domain wipe-out is encountered). For these $s + 1$ calls (note the initial call) to $seekSupport3(C, X, a)$, we obtain $s * (c + 1) + c$ constraint checks. On the other hand, the number of other operations (validity checks and updates of the supp structure) in revise3rm performed with respect to a is bounded by d. Then, we have a worst-case cumulated complexity in O($sc + s + c + d$) = O($cs + d$). □

What is interesting with AC3rm is that, even if this algorithm is not optimal, it is adapted to instances involving constraints of low or high tightness. Indeed, when the constraint tightness is low (more precisely, when c is O(1)) or high (when s is O(1)), the worst-case cumulated time complexity becomes $O(d)$, what is optimal. On the other hand, sc is maximized when $c = s = d/2$, what corresponds to a medium constraint tightness. However, AC3rm can also be expected to have a good (practical) behavior for medium constraint tightness since, on average (i.e. asymptotically), considering random constraints, 2 constraint checks are necessary to find a support when the tightness is 0.5. We can deduce the following result.

Proposition 4. *The worst-case time complexity of AC3rm (and AC3r) is:* $O(ed^2 + \sum_{C,X,a} c_{(C,X,a)} * s_{(C,X,a)})$.

Finally, remark that we have not introduced AC3rm with respect to the framework AC-* [15] as AC3rm is not an instance of AC-*. However, we think that it should be possible to extend this framework to include the concept of residues.

3.3 Overview of Complexities

Table 2 indicates the overall worst-case complexities to establish arc consistency with algorithms AC3, AC3r(m)[2], AC2001 and AC3.2 (due to lack of space, AC2001 and AC3.2 are not described in this paper).

[2] AC3r(m) will interchangeably denote both algorithms AC3r and AC3rm.

	Space	Time
AC3	$O(e+nd)$	O($d * \sum_{C,X,a} c_{(C,X,a)} + \sum_{C,X,a} s_{(C,X,a)}$)
AC3r(m)	$O(ed)$	O($ed^2 + \sum_{C,X,a} c_{(C,X,a)} * s_{(C,X,a)}$)
AC2001	$O(ed)$	$O(ed^2)$
AC3.2	$O(ed)$	$O(ed^2)$

Table 2. Worst-case complexities to establish AC.

	Tightness			
	Any	Low	Medium	High
AC3	$O(cd+s)$	$O(d)$	$O(d^2)$	$O(d^2)$
AC3r(m)	$O(cs+d)$	$O(d)$	$O(d^2)$	$O(d)$
AC2001	$O(d)$	$O(d)$	$O(d)$	$O(d)$
AC3.2	$O(d)$	$O(d)$	$O(d)$	$O(d)$

Table 3. Cumulated worst-case time complexities to seek successive supports for a given cn-value (C,X,a). We have $c + s = d$.

It is also interesting to look at worst-case cumulated time complexities to seek successive supports for a given cn-value (C,X,a). Even if it has not been introduced earlier, it is easy to show that optimal algorithms admit a cumulated complexity in O(d). By observing Table 3, we do learn that AC3 and AC3r(m) are optimal when the tightness is low (i.e. c is O(1)), and that, unlike AC3, AC3r(m) is also optimal when the tightness is high (i.e. s is O(1)).

4 Maintaining Arc Consistency

In this section, we focus on maintaining arc consistency during search. More precisely, we study the impact, in terms of time and space, of embedding some AC algorithms in MAC. The MAC algorithm aims at solving a CSP instance and performs a depth-first search with backtracking while maintaining arc consistency. At each step of the search, a variable assignment is performed followed by a filtering process that corresponds to enforcing arc-consistency.

When mentioning MAC, it is important to indicate which branching scheme is employed. Indeed, it is possible to consider binary (2-way) branching or non binary (d-way) branching. These two schemes are not equivalent as it has been shown that binary branching is more powerful (to refute unsatisfiable instances) than non-binary branching [7]. With binary branching, at each step of the search, a pair (X,a) is selected where X is an unassigned variable and a a value in $dom(X)$, and two cases are considered: the first one corresponds to the assignment $X = a$ and the second one to the refutation $X \neq a$. Figure 4 depicts a branch in a binary tree built by MAC. This branch that leads to a solution (represented by a square) involves two variable assignments ($Y = b$ and $X = b$) and two value refutations ($X \neq a$ and $X \neq c$). These refutations are the consequences of two explored sub-trees (from $X = a$ and $X = c$).

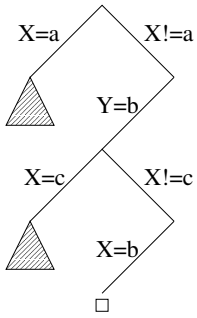


Fig. 1. Branch in a binary tree

Also, it is important to remark that all known AC algorithms (including AC3r and AC3rm) are incremental. An arc-consistency algorithm is incremental if its worst-case time complexity is the same when it is applied one time on a given network P and when it is applied up to nd times on P where between two consecutive executions at least one value has been deleted. By exploiting incrementality, one can get the same complexity, in terms of constraint checks, for any branch of the search tree as for only one establishment of AC.

For AC3 and AC3r(m), the (non optimal) worst-case time complexity for any branch of the search tree is guaranteed (by incrementality) even if, meanwhile, sub-trees have been explored and then backtracking has occurred. However, for optimal algorithms AC2001 and AC3.2, it is important to manage the data structure, denoted $last$, in order to restart search, after exploring a sub-tree, as if backtracking never occurred. In this paper, MAC2001 and MAC3.2 correspond to the algorithms that record the smallest supports that have been successively found all along the current branch. Note that it is at the price of a space complexity in O(min(n,d)ed) [18]. Although an elegant approach to avoid such additional cost has been proposed in [16], the proposed method, which involves recomputing last smallest supports, is complex and the preliminary results given by the author are disappointing. In any case (that is to say, for any variant), one has to be careful of also taking into account the time cost associated with the requirement of maintaining (initializing) the data structure $last$ whenever the AC algorithm is called.

Proposition 5. *In MAC2001 and MAC3.2, the worst-case cumulated time complexity of reinitializing the structure $last$ is O(μed) for any branch involving μ refutations.*

Proof. For any refutation occurring in a branch, we need to restore the data structure $last$ since, otherwise, we could not keep exploring the search tree. In the worst-case, we have at most $e * 2 * d$ operations since for each cn-value (C,X,a), we have to reinitialize $last[C, X, a]$ to a stacked value (or, for variants, to $\perp$ or a new recomputed value). Hence, we obtain (μed). □

When $\mu = 0$, it means that a solution has been found without any backtracking. In this case, there is no need to restore the structure $last$ as the instance is solved. At the opposite, we know that the longest branch that can be built contains nd edges as

	Space	Time (per branch)
MAC3	$O(e+nd)$	O($ed^2 + d * \sum_{C,X,a} c_{(C,X,a)}$)
MAC3r(m)	$O(ed)$	O($ed^2 + \sum_{C,X,a} c_{(C,X,a)} * s_{(C,X,a)}$)
MAC2001	$O(min(n,d)ed)$	$O(ed(d+\mu))$
MAC3.2	$O(min(n,d)ed)$	$O(ed(d+\mu))$

Table 4. Worst-case complexities to run MAC. Time complexity is given for a branch involving μ refutations.

follows: for each variable X, there are exactly $d-1$ edges that correspond to refutations and only one edge that corresponds to an assignment. Then, we obtain a worst-case cumulated time complexities of reinitializing the structure $last$ in O(end^2) and although it is omitted here, we can also show that it is $\Omega(end^2)$.

One nice feature of AC3r(m) is that, when they are embedded in MAC, no initialization is necessary at each step (when going forward and when backtracking) since the principle of this algorithm is to record the last found support which does not systematically correspond to the last smallest one. In fact, we reported in [8] that it is worthwhile to leave unchanged last found supports (using AC3.2) while backtracking, having the benefit of a so-called memorization effect. It means that a support found at a given depth of the search has the opportunity to be still valid at a higher depth of the search (when going forward) but also at a lower depth (after backtracking). In other words, it is worthwhile to exploit residues during search. The importance of limiting in MAC the overhead of maintaining the data structures employed by the embedded AC algorithm was pointed out in [11] (but no complexity result was given). In fact, MAC3r corresponds to the algorithm mac3.1residue introduced in [11]. Another elegant proposal, denoted ADO, involves reordering domains after each backtrack [12] in order to avoid the overhead of maintaining the structure $last$. Unluckily, the optimality of the algorithm, given in the paper, does not hold (Likitvivatanavong, Personal Communication).

By taking into account Proposition 5 and Table 2, we obtain the results given in Table 4. It appears that, for the longest branch, when $\mu > d^2$, MAC3 and MAC3r(m) have a better worst-case time complexity than other MAC algorithms based on optimal AC algorithms since we know that, for any branch, due to incrementality, MAC3 and MAC3r(m) are O(ed^3). Also, if the tightness of any constraint is either low or high (more precisely, if for any cn-value (C,X,a), either $c_{(C,X,a)}$ is O(1) or $s_{(C,X,a)}$ is O(1)), then MAC3r(m) admits an optimal worst-case time complexity in O(ed^2) per branch. In this case, MAC3r(m) outperforms MAC2001 as soon as $\mu > d$. These observations suggest that MAC3r(m) should be very competitive.

5 Establishing Singleton Arc Consistency

There is a recent focus about singleton consistencies, and more particularly about SAC (Singleton Arc Consistency), as illustrated by some recent works [2, 9]. A constraint network is singleton arc consistent iff any singleton check does not show unsatisfiability, i.e., iff after performing any variable assignment, enforcing arc consistency on the

resulting network does not entail a domain wipe-out. One interesting question is: what about the cost of reinitializing AC structures in the context of SAC algorithms?

Due to lack of space, in this paper, we will focus on SAC-1 [5]. In fact, as any AC algorithm can be embedded in SAC-1, the question asked above is relevant when an optimal AC algorithm such as AC2001 is considered. The worst-case is obtained by considering O(n^2d^2) singleton checks, each of them having a cost in O(ed^2). However, in order to use an optimal AC algorithm without sacrificing space, one has to reinitialize the structure $last$ whenever a singleton check has to be performed. In SAC-1, the cumulated worst-case time complexity of reinitializing the AC2001 structures is then O(en^2d^3). The same result holds for SAC-2 [1]. Although is it less than O(en^2d^4), it can have a significant impact on the average time complexity, as shown below.

6 Experiments

To compare the different algorithms mentioned in this paper, we have performed a vast experimentation (run on a PC Pentium IV 2.4GHz 512MB under Linux) with respect to random, academic and real-world problems. Performances[3] have been measured in terms of the CPU time in seconds (cpu) and the number of constraint checks (ccks). In MAC (equipped with dom/deg[4]) and SAC-1, we have used a slightly modified version of AC3r(m): the smallest supports found at preprocessing are recorded in order to save more constraint checks. A new search for a support does not start from scratch but from these recorded values. Note that space and time complexities remain the same.

To start, we have tested MAC by considering 7 classes of binary random instances situated at the phase transition of search. For each class $\langle n,d,e,t\rangle$, defined as usually, 100 instances have been generated. The tightness t denotes the probability that a pair of values is allowed by a relation. What is interesting here is that a significant sampling of domain sizes, densities and tightnesses is introduced. In Table 5, we can observe the results obtained with MAC embedding the various AC algorithms. As expected, the best embedded algorithms are AC3 and AC3r(m) when the tightness is low (here 0.1) and AC3r(m) when the tightness is high (here, 0.9). Also, AC3r(m) is the best when the tightness is medium (here 0.5) as expected on random instances. All these results are confirmed for some representative selected academic and real-world instances. Clearly, MAC3r(m) outperform all other MAC algorithms in terms of cpu while MAC3.2 is the best (although beaten on a few instances by MAC3rm) in terms of constraint checks. Interestingly, in an overall analysis, we can remark that MAC3r(m) and MAC2001 roughly perform the same number of constraint checks. As, on the other hand, MAC3r(m) does not require any data structure to be maintained, it explains why it is the quickest approach. These results confirm those obtained for MAC3r (mac3.1residue) in [11].

Here, we have to mention that MAC3rm and MAC3r (mac3.1residue) [11] are close in terms of performance with respect to binary instances. Considering all results (not included, here) obtained for instances used as benchmarks of the first CSP solver compe-

[3] In our experimentation, all constraint checks are performed in constant time and are as cheap as possible since constraints are represented in extension (using arrays).

[4] We used this simple heuristic to allow an easy reproduction of our results.

		$MAC\ embedding$				
		$AC3$	$AC3r$	$AC3rm$	$AC2001$	$AC3.2$
Classes of random instances (mean results for 100 instances)						
⟨40-8-753-0.1⟩	cpu	22.68	21.71	22.96	34.38	33.35
	$ccks$	$81M$	$17M$	$17M$	$24M$	$16M$
⟨40-11-414-0.2⟩	cpu	21.19	18.76	19.23	27.91	26.34
	$ccks$	$97M$	$23M$	$22M$	$28M$	$19M$
⟨40-16-250-0.35⟩	cpu	21.86	18.03	18.31	25.18	23.38
	$ccks$	$121M$	$28M$	$28M$	$33M$	$24M$
⟨40-25-180-0.5⟩	cpu	37.35	24.52	25.53	35.30	32.27
	$ccks$	$233M$	$57M$	$56M$	$60M$	$45M$
⟨40-40-135-0.65⟩	cpu	37.62	26.74	26.62	35.98	34.45
	$ccks$	$344M$	$85M$	$83M$	$82M$	$64M$
⟨40-80-103-0.8⟩	cpu	89.01	50.37	51.62	67.74	61.48
	$ccks$	$1072M$	$243M$	$240M$	$225M$	$180M$
⟨40-180-84-0.9⟩	cpu	166.12	75.12	76.99	98.69	87.50
	$ccks$	$2540M$	$514M$	$506M$	$479M$	$381M$
Academic instances						
ehi-85-12	cpu	394	362	377	557	511
	$ccks$	$642M$	$58M$	$60M$	$190M$	$83M$
geo-50-20-19	cpu	194	148	157	278	263
	$ccks$	$1117M$	$249M$	$244M$	$284M$	$199M$
qa-5	cpu	31.60	27.21	28.31	37.49	36.02
	$ccks$	$130M$	$37M$	$36M$	$38M$	$27M$
qcp-819	cpu	139	136	143	215	208
	$ccks$	$116M$	$21M$	$21M$	$41M$	$25M$
Real-world instances						
$fapp$01-0200-9	cpu	0.54	0.39	0.37	0.60	0.60
	$ccks$	$6905K$	$3310K$	$3080K$	$3018K$	$2778K$
js-$enddr$2-3	cpu	53.66	28.24	29.08	39.24	29.60
	$ccks$	$596M$	$104M$	$104M$	$88M$	$48M$
$scen$-11	cpu	15.67	10.92	11.88	16.26	14.58
	$ccks$	$92M$	$18M$	$18M$	$15M$	$10M$
$graph$-10	cpu	0.64	0.53	0.54	0.69	0.68
	$ccks$	$4842K$	$2563K$	$2216K$	$2228K$	$1925K$

Table 5. Cost of running MAC

tition [19, 4], it appears that MAC3r is 5% faster than MAC3rm on some series ($bqwh$, ehi, frb, geo, qa, qk) whereas MAC3rm is 5% faster on $hanoi$ and $pigeons$, and 20% faster on $domino$ instances (where MAC3rm takes advantage of multidirectionality).

We have then embedded the AC algorithms in SAC-1. In Table 6, one can observe that, for domino instances which involve constraints of high tightness, AC3r(m) clearly shows its superiority to AC3. For real-world instances, the gap between AC3r(m) and the other AC algorithms, increases. For example, SAC-1 embedding AC3r(m) is about three times more efficient than SAC-1 embedding AC2001 on $scen11$ and about four time more efficient than SAC-1 embedding AC3 on $fapp$01-0200-9.

7 Residues for Non Binary Constraints

One can wonder what is the behaviour of an algorithm that exploits residues when applied to non binary instances. First, it is important to remark that seeking a support

		$SAC-1$ *embedding*				
		$AC3$	$AC3r$	$AC3rm$	$AC2001$	$AC3.2$
Academic instances						
$domino$-300-300	cpu	446.32	14.40	9.56	14.40	9.67
	$ccks$	$1376M$	$40M$	$26M$	$40M$	$26M$
$domino$-500-100	cpu	4.37	0.71	0.53	0.71	0.57
	$ccks$	$88M$	$7425K$	$4950K$	$7425K$	$4950K$
geo-50-20-19	cpu	1.18	0.73	0.74	1.49	1.24
	$ccks$	$9525K$	$1032K$	$1165K$	$2671K$	$1157K$
qa-5	cpu	1.22	0.82	0.89	1.91	1.34
	$ccks$	$10M$	$2700K$	$3104K$	$5001K$	$3085K$
Real-world instances						
$fapp$01-0200-9	cpu	637	153	158	312	192
	$ccks$	$10047M$	$838M$	$905M$	$1795M$	$904M$
js-$enddr$2-3	cpu	58.95	13.11	12.35	24.66	14.38
	$ccks$	$980M$	$55M$	$54M$	$128M$	$55M$
$graph$-10	cpu	980	424	439	836	581
	$ccks$	$12036M$	$1073M$	$1307M$	$2467M$	$1303M$
$scen$-11	cpu	44.89	20.72	21.07	56.03	53.21
	$ccks$	$479M$	$30M$	$33M$	$52M$	$37M$

Table 6. Cost of establishing SAC-1

		$MGAC$ *embedding*				
Instances		$GAC3$	$GAC3r$	$GAC3rm$	$GAC2001$	$GAC3.2$
Random instances (mean results for 10 instances)						
⟨6-20-6-32-0.55⟩	cpu	0.75	0.50	0.46	0.58	0.49
	$ccks$	$676K$	$357K$	$278K$	$364K$	$235K$
⟨6-20-6-36-0.55⟩	cpu	13.1	8.7	8.0	10.2	8.5
	$ccks$	$12M$	$6481K$	$4997K$	$6825K$	$4324K$
⟨6-20-8-22-0.75⟩	cpu	2.5	1.5	1.3	1.6	1.3
	$ccks$	$2313K$	$1240K$	$971K$	$1232K$	$804K$
⟨6-20-8-24-0.75⟩	cpu	51.7	31.8	27.7	34.6	26.8
	$ccks$	$48M$	$26M$	$20M$	$26M$	$17M$
⟨6-20-10-13-0.95⟩	cpu	35.2	20.7	15.8	20.8	13.9
	$ccks$	$40M$	$23M$	$17M$	$22M$	$14M$
⟨6-20-10-14-0.95⟩	cpu	220	135	102	135	89
	$ccks$	$249M$	$151M$	$108M$	$149M$	$91M$
⟨6-20-20-10-0.99⟩	cpu	659	392	267	254	177
	$ccks$	$1653M$	$1037M$	$647M$	$662M$	$462M$
⟨6-20-20-15-0.99⟩	cpu	869	489	301	351	220
	$ccks$	$2255M$	$1289M$	$785M$	$887M$	$583M$
Structured instances						
tsp-20-366	cpu	387	242	243	266	235
	$ccks$	$607M$	370	$364M$	$387M$	$333M$
gr-44-9-$a3$	cpu	73.1	37.2	38.4	56.3	43.6
	$ccks$	$166M$	$44M$	$41M$	$74M$	$33M$
gr-44-10-$a3$	cpu	2945	1401	1465	2129	1631
	$ccks$	$6819M$	$1513M$	$1527M$	$2914M$	$1224M$
$series$-14	cpu	233	218	217	312	285
	$ccks$	$1135M$	$531M$	$490M$	$618M$	$422M$
$renault$	cpu	25.0	25.4	16.2	25.2	15.2
	$ccks$	$68M$	$66M$	$42M$	$66M$	$42M$

Table 7. Cost of running MGAC

of a cn-value from scratch requires iterating O(d^{r-1}) tuples in the worst-case for a constraint of arity r. We then obtain a worst-case cumulated time complexity of seeking a support of a given cn-value in O(r^2d^r) for GAC3 and O(rd^{r-1}) for GAC2001 [3] since we consider that a constraint check is O(r) and since there are O(rd) potential calls to the specific $seekSupport$ function. Then, we can observe that there is a difference by a factor dr.

On the other hand, if we assume that $c > 0$ and $s > 0$ respectively denote the number of forbidden and allowed tuples of the constraint, then we obtain, by generalizing our results of Section 3, a complexity in O(crd^{r-1}) for GAC3 and in O(cs) for GAC3r(m). We can then deduce that the worst-case cumulated time complexity of seeking a support is O($min(c, rd).rd^{r-1}$) for GAC3 and O($min(csr, r^2d^r)$) for GAC3r(m). If $c = O(1)$ or $s = 0(1)$, we obtain O(rd^{r-1}) for GAC3r(m) as $c + s = d^{r-1}$, that is to say optimality. However, we admit that, in practice, the likelihood of having small values of c or s when dealing with non binary constraints is weak.

We have performed a preliminary experimentation by maintaining GAC algorithms during search (with dom/deg) on series of random non binary instances. These instances belong to classes of the form $\langle r, n, d, e, t\rangle$ where r denotes the arity of the constraints and all other parameters are defined as usual. Here, we chose constraints of arity 6 and studied the behaviour of the algorithm for a tightness $t \in \{0.55, 0, 75, 0.95, 0.99\}$. For small values of t, we observed (as in the binary case) that the difference between all algorithms was limited. On these random instances, one can observe in Table 7 that GAC3rm and GAC3.2 are the most efficient embedded algorithms. Of course, when the tightness is high, GAC3 is penalized and GAC3r is less efficient then GAC3rm as exploiting multi-directionality pays off. On the non binary structured instances of the competition, one can observe the good behaviour of GAC3r and GAC3rm. Of course, we believe that a more extensive experimentation must be performed including structured instances and constraints of larger arities.

8 Conclusion

In this paper, we have introduced some theoretical results about the use of residual supports, or residues, in Arc Consistency algorithms. The concept of residue has been introduced under its multi-directional form in [8] and under its uni-directional form in [11]. We have first proved that the basic algorithm AC3 which is optimal for low constraint tightness, also becomes optimal for high constraint tightness when it is extended to exploit uni-directional or multi-directional residues. Furthermore, these extensions to AC3, respectively called AC3r and AC3rm, can be expected to have a good (practical) behavior for medium tightness as asymptotically, for random constraints, 2 constraint checks are necessary to find a support when the tightness is 0.5. Then, we have shown that MAC3r(m) admit a better worst-case time complexity than MAC2001 for a branch of the binary search tree when either $\mu > d^2$ or $\mu > d$ and the tightness of any constraint is low or high, with μ denoting the number of refutations of the branch.

On the practical side, we have run a vast experimentation including MAC and SAC-1 algorithms on binary and non binary instances. The results that we have obtained clearly show the interest of exploiting residues as AC3r(m) (embedded in MAC or SAC-1) were almost always the quickest algorithms (only beaten by AC3.2 on some non

binary instances). It confirms for MAC3r (mac3.1residue) the results presented in [11]. In terms of constraint checks, it appears that AC3r(m) is quite close to AC2001 (but usually beaten by AC3.2). We also noted that AC3rm was more robust than AC3r on non binary instances and constraints of high tightness.

Finally, we want to emphasize that implementing (G)AC3r(m) (and embedding it in MAC or SAC) is quite easy as no maintenance of data structures upon backtracking is required. It should be compared with the intricacy of fine-grained algorithms which requires a clever use of data structures, in particular when applied to non binary instances. The simplicity of AC3r(m) offers another scientific advantage: the easy reproducibility of the experimentation by other researchers.

References

1. R. Bartak and R. Erben. A new algorithm for singleton arc consistency. In *Proceedings of FLAIRS'04*, 2004.
2. C. Bessiere and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI'05*, pages 54–59, 2005.
3. C. Bessiere, J.C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.
4. F. Boussemart, F. Hemery, and C. Lecoutre. Description and representation of the problems selected for the first international constraint satisfaction problem solver competition. In *Proceedings of CPAI'05 workshop held with CP'05*, pages 7–26, 2005.
5. R. Debruyne and C. Bessière. Some practical filtering techniques for the constraint satisfaction problem. In *Proceedings of IJCAI'97*, pages 412–417, 1997.
6. R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.
7. J. Hwang and D.G. Mitchell. 2-way vs d-way branching for CSP. In *Proceedings of CP'05*, pages 343–357, 2005.
8. C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *Proceedings of CP'03*, pages 480–494, 2003.
9. C. Lecoutre and S. Cardon. A greedy approach to establish singleton arc consistency. In *Proceedings of IJCAI'05*, pages 199–204, 2005.
10. C. Lecoutre and R. Szymanek. Generalized arc consistency for positive table constraints. In *Proceedings of CP'06*, 2006.
11. C. Likitvivatanavong, Y. Zhang, J. Bowen, and E.C. Freuder. Arc consistency in MAC: a new perspective. In *Proceedings of CPAI'04 workshop held with CP'04*, pages 93–107, 2004.
12. C. Likitvivatanavong, Y. Zhang, J. Bowen, and E.C. Freuder. Maintaining arc consistency using adaptive domain ordering. In *Proceedings of IJCAI'05*, pages 1527–1528, 2005.
13. A.K. Mackworth. Consistency in networks of relations. *Art. Intelligence*, 8(1):99–118, 1977.
14. A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.
15. J.C. Régin. AC-*: a configurable, generic and adaptive arc consistency algorithm. In *Proceedings of CP'05*, pages 505–519, 2005.
16. J.C. Régin. Maintaining arc consistency algorithms during the search without additional space cost. In *Proceedings of CP'05*, pages 520–533, 2005.
17. D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP'94*, pages 10–20, 1994.
18. M.R.C. van Dongen. Saving support-checks does not always save time. *Artificial Intelligence Review*, 21(3-4):317–334, 2004.
19. M.R.C. van Dongen, editor. *Proc. of CPAI'05 workshop held with CP'05*, volume II, 2005.

Maintaining Singleton Arc-Consistency

Christophe Lecoutre[1] and Patrick Prosser[2]

[1] CRIL-CNRS FRE 2499, Université d'Artois, Lens, France,
lecoutre@cril.univ-artois.fr
[2] Department of Computing Science, University of Glasgow, Scotland, pat@cs.gla.ac.uk

Abstract. Singleton Arc-Consistency (SAC) [8] is a simple and strong level of consistency but is costly to enforce. To date, research has focused on improving the performance of algorithms that achieve SAC, and comparing algorithms as a preprocessing step before actually solving a problem. Here, we show for the first time how a basic SAC algorithm can be readily incorporated into an open source constraint programming toolkit and then used within the search process i.e. the search process maintains SAC. We also present three new levels of SAC: Bound-SAC where the first and last values in domains are SAC, First-SAC where only the first value is SAC, and Existential-SAC where some value in the domain is SAC. Again, we show how these levels of SAC can be maintained by the search process, and present the first empirical study of their behaviours. This leads us to the point where we can investigate the effect of maintaining different levels of consistency on different sets of variables within a problem. We show experimentally that it can result in significant performance improvements.

1 Introduction

In 1997, Debruyne and Bessière introduced Singleton Arc-Consistency (SAC) [8]. In a constraint network P, a value a in the domain of a variable x is SAC if the variable x can be assigned the value a and P can then be made arc-consistent. P is SAC if all values in the domains of all variables are SAC. This gives a stronger level of consistency than AC but at a substantially higher cost. The complexity of achieving AC is $O(ed^2)$ [17] whereas the optimal cost of SAC is $O(end^3)$ [3], where e is the number of constraints, d is the size of the largest domain, and n is the number of variables. In [2], it was proved that SAC is a non-local property, unlike AC. Consequently, we should expect that it will be non-trivial to achieve practically efficient algorithms for this consistency. There have been three notable attempts at proposing practical algorithms. The first one [1] aims at avoiding some useless singleton checks by recording supports while the two others [3, 13] exploit the incrementality of arc-consistency. However, except for SAC3 [13], all proposed algorithms either require large data structures or are non-trivial to implement.

To date, SAC has been studied only as a preprocessing step prior to actually solving a problem, and has been typically applied to random instances, frequency assignment problems and problems of distance [8, 19, 1, 3, 13]. The study in [19] showed that SAC, as a preprocess, was rarely cost effective on random instances, but on structured problems such as networks with small-world graphs or Golomb rulers, SAC was often beneficial. Therefore it appears, so far, that although SAC may be promising it has not yet

been exploited inside search in the same way AC has [20], and that it has not yet been practically tested[3].

In this paper, we go some way towards putting this right. First, we go back to the basic SAC algorithm proposed in [8] and incorporate it into a constraint programming toolkit. We do this showing the actual code, demonstrating just how easily this can be engineered. This then allows us to investigate, for the first time, the behaviour of SAC inside search whilst exploiting all of the features of the constraint toolkit. That is, we maintain SAC within the search process and compare this to the gold standard of constraint programming, namely MAC [20].[4] We then take a pragmatic approach in our quest for performance improvements and restrict SAC such that only some of the variables in the problem are made SAC, and further, that only some of the values in the domains are SAC.

Therefore, we present three partial forms of SAC. The first is Bound-SAC where the first and last values in the domains of variables are SAC, and all other domain values are arc-consistent. The second level of SAC follows on immediately and we call it First-SAC, where the first value in the domain of a variable is SAC and all other values are AC. Finally we present Existential-SAC ($\exists$-SAC), where we guarantee that some value in the domain is SAC and all others are AC. These different levels of consistency can then be maintained on different sets of variables within a problem. For example when modelling a problem we might maintain SAC on one set of variables, Bound-SAC on another set of variables, and AC on the remaining variables. That is, we might use varying levels of consistency across different parts of a problem, attempting to find a good balance between inference and exploration. It is related to what is called *mixed-consistency* in [10] and *hybrid-consistency* in [4]. We then show how such a feature might be engineered into a solver so that a constraint programmer can control the mix of consistency and we present an empirical study that shows how this can be put to good effect.

The paper is organized as follows. We start by introducing some partial forms of Singleton Arc-Consistency and show their relations. We then show how to incorporate SAC, and its partial forms, into an object-oriented constraint programming toolkit. Next, we present the analysis and results of empirical studies on random, scheduling and Golomb ruler problems. A new algorithm (using a greedy approach) that checks if a constraint network is Existential-SAC is then presented along with an empirical study. Finally we conclude.

[3] However, one exception is the Quick Shaving approach of Lhomme [16]. The Quick Shaving principle is to test when backtracking occurs at depth k the consistency of values that were shavable (i.e. singleton arc inconsistent) at depth $k + 1$. Filtering is operational (i.e. a feature of the search algorithm) and does not correspond to a property of the constraint network.

[4] In a sense this part of our work is then somewhat in the spirit of [20] where MAC was compared to forward checking. Now we compare maintaining SAC against maintaining AC.

2 Partial Singleton Arc Consistencies

In this section, we introduce some technical background about constraint networks and consistencies. In particular, we introduce three partial forms of Singleton Arc-Consistency called Bound-SAC, First-SAC and Existential-SAC.

A (finite) Constraint Network (CN) P is a pair $(\mathscr{X}, \mathscr{C})$ where $\mathscr{X}$ is a finite set of variables and $\mathscr{C}$ a finite set of constraints. Each variable $X \in \mathscr{X}$ has an associated domain, denoted *dom*(X), which contains the set of values allowed for X. Each constraint $C \in \mathscr{C}$ involves a subset of variables of $\mathscr{X}$, called scope and denoted *vars*(C), and has an associated relation, denoted *rel*(C), which contains the set of tuples allowed for the variables of its scope. We will respectively denote the number of variables and constraints of a CN by n and e. For any variable X, min(X) and max(X) represents the smallest and greatest values in *dom*(X). Note that a value will usually refer to a pair (X,a) with $X \in \mathscr{X}$ and $a \in$ *dom*(X). We will note $(X, a) \in P$ (respectively, $(X, a) \notin P$) iff $X \in \mathscr{X}$ and $a \in$ *dom*(X) (respectively, $a \notin$ *dom*(X)).

A CN is said to be satisfiable iff it admits at least one solution. The Constraint Satisfaction Problem (CSP) is the NP-complete task of determining whether a given CN, also called CSP instance, is satisfiable. To solve a CSP instance, a depth-first search algorithm with backtracking can be applied, where at each step of the search, a variable assignment is performed followed by a filtering process called constraint propagation. Usually, constraint propagation algorithms are based on domain filtering consistencies [9], among which the most widely studied ones are called arc-consistency, max-restricted path consistency and singleton arc-consistency. Arc-Consistency (AC) is the basic property of CNs. It guarantees that each value admits at least one support in each constraint.

Definition 1. *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN. A pair (X, a), with $X \in \mathscr{X}$ and $a \in dom(X)$, is arc consistent (AC) iff $\forall C \in \mathscr{C} \mid X \in vars(C)$, there exists a support of (X, a) in C, i.e., a tuple $t \in rel(C)$ such that $t[X] = a$ and $t[Y] \in dom(Y)$ $\forall Y \in vars(C)$[5]. A variable $X \in \mathscr{X}$ is AC iff dom(X) $\neq \emptyset$ and $\forall a \in dom(X)$, (X, a) is AC. P is AC iff $\forall X \in \mathscr{X}$, X is AC.*

Singleton Arc-Consistency (SAC) is a stronger consistency than AC. It means that SAC can identify more inconsistent values than AC can. SAC guarantees that enforcing arc-consistency after performing any variable assignment does not show unsatisfiability, i.e., does not entail a domain wipe-out. To give a formal definition of SAC, we need to introduce some notations. AC(P) denotes the CN obtained after enforcing arc-consistency on a given CN P. AC(P) is such that all values of P that are not arc consistent have been removed. If there is a variable with an empty domain in AC(P), denoted AC(P) = $\bot$, then P is clearly unsatisfiable. $P|_{X=a}$ is the CN obtained from P by restricting the domain of X to $\{a\}$.

Definition 2. *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN. A pair (X, a), with $X \in \mathscr{X}$ and $a \in dom(X)$, is singleton arc consistent (SAC) iff AC($P|_{X=a}$) $\neq \bot$. X is SAC iff $\forall a \in dom(X)$, (X, a) is SAC. P is SAC iff $\forall X \in \mathscr{X}$, X is SAC.*

[5] $t[X]$ denotes the value assigned to X in t.

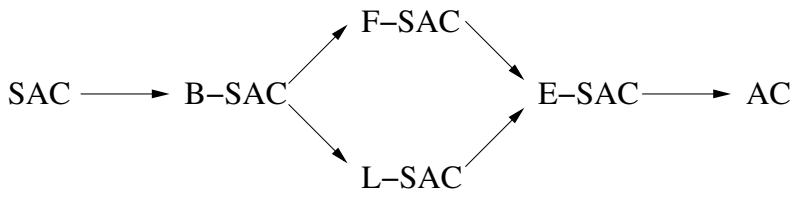


Fig. 1. Relationships between consistencies. $A \rightarrow B$ means consistency A is stronger than B

A consistency ϕ is stronger than a consistency λ iff whenever ϕ holds on a CN, λ holds too. ϕ is strictly stronger than λ iff ϕ is stronger than λ and there exists a CN on which λ holds and ϕ does not hold. It is possible to define partial forms of SAC (i.e. consistencies weaker than SAC) still stronger than AC by restricting SAC to some values.

Definition 3. *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN.*

- *P is First-SAC iff $\forall X \in \mathscr{X}$, X is AC and min(X) is SAC.*
- *P is Last-SAC iff $\forall X \in \mathscr{X}$, X is AC and max(X) is SAC.*
- *P is Bound-SAC iff P is both First-SAC and Last-SAC.*
- *P is Existential-SAC iff $\forall X \in \mathscr{X}$, X is AC and $\exists b \in dom(X)$ s.t. (X,b) is SAC.*

Figure 1 shows the relations existing between the consistencies introduced just above, SAC and AC. An arrow from a consistency ϕ to another consistency λ indicates that ϕ is strictly stronger than λ.

It is natural to conceive algorithms to enforce First-SAC, Last-SAC and Bound-SAC on CNs. Indeed, it suffices to remove all values detected as arc inconsistent and bound values (only the minimal ones for First-SAC and the maximal ones for Last-SAC) detected as singleton arc inconsistent. When enforcing a CN P to be First-SAC, Last-SAC or Bound-SAC, one then obtains the greatest sub-network of P which is First-SAC, Last-SAC or Bound-SAC. As a consequence, if a consistency ϕ is stronger than another consistency λ, then it means that all values removed when enforcing λ on a given network are also removed when enforcing ϕ [9].

In fact, this last statement is true for all (known) consistencies, except for Existential-SAC. Indeed, enforcing Existential-SAC on a CN is meaningless. Either the network is (already) Existential-SAC, or the network is singleton arc inconsistent. It is then better to talk about checking Existential-SAC. An algorithm to check Existential-SAC will have to find a singleton arc consistent value in each domain. As a side-effect, if singleton arc inconsistent values are encountered, they will be, of course, removed. However, we have absolutely no guarantee about the network obtained after checking Existential-SAC due to the non-deterministic nature of this consistency.

3 Maintaining Singleton Arc Consistencies

We now show how SAC can be incorporated into a constraint programming toolkit so that the search process maintains SAC on a specified set of variables. This then leads us

to naturally introduce Bound-SAC and First-SAC. These different levels of consistency (AC, SAC, Bound-SAC, First-SAC) can then be applied selectively across different sets of variables in a problem by the constraint programmer, allowing the programmer to control the blend of mixed-consistency maintained during search. We use the JChoco constraint programming toolkit [12] to demonstrate this.

3.1 Engineering SAC into JChoco

JChoco [12] is a freely available constraint programming toolkit, using the java programming language. Most of the methods used to model and solve problems are called via the $Problem$ class. Constrained variables (be they enumerated, bound, real, or set variables) are added to a problem, and constraints between those variables are posted to it. A problem instance (i.e. an object of class $Problem$) can then be made arc-consistent via the $propagate$ method and solutions found via the $solve$ method. Therefore, in order to incorporate SAC into JChoco we merely produce a new subclass of $Problem$ called $SacProblem$ and over-ride the $propagation$ method. All the $Problem$ methods are inherited and we can then use the constraint toolkit as usual, but rather than maintaining AC, we maintain SAC.

The java code for this is shown below. The boolean method $isSac$ determines if a value a for a variable x is SAC. The method $propagate$ now maintains SAC rather than AC, and the method call $super.propagate()$ is a call to the inherited arc-consistency algorithm used within JChoco. Therefore, if constraints are expressed explicitly as tuples (allowed or disallowed), JChoco will use the optimal algorithm reported in [5], and if a specialised constraint is used, then the appropriate specialised propagator will be applied. The code for the method $propagate$ below should be compared to the procedure $SingletonAC$ given in [8] and the SAC1 procedure in [1]. Our java code is a straightforward translation of these procedures. However, the complexity of this procedure is $O(en^2d^4)$ [19], clearly far worse than the optimal $O(end^3)$ [3].

```
public class SacProblem extends Problem {

     private boolean isSac(IntVar x,int a) throws ContradictionException {
        boolean consistent = true;
        worldPush();
        try{x.setVal(a);super.propagate();}
        catch (ContradictionException e) {consistent = false;}
        worldPop();
        return consistent;
     }

     public void propagate() throws ContradictionException {
        super.propagate();
        boolean change = true;
        while (change) {
           change = false;
           for (int i=0;i<getNbIntVars();i++){
              IntVar x = getIntVar(i);
              IntDomain d = x.getDomain();
              IntIterator domIter = d.getIterator();
              while (domIter.hasNext()){
                 int a = domIter.next();
                 if (!isSac(x,a))
                     {x.remVal(a);change = true;super.propagate();}
              }
           }}}}
```

We believe that SAC can be similarly incorporated in other constraint toolkits that take an object oriented approach (e.g. Koalog's constraint solver [11]). Therefore, we expect that the above engineering approach could be quite generic. However, one obvious limitation of the $SacProblem$ class above is that it will only work on variables with enumerated domains. How can we handle bound integer variables?

3.2 Bound-SAC and First-SAC

In [15], an algorithm is proposed for establishing Bound-SAC. Bound-SAC means that the first and last values in any domain is SAC while all other values are arc consistent. Again we can produce yet another subclass of $Problem$ which we might call $BoundSacProblem$ with a $propagate$ method as shown below.

```
public void propagate() throws ContradictionException {
   super.propagate();
   boolean change = true;
   while (change) {
      change = false;
      for (int i=0;i<getNbIntVars();i++){
         IntVar x = getIntVar(i);
         if (x.getDomainSize()>1){
            while (!isSac(x,x.getInf()))
               {x.remVal(x.getInf());change = true;super.propagate();}
            while (!isSac(x,x.getSup()))
               {x.remVal(x.getSup());change = true;super.propagate();}
         }
      }}}
```

The method call $x.getInf()$ above gets the lower bound of x and $x.getSup()$ gets the upper bound of x. The inner $while$ loops find respectively the smallest and largest SAC values in the domain of x. In addition we can decide to only make the first value in the domain SAC, and we call this First-SAC. To engineer First-SAC all that need be done is to delete the second inner $while$ loop in the code above.

3.3 Mixed-Consistency

If we adopt the implementations above we are then in the position that we can either model problems with enumerated domains or bound domains, but not both. An obvious engineering fix is to be able to detect inside the $propagate$ method the class of variable and then either apply AC, SAC, Bound-SAC or First-SAC. Another approach is to associate three lists with our $SacProblem$: one for enumerated variables to be made SAC, another for enumerated and bound variables to be made Bound-SAC, and a third for enumerated and bound variables to be made First-SAC. Any other variables will be made arc-consistent due to the default call to the AC propagator via $super.propagate()$. This is what we in fact do (but don't show), and have an additional method such that we explicitly add to a $SacProblem$ the variables to be made SAC, Bound-SAC, and First-SAC. This then allows the programmer to blend the level of mixed-consistency across variables in a problem.

4 Empirical Studies

We now present experimental studies showing the effects of maintaining different levels of SAC during search. First, we investigate problems with no structure (random problems), then we look at problems with obvious structure and demonstrate the effect of blending mixed-consistency.

4.1 A Study of Maintaining Levels of SAC on Random Problems

First, we study the effect of maintaining SAC during search on random instances of the class $\langle 20, 10, 0.5 \rangle$ (i.e. problems with 20 variables, each with domain size 10, with a probability of 0.5 that there is a constraint between a pair of variables), answering the decision problem "Is there a solution?". Our problems are modelled in JChoco using enumerated integer variables, constraints represented as allowed tuples, and arc-consistency achieved via the optimal coarse grained algorithm proposed in [5]. We compare MAC against different restrictions of SAC. We realise MAC within our framework as a problem with no SAC or Bound-SAC variables. Consequently, in the $propagate$ method only the call $super.propagate()$ is made. The next experiment is of SAC, where all the values in the domains of variables are maintained singleton arc-consistent. In our third experiment all the variables are maintained Bound-SAC. Finally, we maintain First-SAC on all variables.

Experiments were performed on a domestic machine with a 2.79 GHz processor, with 512 MB RAM, and Windows XP. Measurements were taken of average runtime in milliseconds and the number of nodes explored. We could not measure consistency checks, as is the norm, as these are not available within the JChoco toolkit. However, we consider run times to be as reliable and meaningful a measure as consistency checks.

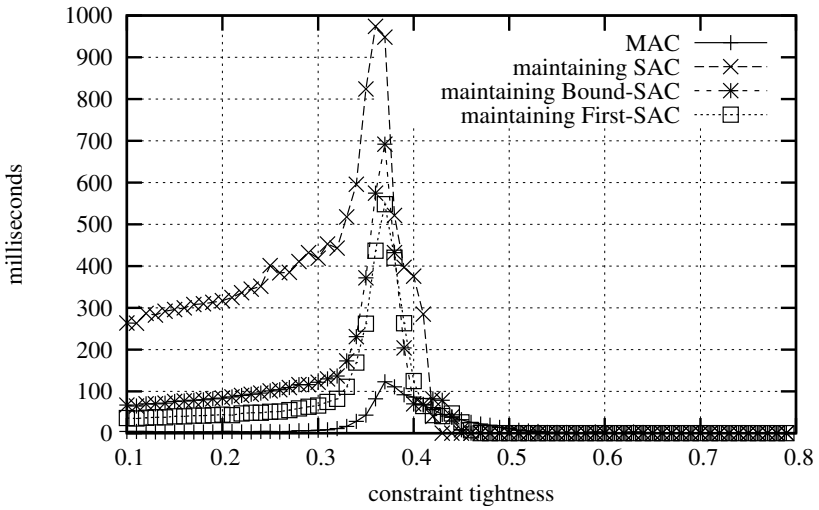


Fig. 2. Average CPU time in milliseconds for $\langle 20, 10, 0.5 \rangle$

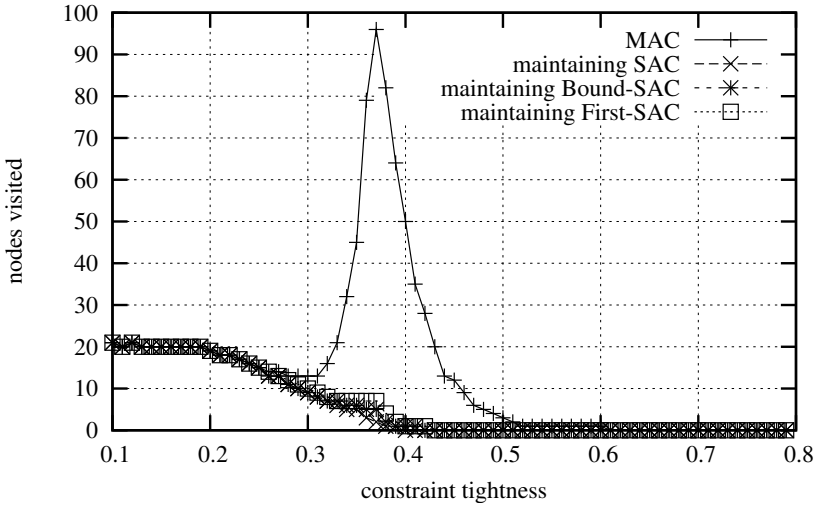


Fig. 3. Average number of nodes visited for $\langle 20, 10, 0.5 \rangle$

In Figure 2, we show the average run time in milliseconds to answer the decision problem, with a sample size of 100. We see the familiar complexity peak at the phase transition, with MAC outperforming the other consistency levels in all but the easy insoluble region ($p_2 > 0.45$). However, we do see that in the easy soluble region ($p_2 < 0.3$) Bound-SAC and First-SAC perform quite well, at least when compared to SAC. In previous studies, it was shown that SAC was a very expensive preprocess in the easy soluble region, and now we see quite acceptable costs for Bound and First-SAC, at least when applied to the decision problem.

Figure 3 shows the average number of nodes visited. We see that there is very little difference in the number of nodes visited between our three versions of SAC. MAC again shows typical phase transition behaviour, with a complexity peak at the crossover point. However the SAC algorithms do not show this behaviour, but instead a gradual fall in nodes visited as we increase constraint tightness. Therefore, we do continue to see a complexity peak in terms of runtime, but this takes place within the SAC algorithms, i.e. it takes longer to reach the SAC fixed point as we approach the phase transition.

This raises an interesting question: if we do not see a complexity peak in the size of the search tree, and the cost of SAC is polynomial, will we actually fail to see a complexity peak as problems get larger? Put another way, will search cost scale polynomially at the phase transition? Of course, our intuition suggests that the answer would be no, and that nodes would rise again. However, to answer (at least, partially) this question, we have investigated the problem classes $\langle 20, 20, 0.5 \rangle$ and $\langle 50, 10, 0.1 \rangle$ using JChoco and abscon [14]. The results were similar to those for $\langle 20, 10, 0.5 \rangle$ with MAC being dominant in runtime. However in the class $\langle 20, 20, 0.5 \rangle$ a small but noticeable complexity peak in nodes visited begins to emerge whilst maintaining SAC.

4.2 A Study of Mixed-Consistency Applied to Scheduling Problems

We performed experiments on 15 of the Lawrence Job-Shop scheduling instances, la01 to la15, available at ORLIB. The instances la01 to la05 are 10×5 (i.e. 10 jobs and 5 resources), la06 to la10 are 15×5, and la11 to la15 are 20×5. Experiments were performed to determine if any particular blend of SAC was beneficial with respect to the quality of solution found when CPU time was bounded. Experiments were run on a 1.3 GHz machine with 256 MB RAM. CPU time was limited to 600 seconds (10 minutes) on each instance. The scheduling problems were represented conventionally, as a disjunctive graph. That is, for a job-shop instance with n jobs and m resources there would be $m.n(n-1)/2$ zero/one variables to decide the order of operations on resources and $(n.m+1)$ bound integer variables to represent operation start times along with the optimisation variable. Therefore we have two distinct sets of variables: the set of 0/1 variables that control disjunctive precedence constraints on resources and the set of start times attached to operations. Consequently, this is a good model to explore the effects of mixed-consistency, i.e. we can maintain different levels of consistency across different sets of variables. The problem specific objective is to find the schedule that minimises the makespan. The results of four of our experiments are shown in Table 1.

The first experiment used MAC (all variables were maintained arc-consistent) and is tabulated as column MAC. Again, MAC was realised by using our SAC solver but with an empty list of variables to make SAC. The second experiment used Bound-SAC on the 0/1 decision variables (it then corresponds to use SAC) and MAC on all other variables, and this is column B-SAC$_{dn}$. Experiment three maintains Bound-SAC on the start times of operations and the optimisation variable, and this is column B-SAC$_{st}$. Finally, in experiment four, all variables are made Bound-SAC, and this is column B-SAC. In all the experiments, the search variables were the 0/1 decision variables. In Table 1, we report the cost of the best solution found within the CPU time limit, and a entry of $-$ signifies that no solution was found in the time limit.

What we see is that Bound-SAC can indeed be beneficial, allowing us to frequently find better solutions than just using MAC on its own. In particular, the B-SAC$_{dn}$ results show that more often than not Bound-SAC on the decision variables alone results in significantly lower makespans than does MAC. However, too much SAC appears to be a bad thing. In experiments B-SAC$_{st}$ and B-SAC we see that too much time is spent in SAC processing compared to time spent in search. Consequently solution quality suffers. In fact, as instance size increases from la11 onwards no solutions were found as all the CPU time was spent in SAC and none in search.

These experiments have demonstrated that a small amount of SAC can be a good thing. But this raises the question: why? In experiment B-SAC$_{dn}$, we maintain Bound-SAC on the 0/1 decision variables. This might be thought of as a weak form of edge-finding [7], i.e. attempting to determine what operations must come first or last on a resource. In experiment B-SAC$_{st}$ we maintain Bound-SAC on the start times of operations, and this in turn is similar to shaving [18]. And finally, in experiment B-SAC we are maintaining weak edge-finding and shaving, but at the expense of reduced exploration.

Instance	MAC	Maintaining		
		B-SAC$_{dn}$	B-SAC$_{st}$	B-SAC
la01	666	666	666	666
la02	655	655	655	655
la03	653	597	603	603
la04	628	598	590	590
la05	593	665	665	665
la06	1245	1146	1233	1237
la07	1214	897	1336	1359
la08	1161	1084	1400	1393
la09	1498	1049	1527	1520
la10	1658	972	1192	1259
la11	1453	1787	–	–
la12	1467	1504	–	–
la13	2899	2310	–	–
la14	1970	1784	–	–
la15	2368	2200	–	–

Table 1. Cost of best solution found for Lawrence scheduling instances, given 10 minutes CPU.

4.3 A Study of Mixed-Consistency on Golomb Rulers

In [19], experiments were performed on Golomb rulers. In particular, given the length l of the shortest ruler with n ticks (or marks), the objective is to find that ruler and prove it optimal. The study showed that SAC preprocessing and restricted SAC preprocessing could lead to a modest reduction in run-times. We repeat those experiments, but now maintain a mix of SAC during the search process.

The problem was represented in JChoco using n $tick$ variables with enumerated domains whose values range from 0 to l, and, in addition $n(n-1)/2$ $diff$ variables with similar domains. Constraints posted to the problem are: $diff[i][j] = tick[j] - tick[i]$ and $tick[i] < tick[j]$ for any pair (i,j) such that $1 \leq i < j \leq n$, and a $boundAllDiff$ constraint enforcing all the $diff$ variables to be different. The $tick$ variables are the decision variables and these were instantiated in a static lexicographic order. Again we have a problem with two obviously different sets of variables, the $tick$ variables and the $diff$ variables, and this again gives us an opportunity to investigate the effects of blending mixed-consistency. Our experiments were run on a 3.01 GHz processor with 512 MB of RAM using Windows XP.

The results of the experiments are given in Table 2 which clearly shows that maintaining Bound-SAC on the $tick$ variables (denoted B-SAC$_{tk}$) dominates MAC, whereas maintaining SAC on the $tick$ variables (denoted SAC$_{tk}$) is far too expensive. We also experimented with maintaining restricted Bound-SAC on the $tick$ variables (denoted RB-SAC$_{tk}$), i.e. the $propagate$ method for Bound-SAC was edited such that the outer $while(change)...$ loop was deleted, consequently only a single pass is made over the variables. This is the same as the restriction proposed in [19]. Table 2 shows that this results in our best performance.

Although not tabulated, we also investigated maintaining SAC, Bound-SAC, and First-SAC on all the variables ($tick$'s and $diff$'s) but run-times did not compete with

Instance	MAC	Maintaining		
		SAC_{tk}	B-SAC_{tk}	RB-SAC_{tk}
5/11	0.01 (5)	0.12 (3)	0.08 (3)	0.08 (3)
6/17	0.1 (18)	0.27 (5)	0.14 (5)	0.14 (5)
7/25	0.47 (116)	0.81 (6)	0.30 (7)	0.34 (11)
8/34	3.6 (904)	14.6 (19)	1.8 (23)	1.6 (33)
9/44	29.1 (5502)	136 (62)	11.3 (68)	9.6 (103)
10/55	217.3 (30097)	1075 (218)	68.8 (245)	59 (479)
11/72	7200 (773560)	—	5534 (11742)	4645 (20056)

Table 2. The runtime in seconds (and in brackets number of nodes visited) to find and prove optimal a Golomb ruler n/l, with n ticks of length l. Restricted Bound-SAC on the $tick$ variables (RB-SAC_{tk}) is fastest.

MAC over all instances. Also, First-SAC on the $tick$ variables was competitive with MAC except on the largest problem $11/72$ taking 12371 seconds and 41334 nodes, much slower than MAC.

Some of the experiments were also repeated but using a different model, i.e. we replaced the $boundAllDiff$ constraint with a clique of not-equals constraints. In this model MAC was dominant, typically running three times or more faster than Bound-SAC on the $tick$ variables. Similar behaviour was noted over the quasigroup completion problems in [19]. This is due to the weak propagation of the not-equals constraint, and that values will tend to be SAC until the domain of an adjacent variable is reduced to a singleton. Therefore, we see that when maintaining SAC, we not only have to consider the level of SAC to maintain and the variables over which to maintain that level, but also the model itself.

5 Checking Existential SAC

Existential-SAC is the weakest (see Figure 1) partial form of SAC that we have introduced. We now propose an algorithm to check existential-SAC and we present some empirical results.

5.1 ∃-SAC3

We have presented limited forms of SAC on the basis of the most simple algorithm, SAC1 [8]. SAC1 checks the singleton arc-consistency of all variables whenever a singleton arc-inconsistent value is detected and removed. Assuming an underlying optimal arc-consistency algorithm, worst-case space and time complexities of SAC1 are respectively O(ed) and O(en^2d^4).

In [13], an original approach to establish SAC has been proposed. The principle of this is to perform several runs of a greedy search, where at each step arc-consistency is maintained. As a result, the incrementality of arc-consistency algorithms is exploited but in a different manner to that proposed in [3]. Unfortunately, a bound-SAC version of this approach does not seem to be feasible. Indeed, the main goal is to build

Algorithm 1 buildBranch()

```
 1: branchSize ← 0
 2: P_before ← P
 3: repeat
 4:    pick and remove X from Q
 5:    select a value a ∈ dom(X)
 6:    P ← AC(P|_{X=a},{X})
 7:    if P = ⊥ then
 8:       add X to Q
 9:    else
10:       branchSize ← branchSize + 1
11:    end if
12: until P = ⊥ ∨ Q = ∅
13: P ← P_before
14: if branchSize = 0 then
15:    remove a from dom(X)
16:    P ← AC(P,{X})
17:    Q ← {X | X ∈ 𝒳}
18: end if
```

Algorithm 2 E-SAC-3($P = (\mathscr{X}, \mathscr{C})$: CN)

```
1: P ← AC(P,𝒳)
2: Q ← {X | X ∈ 𝒳}
3: while P ≠ ⊥ ∧ Q ≠ ∅ do
4:       buildBranch()
5: end while
```

branches (corresponding to greedy runs) as long as possible in order to benefit from incrementality, and potentially to find solutions during inference. When we are exclusively maintaining Bound-SAC via this approach the resultant propagation branches tend to be short, and therefore uneconomical. However, using a greedy approach to check Existential-SAC seems to be quite appropriate. In particular, it is straight forward to adapt the algorithm SAC3 [13] to guarantee $\exists$-SAC. As mentioned in Section 2, such an algorithm can generate different constraint networks depending on the order that variables and values are considered i.e. it might have multiple fixed points.

Below, we give the description of this new algorithm, denoted $\exists$-SAC3. It is given in the context of using an underlying coarse-grained arc-consistency algorithm such as AC2001/3.1 [5]. But first, we introduce some notations. If $P = (\mathscr{X}, \mathscr{C})$, then AC($P$,$Q$) with $Q \subseteq \mathscr{X}$ means enforcing arc-consistency on P from the given propagation set Q. For a description of AC, see, for instance, the function $propagateAC$ in [3]. Q is the set of variables whose existential consistency must be checked. Finally, an instruction of the form $P_{before} \leftarrow P$ should not be systematically considered as a duplication of the problem. Most of the time, it correspond to store or restore the domain of a network (and the structures of the underlying arc-consistency algorithm)

Algorithm 2 starts by enforcing arc-consistency on the given network (line 1). Then, all variables are put in the structure Q (line 2) and in order to check Existential-SAC,

successive branches are built (line 4). The process continues until Existential-SAC is checked, or singleton arc inconsistency detected (line 3). Algorithm 1 allows building a branch by performing successive variable assignments while maintaining arc-consistency (line 4 to 6). When an inconsistency is detected or the set Q becomes empty, the greedy run is stopped (line 12). If the branch is of size 0 (line 14), we have to manage the removal of a value (since it is singleton arc inconsistent), to reestablish arc-consistency and to restart checking Existential-SAC from scratch (line 17).

Space required specially by ∃-SAC3 is O(n) since the only structure introduced is Q which is O(n). The time complexity of ∃-SAC3 is that of SAC3, that is to say O(bed^2) where b denotes the number of branches built by the algorithm (using an optimal AC algorithm such as AC2001, each branch built is O(ed^2) due to the incrementality of AC2001). In the best case, only one branch will be built (leading then directly to a solution), and then we obtain O(ed^2). In the worst-case, before detecting a singleton arc inconsistent value, $n-1$ branches of size 1 can be built. As the number of values that can be removed is O(nd), we obtain O(en^2d^3). Finally, when no inconsistent value is detected, the worst-case time complexity of ∃-SAC3 is O(end^2).

5.2 Experimental Results

We believe that it is worth studying the effect of maintaining ∃-SAC on satisfiable instances using ∃-SAC3, as due to greedy runs solutions can be found at any step of the search. This is illustrated in Table 3 with some instances of the n-queens problem (we only searched the first solution). These instances were modelled (with binary constraints) in abscon [14] and run on a PC Pentium IV 2.4GHz 512MB RAM under Linux. AC2001 was used as the underlying AC algorithm and dom/wdeg [6] as the variable ordering heuristic. We also show results for forward checking (FC), maintaining arc-consistency (MAC), first-SAC (F-SAC), bound-SAC (B-SAC), and SAC maintained using the SAC1 algorithm. It is interesting to note that for all these satisfiable instances, maintaining SAC3 or ∃-SAC3 explore no more than 2 nodes. However, one should expect to find less impressive results with unsatisfiable instances. To check this, we have tested, using abscon, some difficult (modified) unsatisfiable instances of the Radio Link Frequency Assignment Problem that came from the CELAR (Centre electronique de l'armement). Here, we do not consider optimisation, but only satisfaction. These instances were used as benchmarks for the first CSP solver competition and can be downloaded at `http://cpai.ucc.ie/05/Benchmarks.html`. In Table 3, it appears that maintaining SAC3 or ∃-SAC3 really limits the number of nodes that have to be visited. It can be explained by the fact that both algorithms learn from failures (of greedy runs) as the employed heuristic is $dom/wdeg$.

We then compared maintaining ∃-SAC to MAC on the full set of 1064 instances in the benchmark suite. When counting the number of solved instances within 10 minutes, MAC outperforms ∃-SAC3 by 60 instances when using the $dom/wdeg$ heuristic and by only 23 instances with the dom heuristic. However, regardless of heuristics, ∃-SAC3 behaves relatively poorly on random problems with MAC dominating on the majority of instances in series frb (random instances forced to be satisfiable) and $random$-$\{23, 24, 25\}$. Interestingly, MAC and ∃-SAC3 behave quite differently on different series. One example is all the instances in $series5$ to $series40$ where ∃-SAC dominates.

Instance	FC	MAC	Maintaining				
			F-SAC	B-SAC	SAC1	SAC3	∃-SAC3
100-queens (sat)	0.5 (194)	4.2 (118)	267 (101)	421 (101)	–	17.4 (0)	18.9 (2)
110-queens (sat)	–	–	–	–	–	37.9 (0)	22.7 (1)
120-queens (sat)	–	1636 (323K)	–	–	–	16.7 (0)	47.3 (2)
scen11-f12 (unsat)	69.1 (18K)	3.6 (695)	63.3 (60)	110 (48)	1072 (41)	418 (5)	48.3 (30)
scen11-f10 (unsat)	131 (34K)	4.4 (862)	84.4 (70)	140 (55)	1732 (52)	814 (8)	38.3 (25)
scen11-f8 (unsat)	260 (66K)	67.8 (14K)	1660 (2K)	–	–	–	290 (213)

Table 3. CPU time (and number of visited nodes) for instances of the n-queens and the RLFAP, given 30 minutes CPU.

6 Conclusion

We have taken what is probably an unusual step, reporting on how we can engineer the least efficient version of the SAC algorithm into an actual constraint programming toolkit. By doing this we have been able to perform the first investigation of the behaviour of maintaining SAC within the search process. This has led us to proposing three new levels of SAC, i.e. Bound-SAC, First-SAC and ∃-SAC. We have also allowed ourselves to specify the set of variables in a problem that we make full, bound or first SAC, i.e. we have shown how the programmer can produce a blend of mixed-consistencies and we have shown empirically the effect this can have on runtime performance. We have shown that maintaining the right blend of consistencies can result in significant performance improvements.

When maintaining SAC in small random problems we see a peculiar signature when measuring the size of the search tree (nodes) as we pass through the phase transition. All of our SAC algorithms do not show a complexity peak. We also note that the size of the search tree is relatively insensitive to the amount of restriction put upon SAC, and that when problems are easy and soluble First-SAC and Bound-SAC perform remarkably well. This is one area where earlier studies have shown that SAC is nothing but an expense. For larger random problems, our preliminary study suggests that the size of the search tree should again exhibit a complexity peak, provided that the size of the problems is sufficiently large.

In the job-shop scheduling problem, restrictions on SAC have been beneficial, leading us to better solutions than MAC when CPU time is limited. One explanation is that our restrictions allow us to emulate a weak form of edge-finding or shaving, and that we can combine both of these. However, this has to be used with caution; we need to consider just what variables will benefit from SAC (and that was the 0/1 decision variables).

The Golomb ruler experiments show that we also need to take into consideration how we model the problem. In a model with weakly propagating constraints, values will tend to be SAC and SAC processing will tend to be nothing but an expense. However, with a good model and a well chosen level of SAC (Bound-SAC on the decision variables) we were able to outperform the gold standard, a MAC solver using state of the art constraint propagation algorithms.

The results are surprising when we consider that when using a basic sub-optimal algorithm for SAC we can frequently beat MAC. From an abstract point of view, we have demonstrated that rather than using the same level of inference (maintaining arc-consistency) all the time (during search) everywhere (over all the variables) we can often do much better by varying the level of inference (mixing the consistency levels AC, SAC, Bound-SAC, First-SAC) and doing this over only parts of the problem (a subset of the variables). Finally, we have introduced a simple and efficient implementation of an algorithm that checks $\exists$-SAC. Empirical results suggests that this represents a promising generic approach.

References

1. R. Bartak and R. Erben. A new algorithm for Singleton Arc Consistency. In *Proceedings of FLAIRS'04*, 2004.
2. C. Bessiere and R. Debruyne. Theoretical analysis of Singleton Arc Consistency. In *Proceedings of MSPC'04 workshop held with ECAI'04*, pages 20–29, 2004.
3. C. Bessiere and R. Debruyne. Optimal and suboptimal Singleton Arc Consistency algorithms. In *Proceedings of IJCAI'05*, pages 54–59, 2005.
4. C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. The range and roots constraints: Specifying counting and occurrence problems. In *Proc. of IJCAI'05*, pages 60–65, 2005.
5. C. Bessiere, J.C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained Arc Consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.
6. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.
7. J. Carlier and E. Pinson. A practical use of Jackson's preemptive schedule for solving the Job-Shop problem. *Annals of Operations Research*, 26:269–287, 1990.
8. R. Debruyne and C. Bessiere. Some practicable filtering techniques for the Constraint Satisfaction Problem. In *Proceedings of IJCAI'97*, pages 412–417, 1997.
9. R. Debruyne and C. Bessiere. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.
10. G. Dooms, Y. Deville, and P. Dupont. CP(Graph): Introducing a Graph Computation Domain in Constraint Programming. In *Proceedings of CP'005*, pages 211–225, 2005.
11. Y. Georget and M. Philip. The Koalog Constraint Solver. `http://www.koalog.com`.
12. F. Laburthe and N. Jussien. Jchoco: A java library for constraint satisfaction problems. `http://choco.sourceforge.net`.
13. C. Lecoutre and S. Cardon. A greedy approach to establish Singleton Arc Consistency. In *Proceedings of IJCAI'05*, pages 199–204, 2005.
14. C. Lecoutre and F. Hemery. Abscon 2006. `http://www.cril.univ-artois.fr/~lecoutre`.
15. C. Lecoutre and J. Vion. Bound consistencies for the discrete CSP. In *Proceedings of CPAI'05 workshop held with CP'05*, pages 17–31, 2005.
16. O. Lhomme. Quick shaving. In *Proceedings of AAAI'05*, pages 411–415, 2005.
17. A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.
18. P. Martin and D.B. Shmoys. A new approach to computing optimal schedules for the Job-Shop scheduling problem. In *Proceedings of IPCO'96*, pages 389–403, 1996.
19. P. Prosser, K. Stergiou, and T. Walsh. Singleton Consistencies. In *Proceedings of CP'00*, pages 353–368, 2000.
20. D. Sabin and E. Freuder. Contradicting conventional wisdom in Constraint Satisfaction. In *Proceedings of ECAI'94*, pages 125–129, 1994.

Probabilistic Singleton Arc Consistency

Deepak Mehta* and M.R.C. van Dongen

Computer Science Department, University College Cork

Abstract. Singleton Arc Consistency (SAC) is a meta-consistency that enhances the pruning capability of arc consistency. It guarantees that the network can be made arc consistent after assigning a value to any variable. Establishing singleton arc consistency, generally, prunes more inconsistent values from the domains than establishing arc consistency. However, due to much ineffective constraint propagation, it often consumes more time and can be a huge overhead. If we can reduce ineffective constraint propagation, then the performance of the algorithm can be improved significantly. In order to do so, we use a probabilistic approach to determine when to propagate and when not to. The idea is to perform only the useful consistency checking by not seeking a support when there is a high probability that a support exists. The idea of probabilistic support inference is general and can be applied to any kind of local consistency algorithm. In this paper we shall investigate its impact with respect to singleton arc consistency. Experimental results demonstrate that enforcing probabilistic SAC almost always enforces SAC but that it requires significantly less time than SAC. Likewise, maintaining probabilistic singleton arc consistency requires significantly less time than maintaining singleton arc consistency.

1 Introduction

Constraint Satisfaction Problems (CSPs) are ubiquitous in Artificial Intelligence. They involve finding values for problem variables subject to constraints. For simplicity, we restrict our attention to binary CSPs. Maintaining some levels of local consistency before and/or during backtrack search have become the de facto standard to solve CSPs, which generally reduces the thrashing behavior of a backtrack algorithm. However, as the strength of local consistency increases, so does the amount of ineffective constraint propagation, which may penalize the algorithm in terms of CPU time.

Arc consistency (AC) is the most widely used local consistency algorithm to reduce the search space of CSPs. Coarse-grained algorithms such as AC-3 [10], and AC-2001/ AC-3.1[3], are competent, when it comes to transform a problem into its arc consistent equivalent. These algorithms repeatedly carry out revisions, which require support checks for identifying and deleting unsupported values from the domain of a variable. However, for difficult problems, in many revisions, some or *all* values successfully find some support, that is to say, *ineffective constraint propagation* occurs.

Recently, Singleton Arc Consistency (SAC) [6] has been receiving much attention. SAC is a meta-consistency that enhances the pruning capability of arc consistency. It guarantees that the network can be made arc consistent after assigning a value to

* Supported by the Boole Centre for Research in Informatics.

the variable. It is a stronger consistency than arc consistency. Therefore, it can avoid much unfruitful exploration in the search-tree. Nevertheless, applying SAC in SAC-1 [6] style can invoke the underlying arc consistency algorithm n^2d^2 times in the worst-case, where n is the number of variables and d is the maximum domain size. Thus, the amount of ineffective constraint propagation will be much more in singleton arc consistency than in arc consistency. Establishing SAC before search can be expensive in terms of checks and time, and maintaining it during search can be even more expensive.

At the CPAI'2005 workshop, [11] presented a *probabilistic approach* to reduce ineffective propagation and studied it with respect to arc consistency on random problems. This probabilistic approach is to *avoid the process of seeking a support, when the probability of its existence is above some, carefully chosen, threshold*. This way a significant amount of work in terms of support checks and time can be saved.

In this paper, we study the impact of using the probabilistic approach with respect to SAC and LSAC (limited version of SAC proposed in this paper) on a variety of problems. We call the resulting algorithms Probabilistic Singleton Arc Consistency (PSAC), and Probabilistic LSAC (PLSAC). We examine the performances of PSAC and PLSAC when used as a preprocessor before search. Experimental results demonstrate that enforcing PSAC and PLSAC almost always enforces SAC and LSAC but usually require significantly less time. Finally, we investigate the impact of maintaining PSAC and maintaining PLSAC during search on various problems. Experimental results show a significant gain in terms of time on quasi-group problems. Overall, empirical results demonstrate that the original algorithms are outperformed by their probabilistic counterparts.

The remainder of this paper is organised as follows: Section 2 gives an introduction to constraint satisfaction. Section 3 describes the existing singleton arc consistency algorithms. Section 4 explains the concept of probabilistic support inference to reduce ineffective constraint propagation. Experimental results are presented in section 5 followed by conclusions in section 6.

2 Preliminaries

A *Constraint Satisfaction Problem* $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ is defined as a set $\mathcal{V}$ of n variables, a non-empty domain $D(x) \in \mathcal{D}$, for all $x \in \mathcal{V}$ and a set $\mathcal{C}$ of e constraints among subsets of variables of $\mathcal{V}$. For simplicity, we restrict our attention to binary CSPs. However, the ideas presented in this paper can be extended to non-binary CSPs. A binary constraint C_{xy} between variables x and y is a subset of the Cartesian product of $D(x)$ and $D(y)$ that specifies the allowed pairs of values for x and y. The *density* of a CSP is defined as $2\,e/(n^2 - n)$, where e is the number of constraints and n is the number of variables. The *tightness* of the constraint C_{xy} is defined as $1 - |\,C_{xy}\,|/|\,D(x) \times D(y)\,|$.

A value $b \in D(y)$ (also denoted as, (y, b)) is called a *support* for (x, a) if $(a, b) \in C_{xy}$. Similarly, (x, a) is called a support for (y, b) if $(a, b) \in C_{xy}$. A *support check* (consistency check) is a test to find if two values support each other.

A value $a \in D(x)$ is *arc consistent* if $\forall y \in \mathcal{V}$ constraining x the value a is supported by some value in $D(y)$. A CSP is *arc consistent* if and only if $\forall x \in \mathcal{V}$, $D(x) \neq \emptyset$, and $\forall a \in D(x)$, (x, a) is arc consistent. We denote the CSP $\mathcal{P}$ obtained after enforcing arc consistency as $ac(\mathcal{P})$. If there is a variable with an empty domain in $ac(\mathcal{P})$, we denote

it as $ac(\mathcal{P}) = \perp$. The CSP obtained from $\mathcal{P}$ by assigning a value a to the variable x is denoted by $\mathcal{P}|_{x=a}$. A value $a \in D(x)$ is *singleton arc consistent* if and only if $ac(\mathcal{P}|_{x=a}) \neq \perp$. A CSP is singleton arc consistent if and only if each value of each variable is singleton arc consistent.

MAC [14] is a backtrack algorithm that maintains arc consistency after every variable assignment. It ensures that each value in the domain of *each* variable is supported by at least one value in the domain of every variable by which it is constrained. FC [7] can be considered a degenerated form of MAC. It ensures that each value in the domain of each *future* variable is FC *consistent*, i.e. supported by the value assigned to every past and current variable by which it is constrained. MSAC is a backtrack algorithm that maintains singleton arc consistency after every variable assignment.

The *directed constraint graph* of a CSP is a graph having an arc (x, y) for each combination of two mutually constraining variables x and y. We will use G to denote the directed constraint graph of the input CSP. Usually, an input CSP is transformed into its arc consistent equivalent, before starting search. We call the domain of x in this initial arc consistent equivalent of the input CSP the *first arc consistent domain* of x. For the remainder of this paper for any variable x, We use $D_{ac}(x)$ for the first arc consistent domain of x, and $D(x)$ for the current domain of x.

3 Overview of SAC Algorithms

Although there are stronger consistencies than arc consistency, the standard has been to make the problem full/partial arc consistent before and/or during search. This is because applying arc consistency has a low overhead and does not change the structure of the problem. However, recently, there has been a surge of interest in SAC [13, 2, 9] as a preprocessor of MAC, i.e. making the problem singleton arc consistent before search. Various algorithms, for example, SAC-1 [6], SAC-2 [1], SAC-OPT [2], SAC-SDS [2] have been suggested to use it as a preprocessor. The advantage of SAC is that it improves the filtering power of arc consistency without changing the structure of the problem as opposed to other stronger consistencies such as k-consistency $(k > 2)$ and so on.

The first algorithm that has been proposed in [6] to establish singleton arc consistency is called SAC-1. Enforcing SAC in SAC-1 style works by having an outer loop consisting of variable-value pairs. For each (x, a) if $ac(\mathcal{P}|_{x=a}) = \perp$, then it deletes a from $D(x)$. Then it enforces arc consistency. Should this fail then the problem is not SAC. The pseudo-code for SAC-1 is presented in Figure 1. If SAC-1 uses an optimal arc consistency algorithm as the underlying arc consistency then its worst-case time complexity is $\mathcal{O}(n^2d^4e)$. If SAC-1 uses a non-optimal arc consistency algorithm as the underlying arc consistency then its worst-case time complexity is $\mathcal{O}(n^2d^5e)$. Its space complexity is same as the space complexity of the underlying arc consistency algorithm.

One problem with SAC-1 is that deleting a single value triggers the addition of all variable-value pairs in the outer loop, as shown in Figure 1. The *restricted* SAC (RSAC) algorithm proposed in [13] avoids this triggering by considering each variable-value pair only once. We propose *limited* SAC (LSAC) which lies between *restricted* SAC and SAC. The idea is that if a variable-value pair (x, a) is found arc-inconsistent, then only the pairs involving neighbours of x as a variable are added in the outer-loop.

Our experience is that LSAC is more effective than RSAC. Note that SAC and *limited* SAC will always do the same amount of work if the problem is already singleton arc consistent or if the density of the input CSP is 1.

```
Procedure SAC-1 ( P ) :Boolean;
  P ← ac(P)
  Repeat
    Changed ← false
    for each x ∈ V do
      for each a ∈ D(x) do
        if ac(P|x=a) = ⊥ then
          D(x) ← D(x) \ { a }
          P ← ac(P)
          Changed ← true
  until not Changed
```

Fig. 1. SAC-1

The second algorithm SAC-2 has been proposed in [1]. The basic idea of SAC-2 is to minimize the number of calls to the underlying arc consistency algorithm. More specifically, the idea is to consider a value (y, b) in the outerloop of SAC-1 after removing a value (x, a) only if (x, a) belongs to $ac(\mathcal{P}|_{y=b})$. This improves the average time complexity of SAC-1. However, it does not improve the worst-case time complexity of SAC-1. In fact, the worst-case space complexity increases to $\mathcal{O}(n^2d^2)$, since SAC-2 requires auxiliary data structures.

Both SAC-1 and SAC-2 are non optimal algorithms. The first optimal algorithm called SAC-OPT has been presented in [2]. The algorithm basically takes advantage of the incremental property of arc consistency. It duplicates the problem nd times, one for each value (x, a). It enforces SAC in $\mathcal{O}(end^3)$, the worst-case optimal time complexity. However, its worst-case space complexity is $\mathcal{O}(end^2)$, which prohibit its use on large constraint networks. Therefore, [2] proposed another algorithm called SAC-SDS, which represents a trade-off between time and space. The worst-case time and space complexities of SAC-SDS are $\mathcal{O}(end^4)$ and $\mathcal{O}(n^2d^2)$ respectively.

Except SAC-1, the worst-case space complexity of all the algorithms proposed so far which enforce SAC in SAC-1 style is at least $\mathcal{O}(n^2d^2)$. This amount of space can be prohibitive and may not allow to solve large problems. For example, when Lecoutre and Cardon [9] conducted experiments, SAC-SDS with a space complexity of $\mathcal{O}(n^2d^2)$ ran out of memory for GRAPH10 and GRAPH14, instances of the radio link frequency assignment problem taken from the FullRLFAP archive.

Although the space complexity of SAC-1 is relatively low ($\mathcal{O}(e + nd)$ when a non-optimal algorithm such as AC-3 is used and $\mathcal{O}(ed)$ when an optimal algorithm such as AC-2001 is used), its worst-case time complexity is huge. Thus, establishing SAC using SAC-1 can be expensive in terms of solution time. One way to overcome this is to avoid ineffective constraint propagation. The underlying arc consistency algorithm employed by SAC-1 can be invoked n^2d^2 times in the worst-case. Moreover, if the underlying arc consistency algorithm is a coarse-grained algorithm, then in each call, in many revisions, some or all values successfully find some support, that is to say, ineffective con-

straint propagation occurs. If we can reduce this ineffective constraint propagation then the average time complexity of SAC-1 can be improved significantly. We investigate the application of probabilistic support inference [11] to reduce ineffective propagation in singleton arc consistency.

4 Probabilistic Support Inference

The traditional approach to infer the existence of a support for a value $a \in D(x)$ in $D(y)$ is to *identify* some $b \in D(y)$ that supports a. This usually results in a sequence of support checks. Identifying the support is more than needed: knowing that a support exists is enough. The notions of a *support condition* (SC) and a *revision condition* (RC) were introduced in [12] to reduce the task of identifying a support up to some extent for coarse-grained arc consistency algorithms. If SC holds for (x, a) with respect to y, then it guarantees that (x, a) has *some* support in $D(y)$ and the process of identifying a support is avoided, since the probability that a support exists is *exactly* 1. If RC holds for an arc, (x, y), then it guarantees that *all* values in $D(x)$ have *some* support in $D(y)$ without identifying them, and the revision of $D(x)$ is avoided against $D(y)$, since the probability that a support exists for *each* value in the domain is *exactly* 1. In the following paragraphs, we describe the special versions of SC and RC which facilitates the introduction of their probabilistic versions.

Let C_{xy} be the constraint between x and y, let $a \in D(x)$, and let $R(y) = D_{ac}(y) \setminus D(y)$ be the values removed from $D_{ac}(y)$, i.e. the first arc consistent domain of y. The *support count* of (x, a) with respect to y, denoted $sc(x, y, a)$, is the number of values in $D_{ac}(y)$ supporting a. Note that $|R(y)|$ is an upper bound on the number of lost supports of (x, a) in y. If $sc(x, y, a) > |R(y)|$ then (x, a) is supported by y. This condition is called a special version of a support condition. For example, if $|D_{ac}(y)| = 20$, $sc(x, y, a) = 2$, and $|R(y)| = 1$, i.e. 1 value is removed from $D_{ac}(y)$, then SC holds and (x, a) has a support in $D(y)$ with a probability of 1. Hence, there is no need to seek support for a in $D(y)$.

For a given arc, (x, y), the *support count* of x with respect to y, denoted $sc(x, y)$, is defined by $sc(x, y) = \min(\{sc(x, y, a) : a \in D(x)\})$. If $sc(x, y) > |R(y)|$, then each value in $D(x)$ is supported by y. This condition is a special version of what is called a revision condition in [12]. For example, if $|D_{ac}(y)| = 20$, $sc(x, y) = 2$ and $|R(y)| = 1$ then each value $a \in D(x)$ is supported by some value of $D(y)$ with a probability of 1. Hence, there is no need to revise $D(x)$ against $D(y)$.

In the examples considered above, if $|R(y)| = 2$, then SC and RC will fail. Despite of having a high probability of the support existence for (x, a), the algorithm will be forced to search for a support in $D(y)$. Inferring the existence of a support with a high probability, but less than 1, may not always guarantee the existence of a support but can be worthwhile and may prevent many sequences of support checks ultimately leading to a support. In order to do so, the notions of a *probabilistic support condition* (PSC) and a *probabilistic revision condition* (PRC) were recently introduced in [11]. The PSC holds for (x, a) with respect to y, if the probability that a support exists for (x, a) in $D(y)$ is above some, carefully chosen, threshold. The PRC holds for an arc (x, y), if the

probability of having some support for each value $a \in D(x)$ in $D(y)$, is above some, carefully chosen, threshold.

4.1 Probabilistic Support Condition

Let $Ps_{(x,y,a)}$ be the probability that there exists some support for (x, a) in $D(y)$. If we assume that each value in $D_{ac}(y)$ is equally likely to be removed during search, then it follows that

$$Ps_{(x,y,a)} = 1 - \binom{|\, R(y)\,|}{sc(x,y,a)} / \binom{|\, D_{ac}(y)\,|}{sc(x,y,a)}. \quad (1)$$

The justification for this equation is that its right hand side is equal to the probability that none of the $\binom{|\, R(y)\,|}{sc(x,y,a)}$ subsets of size $sc(\, x, y, a\,)$ of $R(y)$ contain all supports of (x, a). Note that if $|\, R(y)\,| < sc(x, y, a)$ (a special version of support condition), then Equation (1) reduces to $Ps_{(x,y,a)} = 1$. Indeed, if fewer values have been removed from $D_{ac}(y)$ than there were supports in $D_{ac}(y)$ then the probability that a support exists is equal to 1.

Let T, $0 \leq T \leq 1$, be some desired threshold. If $Ps_{(x,y,a)} \geq T$ then (x, a) has some support in $D(y)$ with a probability of T or more. This condition is called a *Probabilistic Support Condition* (PSC) in [11]. If it holds, then the process of seeking a support for (x, a) is avoided. For example, if $T = 0.9$, $|D_{ac}(y)| = 20$, $sc(x, y, a) = 2$, and this time if $|\, R(y)\,| = 2$, then (x, a) has a support in $D(y)$ with a probability of 0.994. Thus PSC holds.

4.2 Probabilistic Revision Condition

Recall that for a given arc, (x, y), the *support count* of x with respect to y, denoted $sc(x, y)$, is defined by $sc(x, y) = \min(\{sc(x, y, a) : a \in D_{ac}(x)\})$. It is the least support count of the values of $D_{ac}(x)$ with respect to y. Let $Ps_{(x,y)}$ be the least probability of the values of $D_{ac}(x)$ that there exists some support in y, then

$$Ps_{(x,y)} = 1 - \binom{|\, R(y)\,|}{sc(x,y)} / \binom{|\, D_{ac}(y)\,|}{sc(x,y)}. \quad (2)$$

For any value $a \in D(x)$, we immediately have $Ps_{(x,y,a)} \geq Ps_{(x,y)}$. Note that when $|\, R(y)\,| < sc(x, y)$ (a special version of revision condition presented in [12]), then $\binom{|\, R(y)\,|}{sc(x,y)} = 0$, and with a probability of 1, all values in $D(x)$ are supported by y.

Let T be some threshold. If $Ps_{(x,y)} \geq T$ then, each value in $D(x)$ is supported by y with a probability of T or more. This condition is called a *Probabilistic Revision Condition* in [11]. If it holds then the revision of $D(x)$ against $D(y)$ is skipped.

4.3 PAC-3

Both PSC and PRC can be embodied in any coarse-grained AC algorithm. Figure 2 depicts the pseudocode of PAC-3, the result of incorporating PSC and PRC into AC-3 [10]. Depending on the threshold, sometimes it may achieve less than full arc consistency. If

PSC holds then the process of identifying a support is avoided. This is depicted in Figure 3. If PRC holds then it is exploited either *after* selecting the arc (x, y) for the next revision or *before* adding the arcs to the queue. In the former case the corresponding revision is not carried out and in the latter case (x, y) is not added to the queue. We will use the PRC by tightening the condition for adding arcs to the queue: arcs should only be added if the PRC does not hold. This is depicted in Figure 2.

In order to use PSC and PRC, the support count for each arc-value pair must be computed prior to search. Once these support counts are computed, they remain static. The algorithm used for computing the support counts is mentioned in [12, Figure 2]. If $T = 0$ then PAC-3 makes the problem FC consistent i.e. revise the domains of the variables *connected* to the current variable using AC-3's original *revise* function [10]. If $T = 1$ then PAC-3 makes the problem arc consistent. If $0 < T < 1$ then the level of consistency established by PAC-3 is between them. The support counters are represented in $\mathcal{O}(e\,d)$ space complexity, which exceeds the space-complexity of AC-3. Thus, the space-complexity of PAC-3 becomes $\mathcal{O}(e\,d)$. The worst-case time complexity of PAC-3 is $\mathcal{O}(e\,d^3)$.

```
Function PAC-3(var current_var) :Boolean;
begin
  Q := G
  while Q not empty do begin
    select any x from {x : (x, y) ∈ Q }
    effective_revisions := 0
    for each y such that (x, y) ∈ Q do
      remove (x, y) from Q
      if y = current_var then
        revise(x, y, change_x)
      else
        revise_p(x, y, change_x)
      if D(x) = ∅ then
        return False
      else if change_x then
        effective_revisions := effective_revisions + 1
        y'' := y;
    if effective_revisions = 1 then
      Q := Q ∪ { (y', x) ∈ G : y' ≠ y'', Ps_(y',x) < T }
    else if effective_revisions > 1 then
      Q := Q ∪ { (y', x) ∈ G : Ps_(y',x) < T }
  return True;
end;
```

Fig. 2. PAC-3

Note that the use of PSC and PRC in PAC-3 is presented in such a way that the idea is made as clear as possible. This should not be taken as the real implementation. Putting more effort into estimating the probability of the support existence for each arc-value pair does not generally pay off in terms of the CPU time. However, there are ways to overcome this. We will only discuss one of them in the following paragraphs.

Expanding Equation (1) and rearranging the terms, gives:

```
Function revise_p(x, y, var change_x)
begin
   change_x := False
   for each a ∈ D(x) do
      if Ps_(x,y,a) ≥ T then
         \* do nothing *\
      else
         if ∄b ∈ D(y) such that b supports a then
            D(x) := D(x) \ { a }
            change_x := True
end
```

Fig. 3. Algorithm $revise_p$

$$Ps_{(x,y,a)} = 1 - \prod_{i=0}^{sc(x,y,a)-1} \left(1 - \frac{\mid D(y) \mid}{\mid D_{ac}(y) \mid - i}\right).$$

The underestimate ${Ps'}_{(x,y,a)}$ of the actual probability $Ps_{(x,y,a)}$ is as follows:

$${Ps'}_{(x,y,a)} = 1 - \left(1 - \frac{\mid D(y) \mid}{\mid D_{ac}(y) \mid}\right)^{sc(x,y,a)}. \tag{3}$$

Note that $Ps_{(x,y,a)} \geq {Ps'}_{(x,y,a)} \geq T$. Substituting the right hand side of Equation (3) into the condition ${Ps'}_{(x,y,a)} \geq T$ and rearranging terms gives us the following condition:

$$\mid D_{ac}(y) \mid \times (1 - T)^{1/sc(x,y,a)} \geq \mid R(y) \mid. \tag{4}$$

The above condition implies the PSC. It states that if the left hand side of Equation (4) is at least equal to $|R(y)|$ i.e. number of values removed from $D_{ac}(y)$, then ${Ps'}_{(x,y,a)}$ is at least equal to the threshold value T and since $Ps_{(x,y,a)} \geq {Ps'}_{(x,y,a)}$, PSC will also hold true. The advantage of Equation (4) is that the left hand side is *constant* and can be computed prior to search for each arc-value pair. Instead of recomputing the probability of the support existence in every iteration and comparing it with the threshold T during search, this constant can be compared with $|R(y)|$ to check if PSC holds. In a similar way, the overhead of of PRC can be brought down.

Integrating PSC and PRC in arc consistency, results in Probabilistic Arc Consistency [1] (PAC) and maintaining it during search, results in Maintaining Probabilistic Arc Consistency (MPAC). Similarly, using probabilistic arc consistency as the underlying algorithm of SAC results in probabilistic singleton arc consistency and maintaining it during search results in maintaining probabilistic singleton arc consitency.

5 Experimental Results

5.1 Introduction

In this section, we present some empirical results demonstrating the practical use of Probabilistic Singleton Arc Consistency (PSAC), and a limited version of PSAC (PLSAC).

[1] Probabilistic Arc Consistency discussed in this paper has no relation with the one mentioned in [8]

We examine the usefulness of PSAC and PLSAC, when used as a preprocessor and when maintained during search. We experimented with variety of problems, which were used as benchmarks for the First International CSP Solver Competition and are described in [4] and may be downloaded from `http://cpai.ucc.ie/05`.

Inferring the existence of a support using PSC and PRC in such a way that the amount of ineffective constraint propagation is minimised and simultaneously the least amount of effective propagation is avoided depends heavily on the threshold value T. In our previous investigations [11] on random problems, we found that inferring the existence of a support with a likelihood, roughly between 0.8 and 0.9, resolves this trade-off for maintaining probabilistic arc consistency (MPAC). Therefore, we decided to choose $T = 0.9$ for the experiments.

AC-3 is used to implement the arc consistency component of SAC. The reason why AC-3 is chosen is that it is easier to implement and is also efficient. For example, the best solvers in the binary and overall category of the First International CSP Solver Competition were based on AC-3. Similarly, PAC-3 is used to implement the probabilistic arc consistency component of the probabilistic version of SAC. SAC-1 is used to implement singleton arc consistency. All search algorithms were equipped with a $dom/wdeg$ [5] conflict-directed variable ordering heuristic. The performance of the algorithms is measured in terms of checks (#chks), time in seconds (time), the number of revisions (#revisions), and the number of visited nodes (#vn). The experiments were carried out on a PC Pentium III having 256 MB of RAM running at 2.266 GHz processor with linux. All algorithms were written in C.

5.2 Results

The algorithms SAC, LSAC, PSAC, and PLSAC were applied to a variety of known problems. Table 1 shows the results obtained on some representative instances of variety of known problems: (1) average of 100 satisfiable instances of balanced Quasigroup with Holes problems *bqwh15_106* and *bqwh18_141* (2) three attacking prime queen instances *qa-6*, *qa-7* and *qa-8*, (3) two queen-knights instances $K_{25} \oplus Q_8$ and $K_{25} \otimes Q_8$, (4) RLFAP instances *scene5* and *scene11*, (5) modified RLFAP instance *scene11_f6*, (6) GRAPH instances *graph10* and *graph14*, (7) two job-shop instances *enddr1-10-by-5-10* and *enddr2-10-by-5-2*, which are called *js-1* and *js-2* in [9] respectively, and (8) two sets of composed random instances *composed-25-10-20* and *composed-75-1-80*.

In Table 1 #rem denotes the number of removed values. The intention is not to test if the preprocessing by SAC has any effect in the global cost of solving the problem, but to see if the same results can be achieved by doing considerably less computation by using probabilistic support inference. When the input problem is already singleton arc consistent, PSAC and PLSAC avoid most of the unnecessary work. For example, for job-shop instances *enddr1-10-by-5-10* and *enddr2-10-by-5-2*, both PSAC and PLSAC spend at least 34 times less time than their counterparts. Even when the input problem is not singleton arc consistent, probabilistic versions of the algorithms are as efficient as the original versions. For most of the problems, they remove *exactly* the same number of values as removed by SAC and LSAC, but consume significantly less time. For example, in case of attacking queen problems, all the algorithms remove the same number of values. However, PSAC and PLSAC are quicker by an order of at least 27. In some cases,

e.g. for RLFAP 5, probabilistic support inference may not remove all possible values. However, proper tuning of threshold may eradicate this problem up to a certain extent.

The results shown for *composed-75-1-80* are average of 10 instances. All the instances can be shown insoluble by enforcing singleton arc consistency as a preprocessor. Though the algorithms PSAC and PLSAC remove fewer values than their their original versions SAC and LSAC, they are able to identify the insolubility of the problem. This is caused by the different revision orders enforced by probablistic versions.

Note that coarse-grained arc consistency algorithms require $\mathcal{O}(ed)$ revisions in the worst-case to make the problem arc consistent. Nevertheless, the maximum number of *effective revisions* (that is when at least a single value is deleted from a domain) cannot exceed $\mathcal{O}(nd)$, irrespective of whether the algorithm used is optimal or non-optimal. Thus, in the worst case, a coarse-grained arc consistency algorithm can perform $\mathcal{O}(ed-nd)$ ineffective revisions. The number of ineffective revisions in the worst-case increases to $\mathcal{O}(n^2d^2(ed-nd))$ for a SAC algorithm such as SAC-1, when its underlying arc consistency algorithm is a coarse-grained algorithm, since in the worst-case SAC-1 can call its underlying arc consistency algorithm n^2d^2 times. One can observe in Table 1 that probabilistic versions of SAC algorithm perform revisions far fewer than their original versions. This clearly shows that PRC is good in saving many ineffective revisions.

For easy problems, due to the expense entailed by computing the number of supports for each arc-value pair, probabilistic support inference may not be beneficial. However, the time required to initialise the support counters is not much. Furthermore, for all the harder instances, that require at least 1 second to solve, it generally pays off, by avoiding much ineffective propagation. In summary, inferring the existence of a support with a high probability slightly affects the pruning capability of SAC, but allows to save much ineffective propagation and saves a lot time.

Seeing the good performance of PSAC and PLSAC, the immediate question arises: can we afford to maintain them during search? So far SAC has been used only as a preprocessor. Maintaining SAC can reduce the number of branches significantly but at the cost of much constraint propagation, which may consume a lot of time. Maintaining it even for depth 1 within search has been found very expensive in [13]. We investigate if PSAC and PLSAC can be maintained within search economically. Table 2 shows the comparison of MAC, MPAC, MSAC (maintaining SAC), MLSAC (maintaining LSAC), MPSAC (maintaining PSAC), and MPLSAC (maintaining PLSAC) on structured problems. Mean results are shown only for quasigroup with holes (QWH) and quasi-completion problems (QCP) categorised as *easy* and *hard*. Note that here *easy* does not mean easy in the real sense. The results are first of their kind and highlight the following points: (1) the probabilistic version of the algorithm is better than its corresponding original version, (2) maintaining full or probabilistic (L)SAC reduces the branches of the search tree drastically, (3) though MLSAC and MPLSAC visit a few nodes more than MSAC and MPSAC, their run-times are low, (4) MPLSAC is the best in terms of checks and solution time when compared to other algorithms.

In our experiments, MPSAC/MPLSAC outperformed MSAC/ MLSAC for almost all the problems which we have considered. But, when compared to MAC and MPAC, they were found to be expensive for most of the problems except for quasi-group with holes and

Table 1. Comparison between SAC, LSAC, PSAC and PLSAC

problem		SAC	LSAC	PSAC	PLSAC
bqwh15_106	#chks	274,675	186,172	32,394	25,671
	#time	0.026	0.020	0.006	0.005
	#rev	71,370	52,295	14,384	11,436
	#rem	23	23	23	23
bqwh18_114	#chks	409,344	299,821	44,534	36,973
	#time	0.040	0.030	0.010	0.008
	#rev	108,326	78,969	19,136	14,317
	#rem	15	15	15	15
qa-6	#chks	163,477,129	166,395,478	4,930,448	4,930,448
	#time	8.095	8.300	0.234	0.232
	#rev	4,738,129	4,818,072	109,303	109,303
	#rem	48	48	48	48
qa-7	#chks	872,208,323	895,600,777	17,484,337	17,484,337
	#time	50.885	52.390	1.929	1.979
	#rev	17,198,583	17,627,515	262,493	262,493
	#rem	65	65	65	65
qa-8	#chks	3,499,340,592	3,533,080,066	51,821,816	51,821,816
	#time	249.696	290.534	9.790	11.496
	#rev	52,702,632	53,187,555	605,913	605,913
	#rem	160	160	160	160
$K_{25} \oplus Q_8$	#chks	206,371,887	206,371,887	16,738,737	16,738,737
	#time	2.407	2.446	0.469	0.450
	#rev	380,548	380,548	23,299	23,299
	#rem	3,111	3,111	3,111	3,111
$K_{25} \otimes Q_8$	#chks	1,301,195,918	1,301,195,918	19,252,527	19,252,527
	#time	13.473	13.469	0.613	0.635
	#rev	724,304	724,304	57,874	57,874
	#rem	3,112	3,112	3,112	3,112
scen5	#chks	9,896,112	11,791,192	2,793,188	2,348,189
	#time	0.700	0.858	0.126	0.120
	#rev	652,816	736,469	269,828	172,054
	#rem	13,814	13,814	13,794	13,794
scen11	#chks	622,229,041	622,229,041	16,376,619	16,376,619
	#time	21.809	21.809	3.005	3.005
	#rev	8,687,412	8,687,412	908,498	908,498
	#rem	0	0	0	0
scen11_f6	#chks	292,600,735	292,600,735	16,415,998	16,415,998
	#time	11.775	11.763	0.811	0.813
	#rev	6,399,661	6,399,661	471,902	471,902
	#rem	3660	3660	3660	3660
graph10	#chks	11,399,924,349	12,148,264,696	867,034,524	577,475,869
	#time	464.528	472.398	20.401	14.289
	#rev	106,347,157	112,774,481	7,585,779	5,312,567
	#rem	2,572	2,572	1,904	1,904
graph14	#chks	574,618,356	574,618,356	239,524,782	239,524,782
	#time	12.502	12.840	5.380	5.390
	#rev	6,964,869	6,964,869	1,028,732	1,028,732
	#rem	0	0	0	0
enddr1-10-by-5-10	#chks	600,508,958	600,508,958	14,100,504	14,100,504
	#time	15.549	15.411	0.446	0.415
	#rev	1,109,813	1,109,813	131,366	131,366
	#rem	0	0	0	0
enddr2-10-by-5-2	#chks	985,446,461	985,446,461	18,291,441	18,291,441
	#time	24.963	25.393	0.601	0.631
	#rev	14299380	14,299,380	160,337	160,337
	#rem	0	0	0	0
composed-25-10-20	#chks	5,272,064	6,184,748	744,324	611,475
	#time	0.258	0.303	0.076	0.061
	#rev	649,468	745,463	213,717	168,105
	#rem	392	391	392	392
composed-75-1-80	#chks	2,143,350	23,359	184,507	184,507
	#time	0.087	0.001	0.012	0.013
	#rev	200,328	2,121	25,610	25,610
	#rem	59	59	56	56

Table 2. Comparison between MAC, MSAC, MLSAC with their probabilistic versions (T = 0.9) on structured problems. (Checks are in terms of 1000s.)

problem		MAC	MPAC	MSAC	MPSAC	MLSAC	MPLSAC
	#chks	5,433	1,220	45,165	3,835	31,580	1,418
qwh-20	#time	2.17	1.34	10.83	3.08	8.69	1.32
(easy)	#vn	21,049	21,049	570	570	1,101	1,101
	#chks	186,872	42,769	756,948	70,666	293,152	34,330
qwh-20	#time	75.35	51.01	172.23	54.56	76.86	31.39
(hard)	#vn	693,662	693,662	3,598	3,598	7,136	7,136
	#chks	502,148	93,136	807,478	57,235	656,270	40,650
qwh-25	#time	252.89	154.00	236.20	63.71	221.46	53.506
(easy)	#vn	1,348,291	1,348,291	2,525	2,525	10,283	10,283
	#chks	600,697	670,370	2,090,689	768,556	954,767	211,826
qcp-20	#time	2,753.77	1,990.31	8,242.28	1,679.01	5,688.19	877.540
(easy)	#vn	26,191,095	26,191,095	107,624	107,624	586,342	586,342

quasi-group completion problems. However, this observation is made only for threshold value 0.9. Thorough testing remains to be done with different values of T. Possibility also exists to tune the value of threshold at different levels in the search.

6 Conclusions and Future Work

This paper considers the use of probabilistic approach to reduce ineffective constraint propagation in the singleton arc consistency algorithm SAC-1. The central idea is to avoid the process of seeking a support when there is a high probability of support existence. Inferring the existence of a support with a high probability allows an algorithm to save a lot of checks and time and slightly affects its ability to prune values. Enforcing probabilistic SAC almost always enforces SAC, but it requires significantly less time than SAC. Overall, experiments highlight the good performance of probabilistic support condition and probabilistic revision condition.

We believe that the idea of probabilistic support inference deserves further investigation. The notions of PSC and PRC can further be enhanced by taking into account the semantics of the constraints. Also, in future we would like to determine the extent to which maintaining probabilistic SAC could be a possible alternative to MAC. Recently, Lecoutre et al. [9] introduced a greedy approach to establish singleton arc consistency and based on that proposed two algorithms, particularly SAC-3 and SAC-3+. One certain perspective would be to study their probabilistic versions.

References

1. R. Barták and R. Erben. A new algorithm for singleton arc consistency. In *FLAIRS Conference*, 2004.
2. C. Bessière and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 54–59, 2005.
3. C. Bessière, J.-C. Régin, R. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

4. F. Boussemart, F. Hemery, and C. Lecoutre. Description and representation of the problems selected for the 1st international constraint satisfaction solver competition. In M. van Dongen, editor, *Proceedings of the 2nd International Workshop on Constraint Propagation and Implementation, Volume 2 Solver Competition*, pages 7–26, 2005.
5. F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs. Boosting systematic search by weighting constraints. In *Proceedings of the 13th European Conference on Artificial Intelligence*, 2004.
6. R. Debruyne and C. Bessière. Some practical filtering techniques for the constraint satisfaction problem. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 412–417, 1997.
7. R. Haralick and G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.
8. M. Horsch and W. Havens. Probabilistic arc consistency: A connection between constraint reasoning and probabilistic reasoning. In *16th Conference on Uncertainity in Artificial Intelligence*, 2000.
9. C. Lecoutre and S. Cardon. A greedy approach to establish singleton arc consistency. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 199–204, 2005.
10. A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
11. D. Mehta and M. van Dongen. Maintaining probabilistic arc consistency. In M. van Dongen, editor, *Proceedings of the 2nd International Workshop on Constraint Propagation and Implementation*, pages 49–64, 2005.
12. D. Mehta and M. van Dongen. Reducing checks and revisions in coarse-grained mac algorithms. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, 2005.
13. P. Prosser, K. Stergiou, and T. Walsh. Singleton consistencies. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, pages 353–368, 2000.
14. D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94)*, pages 125–129. John Wiley and Sons, 1994.

Simplification and Extension of the SPREAD Constraint

Pierre Schaus[1], Yves Deville[1], Pierre Dupont[1], and Jean-Charles Régin[2]

[1] Université catholique de Louvain, Belgium,
{pschaus,yde,pdupont}@info.ucl.ac.be
[2] ILOG, France, regin@ilog.fr

Abstract. Many assignment problems require the solution to be balanced. Such a problem is the Balanced Academic Curriculum Problem (BACP) [1]. Standard deviation is a common way to measure the balance of a set of values. A recent constraint presented by Pesant and Régin [2] enforces the mean μ and the standard deviation σ of a set of variables. Our work extends [2] by showing a more simple propagator from σ and μ to X and by introducing new propagators: from σ together with X to μ and from X together with μ to σ.

1 Introduction

In assignation problems, it is often desirable to have a fair or balanced solution. One example of such a problem is BACP. The goal is to assign periods to courses such that the academic load of each period is balanced, i.e., as similar as possible [1]. A perfectly balanced solution is generally not possible. A standard approach is to include the balance property in the objective function. Alternatively the constraint $SPREAD$ introduced by Pesant and Régin [2] could be used to reduce the search tree while simplifying the model. Constraining the variance of assignments to fall below an upper bound is a proper way to enforce the balance property.

Given a set of variables X and two variables μ and σ, $SPREAD(X, \mu, \sigma)$ states that the collection of values taken by the variables of X exhibits an arithmetic mean μ and a standard deviation σ. While the $SPREAD$ constraint in [2] also involves the median, this will not be considered here.

The $SPREAD$ constraint can be seen as a special kind of soft constraint opening new perspectives in CSP modeling. As a perfect balanced solution is mostly not possible, the perfect balance constraint can be soften with $SPREAD$ allowing a positive maximum standard deviation and an interval for the mean. $SPREAD$ could also be used to combine a set of soft constraints. The usual way to combine a set of soft constraints is to minimize the sum of the violation cost of each of them. The drawback with this approach is that some constraints could be much more violated that the others. A clever way could be to use $SPREAD$ to enforce the violation costs to be balanced among the soft constraints.

Section 2 mainly reviews the material from [2] used in this paper and introduces some statistical background and definitions relative to constraint programming. The problem of the variance minimization over the set of variables X is the starting point of our propagation algorithms. This problem is solved in [2] and explained in Section 2.

The propagator described in [2] filters from standard deviation σ to the set of variables X with quadratic time complexity with respect to the number of variables. It is possible to achieve better pruning by taking also the mean μ into consideration. Although the propagator from [2] can be easily extended to take also μ into account together with σ, this is not explicitly described in [2]. Section 3 presents a simpler filtering algorithm from μ and σ to X with the same time complexity.

We show in Section 4 that the problem of the variance minimization is a convex one. This implies that it admits a global minimum. This result allows us to design a propagator not present in [2]: from X and σ to μ. The filtering algorithm presented in Section 5 also performs in quadratic time with respect to the number of variables.

The filtering of the upper bound of the standard deviation requires a solution to the problem of the variance maximization. Section 6 shows that this problem is NP-hard and presents an algorithm to find an upper bound on the variance running in quadratic time with respect to the number of variables.

2 Background

We start this section with some statistical background and definitions relative to constraint programming. Next we present the problem of the variance minimization over the set of variables X. This problem is solved in [2] and is the starting point of our propagation algorithms.

We assume the reader familiar with common statistical notions such as *mean*, *standard deviation* and *variance* (these notions are defined in Section 2 of [2]). Note simply that a convenient way to compute the variance of a set of values $\{v_1, v_2, ..., v_n\}$ is the following: $\sigma^2 = \left(\frac{1}{n}\sum_{i=1}^{n} v_i^2\right) - \mu^2$.

We use the following notations for the variables and domains considered in this paper:

- A finite-domain (discrete) variable x takes a value in $D(x)$, a finite set called its domain. We denote the smallest (resp. largest) value x may take as $x^{\min}$ (resp. $x^{\max}$).
- A bounded-domain (continuous) variable y takes a value in $I_D(y) = [y^{\min}, y^{\max}]$, an interval on $\mathbb{R}$ called its domain as well.
- Given a finite-domain variable x, $I_D(x)$ denotes its domain relaxed to the continuous interval $[x^{\min}, x^{\max}]$. By extension for a union of domains $\mathcal{D} = \bigcup_{i=1}^{n} D(x_i)$, $I_{\mathcal{D}}$ represents the interval $[\min_{i=1}^{n} x_i^{\min}, \max_{i=1}^{n} x_i^{\max}]$.

The remaining of the section reviews the problem of the variance minimization solved in [2]. We detail successively the key points to find a solution:

1. A property of an optimal solution.
2. An optimal solution can be found by iterating once over a set of contiguous intervals.
3. The construction of this set of contiguous interval is based on the bounds of the domains.
4. For each interval, the optimal solution property can be checked in constant time.

Let first define formally the problem we want to solve. The variance minimization is an optimization problem under a sum constraint:

Definition 1 (Minimization of the variance on X). *Let $X = \{x_1, x_2, ..., x_n\}$ be a set of finite-domain (discrete) variables. For some fixed number q we denote by $\Pi_1(X, q)$ the problem: $\min \sum_{i=1}^{n}(x_i - q/n)^2$ such that $\sum_{i=1}^{n} x_i = q$, $x_i \in I_D(x_i),\ 1 \leq i \leq n$ and we denote by $opt(\Pi_1(X, q))$, or simply $opt(\Pi_1)$, the optimal value to this problem.*

In the above definition, $opt(\Pi_1)$ corresponds to n times the minimal variance and q to n times μ.

The following definition and lemma characterize an optimal solution to $\Pi_1(X, q)$. This property is a particular assignment of a variable x to a value of its relaxed domain to the continuous interval $I_D(x)$.

Definition 2. *An assignment $A : x \rightarrow I_D(x)$ is said to be a v-centered assignment when:*

$$A(x) = \begin{cases} x^{\max} & if\ x^{\max} \leq v \\ x^{\min} & if\ x^{\min} \geq v \\ v & otherwise \end{cases}$$

Lemma 1. *[2]. Any optimal solution to $\Pi_1(X, q)$ is a v-centered assignment.*

Lemma 1 gives a necessary condition for an assignment to be optimal for $\Pi_1(X, q)$ but the v value can be anywhere in $I_{\mathcal{D}}$. [2] introduces a splitting of $I_{\mathcal{D}}$ into contiguous intervals based on the bounds of the domains of variables. The v value of the v-centered assignment characterizing an optimal solution can be found by iterating once over this set of contiguous intervals. Any such interval is either subsumed by a domain or has an empty intersection with it but partial overlap cannot occur.

Definition 3. *Let $B(X)$ be the sorted sequence of bounds of the relaxed domains of the variables of X, in non-decreasing order and with duplicates removed. Define $\mathcal{I}(X)$ as the set of intervals defined by a pair of two consecutive elements of $B(X)$. The k^{th} interval of $\mathcal{I}(X)$ is denoted by I_k. For an interval $I = I_k$ we define the operator $prev(I) = I_{k-1}, (k > 1)$ and $succ(I) = I_{k+1}$.*

Example 1 (Building $\mathcal{I}(X)$). Let $X = \{x_1, x_2, x_3\}$ with $I_D(x_1) = [1, 3]$, $I_D(x_2) = [2, 6]$ and $I_D(x_3) = [3, 9]$ then $\mathcal{I}(X) = \{I_1, I_2, I_3, I_4\}$ with $I_1 = [1, 2]$, $I_2 = [2, 3]$, $I_3 = [3, 6]$, $I_4 = [6, 9]$. We have $prev(I_3) = I_2$ and $succ(I_3) = I_4$.

There are at most $2.n - 1$ intervals in $\mathcal{I}(X)$. Let assume that the value v of the optimal solution to $\Pi_1(X, q)$ lies in the interval $I \in \mathcal{I}(X)$. We denote by $R(I) = \{x | x^{\min} \geq \max(I)\}$ the variables lying to the right of I and by $L(I) = \{x | x^{\max} \leq \min(I)\}$ the variables lying to the left of I. By Lemma 1, all variables $x \in L(I)$ take their value $x^{\max}$ and all variables in $R(I)$ take their value $x^{\min}$. It remains to assign the variables subsuming I. We denote these variables by $M(I) = \{x | I \subseteq I_D(x)\}$ and the cardinality of this set by $m = |M(I)|$. By Lemma 1, the variables of $M(I)$ must take a common value v. The sum constraint (see Definition 1) of $\Pi_1(X, q)$ can be rewritten as

$$\sum_{x \in R(I)} x^{\min} + \sum_{x \in L(I)} x^{\max} + \sum_{x \in M(I)} v = q. \tag{1}$$

Let denote the sum of extrema by

$$ES(I) = \sum_{x \in R(I)} x^{\min} + \sum_{x \in L(I)} x^{\max}.$$

The sum constraint in Equation (1) implies that v must be equal to a specific value $v^* = (q - ES(I))/m$. This results in a valid assignment only if $v^* \in I$. This condition is satisfied if

$$q \in V(I) = [ES(I) + \min(I).m, ES(I) + \max(I).m].$$

We previously said that an optimal solution the problem $\Pi_1(X, q)$ (see Definition 1) could be found by iterating once over a set of contiguous intervals by checking for each interval a property in constant time. The set of intervals is naturally $\mathcal{I}(X)$ introduced in Definition 3 and for each $I \in \mathcal{I}(X)$, the test is: does q belong to $V(I)$? If it is true that $q \in V(I)$, the value v of the v-centered assignment defined in Definition 2 characterizing an optimal solution $\Pi_1(X, q)$ lies in the interval I and has a value of $(q - ES(I))/m$.

We denote the overall minimal (resp. maximal) sum by $\underline{S}(X) = \sum_{x \in X} x^{\min}$ (resp. $\overline{S}(X) = \sum_{x \in X} x^{\max}$). We are sure that for every value $q \in [\underline{S}(X), \overline{S}(X)]$ there is one $I \in \mathcal{I}(X)$ such that $q \in V(I)$. Indeed, we have $\min(V(I_1)) = \underline{S}(X)$, $\max(V(I_{|\mathcal{I}(X)|})) = \overline{S}(X)$ and for two consecutive intervals I_k, I_{k+1} from $\mathcal{I}(X)$, we have $\min(V(I_{k+1})) = \max(V(I_k))$, thus leaving no gap.

Theorem 1. *[2] Given a value q such that $q \in [\underline{S}(X), \overline{S}(X)]$ and $I^q \in \mathcal{I}(X)$ such that $q \in V(I^q)$, the following assignment gives the optimal value to $\Pi_1(X, q)$:*

$$A(x) = \begin{cases} x^{\max} & if\ x \in L(I^q) \\ x^{\min} & if\ x \in R(I^q) \\ v = \frac{q - ES(I^q)}{m} & if\ x \in M(I^q) \end{cases}$$

Example 2 (Solving $\Pi_1(X, q)$). Variables and domains are from Example 1. We obtain the following values:

i	I_i	$R(I_i)$	$L(I_i)$	$M(I_i)$	$ES(I_i)$	$V(I_i)$
1	$[1,2]$	x_2, x_3	ϕ	x_1	5	$[6,7]$
2	$[2,3]$	x_3	ϕ	x_1, x_2	3	$[7,9]$
3	$[3,6]$	ϕ	x_1	x_2, x_3	3	$[9,15]$
4	$[6,9]$	ϕ	x_1, x_2	x_3	9	$[15,18]$

For $q = 10$ we have $q \in V(I_3)$ thus $I^{10} = I_3$. $A(x_1) = 3$, $A(x_2) = A(x_3) = 3.5$. For $q = 9$, we have $q \in V(I_2)$ and $q \in V(I_3)$. Whichever interval we choose between I_2 and I_3, we find the same optimal assignment $A(x_1) = 3$, $A(x_2) = 3$ and $A(x_3) = 3$.

3 Propagation from μ and σ to X

In this section, we propose to reformulate, simplify and extend (by considering explicitly the mean together with the standard deviation) the propagator given in [2]. The procedure to filter the domain of a variable $x \in X$ is the following:

- Shift the domain of x by a positive real quantity d.
- For some maximal shift $d = d^{\max}$, the minimum standard deviation reaches the upper bound $\sigma^{\max}$ of the domain of σ.
- The computed value $d^{\max}$ allows us to filter $D(x)$ since a shift larger that $d^{\max}$ would render the constraint inconsistent.

To clarify the presentation, we first assume that σ is an interval $[\sigma^{\min}, \sigma^{\max}]$ and μ is a given value. We consider afterward the general case where μ is an interval.

We recall and introduce some notations to explain more precisely the filtering procedure. We denote $q = n\mu$, $\pi_1^{\max} = n(\sigma^{\max})^2$ and $I^q \in \mathcal{I}(X)$ is such that $q \in V(I^q)$. In the following we use the shift operation $I + d$ by a positive quantity d on an interval $[I^{\min}, I^{\max}]$ to denote interval $[I^{\min} + d, I^{\max} + d]$. This operation also applies on domains of variables: we simply denote by $x' = x + d$ the variable x with a shifted domain $D(x) + d$.

These notations allow us to explain more precisely the filtering of the domain of one variable $x \in X$. First, the constraint fails if the minimum standard deviation is larger than the upper bound of σ. In this case, there exists no consistent assignment. This happens if $opt(\Pi_1) > \pi_1^{\max}$. When the constraint is consistent ($opt(\Pi_1) \leq \pi_1^{\max}$) we can consider the filtering of each variable $x \in X$. In particular for a variable $x \in R(I^q)$ (resp. $\in L(I^q)$), we compute its maximal consistent value (resp. minimal consistent value) by computing the maximal shift $d^{\max}$. For a variable $x \in M(I^q)$ we compute both. Each value larger (resp. smaller) than the maximal (resp. minimal) consistent value must be filtered. As the problem is symmetrical we only consider the computation of the maximal consistent value for $x \in R(I^q) \cup M(I^q)$.

For a variable $x \in R(I^q) \cup M(I^q)$ we show that shifting its domain $(D(x) + d)$ by $d \in \Re^+$ increases $opt(\Pi_1)$ ($n\times$ the minimum variance) quadratically with d. The bound $\pi_1^{\max}$ is reached for some d denoted by $d^{\max}$. The propagation on X

considers each variable $x \in X$ in turn, computes its maximum shift $d^{\max}$ and prunes $D(x)$ as follows: $D(x) \leftarrow D(x) \cap [x^{\min}, x^{\min} + d^{\max}]$. All the domains can be updated once after consideration of all variables in X. Alternatively, each pruned domain can directly be used for the propagation on the other variables.

Searching $d^{\max}$ for $x \in R(I^q)$ X' denotes X after the shift $x' = x + d$. Let $\Pi_1(X', q)$, $ES'(I^q)$ and $V'(I^q)$ be the corresponding quantities for X'. We have $ES'(I^q) = ES(I^q) + d$ and $V'(I^q) = V(I^q) + d$.

Let assume that $d \leq d_1 = q - \min(V(I^q))$ such that v' remains in I^q. In this case, the value of q leading to the value of $opt(\Pi_1)$ does not change (i.e. $q' = q$). Only the v value will change in the optimal assignment: $v' = v - d/m$. We have $opt(\Pi_1(X', q)) = \left(\sum_{x_i \in L(I^q)} (x_i^{\max})^2\right) + \left(\sum_{x_i \in R(I^q)} (x_i^{\min})^2\right) + d^2 + 2dx^{\min} + \left(\sum_{x_i \in M(I^q)} (v - \frac{d}{m})^2\right) - \frac{q^2}{n} = opt(\Pi_1(X, q)) + d^2 + 2dx^{\min} + m\left(\frac{d^2}{m^2} - 2\frac{d}{m}v\right)$. The value $d^{\max}$ is the positive solution of a second degree equation $ad^2 + 2bd + c$, where $a = (1 + \frac{1}{m})$, $b = x^{\min} - v$ and $c = opt(\Pi_1(X, q)) - \pi_1^{\max}$.

Until now, we made the assumption that $d \leq d_1$. If $d^{\max} > d_1$ this value is not valid since v does not lie within I^q anymore. In this case x is shifted by d_1 (i.e. $x^{\min}$ is increased by d) and the interval $I'^q = prev(I^q)$ is considered. The resulting Algorithm 1 searching for $d^{\max}$ runs in $O(n)$ since there are at most $|\mathcal{I}(\mathcal{X})| < n$ recursive calls and that the body runs in $O(1)$.

Algorithm: FindDMax(x, I^q)
Data: $x \in R(I^q)$; $I^q \in \mathcal{I}$; $q \in V(I^q)$;
Result: $d^{\max}$ s.t. $opt(\Pi_1(X', q)) = \pi_1^{\max}$ with $x' = x + d^{\max}$
$d_1 = q - \min(V(I^q))$
$d^{\max} = \frac{-b + \sqrt{b^2 - ac}}{a}$
if $d^{\max} < d_1$ **then**
  return $d^{\max}$
else
  if $I^q = I_1$ **then**
    return d_1
  else
    return d_1+FindDMax($x + d_1, prev(I^q)$)
  end
end

Algorithm 1: FindDMax

Searching $d^{\max}$ with $x \in M(I^q)$ can be reduced to searching for $d^{\max}$ with a new variable x' with $x'^{\min} = v$. When x is increased ($x' = x + d$), the optimal assignment does not change if $d \leq v - x^{\min}$ (i.e. the values of $A(x)$ remain the

same) . For $d = v - x^{\min}$ two new intervals are created replacing the old I^q: $I_j = [\min(I^q), v]$ and $I_k = [v, \max(I^q)]$ with $q = \max(V'(I_j)) = \min(V'(I_k))$. The optimal assignment is the same but a new problem $\Pi_1(X', q)$ is created with $q \in V'(I_j)$ and $x' \in R(I_j)$. This case reduced to searching for $d^{\max}$ with $x' \in R(I_j)$ is exposed above. The final $d^{\max}$ relative to the variable x is given by:

$$d^{\max} = v - x^{\min} + FindDMax(x', I_j) \;\; where \;\; x' = x + v - x^{\min} \qquad (2)$$

Example 3 (Filtering of one domain). Variables and domains are from Example 1. This example shows the filtering of the domain of variable x_2 for $q = 10$ and $\pi_1^{\max} = 8$. We are in the case of searching $d^{\max}$ with $x \in M(I^q)$. We have $I^{10} = [3, 6]$ because $10 \in V([3, 6]) = [9, 15]$ and the value v of the v-centered assignment is 3.5 (see Example 2). From Equation (2) we have $d^{\max} = 3.5 - 2 + FindDMax(x'_2, [3, 3.5])$ where $x'_2 = x_2 + 1.5$. We now analyze the successive calls to Algorithm 1.

1. $FindDMax(x_2 + 1.5, [3, 3.5])$. We have $ES([3, 3.5]) = 6.5$, $V([3, 3.5]) = [9.5, 10]$, $d_1 = 0.5$, $a = 2$, $b = 0$ and $c = (3 - 10/3)^2 + 2 * (3.5 - 10/3)^2 - 8 \approx -7.83$. We can compute $d^{\max} \approx 1.98$. Since $d^{\max} > d_1$ we have the recursive call $FindDMax(x_2 + 1.5 + 0.5, [1, 3])$.
2. $FindDMax(x_2 + 1.5 + 0.5, [1, 3])$. We have $ES([1, 3]) = 7$, $V([1, 3]) = [8, 10]$, $d_1 = 2$, $a = 2$, $b = 1$ and $c = 2 * (3 - 10/3)^2 + (4 - 10/3)^2 - 8 \approx -7.33$. We can compute $d^{\max} \approx 1.48$. Since $d^{\max} < d_1$ we can return 1.48.

In conclusion, the $d^{\max}$ value of x_2 is $1.5 + 0.5 + 1.48 = 3.48$ and the variable x_2 can be filtered $D(x_2) \leftarrow D(x_2) \cap [2, 5.48]$.

An example of the application of Algorithm 1 is given in Figure 1. The complexity analysis of Algorithm 1 shows that $d^{\max}$ is computed in $O(n)$ making the propagation on whole X running in $O(n^2)$.

Extension to $\mu = [\mu^{\min}, \mu^{\max}]$ The generalization $\mu = [\mu^{\min}, \mu^{\max}]$ is equivalent to $q \in [q^{\min} = n\mu^{\min}, q^{\max} = n\mu^{\max}]$. This extension does not affect our propagator but only requires an additional step before the call to $FindDMax$ for each variable: the computation of a suitable $q \in [q^{\min}, q^{\max}]$. The computation of $d^{\max}$ in the algorithm depends on the value of q. To express this explicitly we denote $d^{\max}$ as a function of q: $d^{\max}(q)$. Since it can be shown to be concave and derivable, one can search a q^0 such that $d^{\max}(q)$ is maximum: $\frac{\partial d^{\max}}{\partial q} \mid_{q=q^0} = 0$. It can be shown that q^0 is the only valid solution of a second degree equation . As $q \in [q^{\min}, q^{\max}]$, if $q^0 > q^{\max}$ (resp. $< q^{\min}$) then $FindDmax$ is called with $q = q^{\max}$ (resp. $q = q^{\min}$). If $q^0 \in [q^{\min}, q^{\max}]$, $FindDmax$ is called with $q = q^0$.

4 Study of $\Pi_1(X, q)$

We show in this section that the problem of the variance minimization with given mean is convex. This result allows us to design a propagator from X and

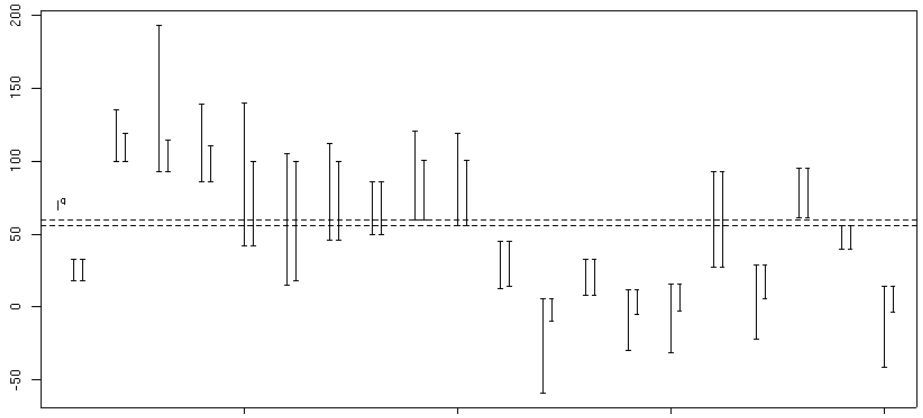


Fig. 1. The propagation on a typical run. The I^q interval lies between the two horizontal lines. The posted constraint is $SPREAD(X, 50, [0, 28])$. There are 20 variables and the domains after the propagations are represented on the right of each original domain.

σ to μ in Section 5. Indeed, the values for the mean leading to a minimum standard deviation larger than the upper bound $\sigma^{\max}$ must be filtered. Thanks to the convexity property, all inconsistent values for the mean will be filtered by computing only two values for the mean such that the upper bound $\sigma^{\max}$ is reached. All the values for the mean not between these two computed values are inconsistent.

More precisely, in this section we characterize completely the function of q $opt(\Pi_1(X, q))$ which is the minimization of the variance for a fixed mean (see Definition 1). We demonstrate that $opt(\Pi_1(X, q))$ is continuous, derivable, convex and accepts one global minimum on $[\underline{S}(X), \overline{S}(X)]$. Figure 2 shows a typical set X of variables with their domains and the corresponding functions $opt(\Pi_1(X, q))$. You can see on the figure that $opt(\Pi_1(X, q))$ is continuous, convex with one global minimum.

Theorem 2 (Characteristics of $opt(\Pi_1(X, q))$). *Assuming a domain for q in the interval $[\underline{S}(X), \overline{S}(X)]$ and a given set of variables X the optimal value to $\Pi_1(X, q)$ denoted by $opt(\Pi_1(X, q))$ is continuous, differentiable and convex having a global minimum for some $q \in [\underline{S}(X), \overline{S}(X)]$.*

Proof. It is sufficient to show that $\partial opt(\Pi_1(X, q))/\partial q$ is a continuous (1) increasing (2) function with $\partial opt(\Pi_1(X, q))/\partial q \mid_{q=\underline{S}(X)} \leq 0$ and $\partial opt(\Pi_1(X, q))/\partial q \mid_{q=\overline{S}(X)} \geq 0$ (3).

The function $opt(\Pi_1(X, q))$ is piecewise defined on $[\underline{S}(X), \overline{S}(X)]$: for $q \in V(I_k)$, $opt(\Pi_1(X, q)) = C + \sum_{x_i \in M(I_k)}((q - ES(I_k))/m)^2 - q^2/n$ where $C = \sum_{x_i \in L(I_k)}(x_i^{\max})^2 + \sum_{x_i \in R(I_k)}(x_i^{\min})^2$. The derivative is also piecewise defined:

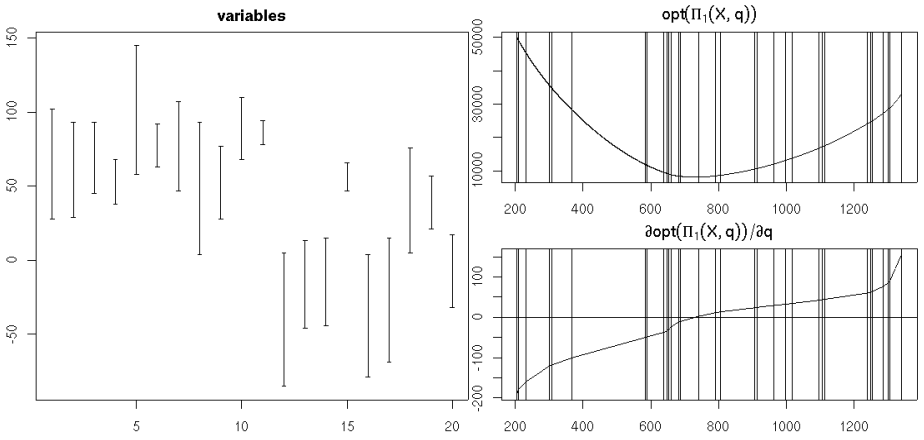


Fig. 2. On the left a typical set X is represented with the domain of each variables. On the right top and bottom $opt(\Pi_1(X,q))$ and $\partial opt(\Pi_1(X,q))/\partial q$ are respectively represented for $q \in [\underline{S}(X), \overline{S}(X)]$. The vertical lines represent $V(I_k), 1 \leq k \leq |\mathcal{I}(X)|$.

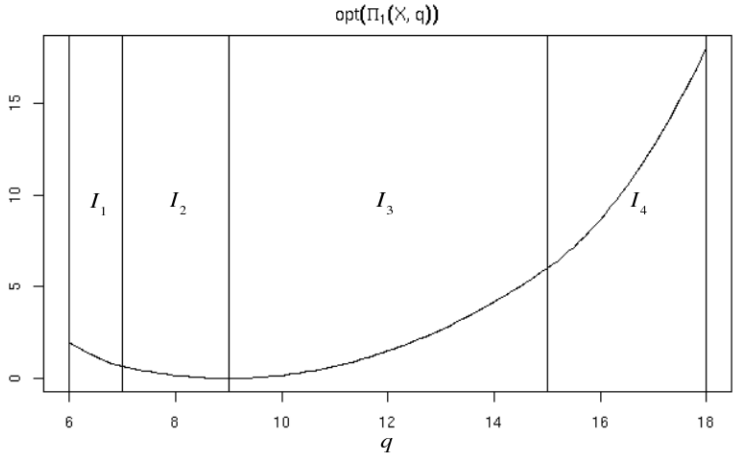
for $q \in V(I_k)$, $\partial opt(\Pi_1(X,q))/\partial q = 2(q - ES(I_k))/m - 2q/n$. The proofs for (1),(2) and (3) are:

1. The derivative is continuous because for $q = \max(V(I_k)) = \min(V(I_{k+1}))$, the values obtained on interval $V(I_k)$ and $V(I_{k+1})$ are the same: $2.\frac{q-ES(I_k)}{|M(I_k)|} - 2\frac{q}{n} = 2\frac{q-ES(I_{K+1})}{|M(I_{K+1})|} - 2\frac{q}{n}$. Indeed, by denoting $\delta^m = |M(I_{k+1})| - |M(I_k)|$ we have $\frac{q-ES(I_k)}{|M(I_k)|} = \frac{ES(I_k)+|M(I_k)|\max(I_k)-ES(I_k)}{|M(I_k)|} = \max(I_k)$ and $\frac{q-ES(I_{k+1})}{|M(I_{k+1})|} = \frac{ES(I_k)+|M(I_k)|\max(I_k)-ES(I_k)+\delta^m \max(I_k)}{|M(I_k)|+\delta^m} = \max(I_k)$.
2. Since $\partial^2 opt(\Pi_1(X,q))/\partial q^2 = 2(\frac{1}{|M(I_k)|} - \frac{1}{n}) \geq 0$, $\partial opt(\Pi_1(X,q))/\partial q$ is non decreasing on $V(I_k)$. Because $\partial opt(\Pi_1(X,q))/\partial q$ is continuous (1) and non decreasing on each interval the function is globally convex on $[\underline{S}(X), \overline{S}(X)]$.
3. Note that $ES(I_1) = \underline{S}(X) - m\min(I_1)$ and $ES(I_{|\mathcal{I}(X)|}) = \overline{S}(X) - m\max(I_{|\mathcal{I}(X)|})$. $\partial opt(\Pi_1(X,q))/\partial q \mid_{q=\underline{S}(X)} = \frac{\underline{S}(X)-\underline{S}(X)+m\min(I_1)}{m} - \frac{\underline{S}(X)}{n} = \min(I_1) - \frac{\underline{S}(X)}{n} \leq 0$. $\partial opt(\Pi_1(X,q))/\partial q \mid_{q=\overline{S}(X)} = \frac{\overline{S}(X)-\overline{S}(X)+m\max(I_{|\mathcal{I}(X)|})}{m} - \frac{\overline{S}(X)}{n} = \max(I_{|\mathcal{I}(X)|}) - \frac{\overline{S}(X)}{n} \geq 0$. □

Example 4 (Study of $opt(\Pi_1(X,q))$). Variables and domains are from Example 1. We study the function $opt(\Pi_1(X,q))$. With help of the table from Example 2 we add one column which is the definition of $opt(\Pi_1(X,q))$ for $q \in V(I_i)$.

i	I_i	$R(I_i)$	$L(I_i)$	$M(I_i)$	$ES(I_i)$	$V(I_i)$	C	$opt(\Pi_1(X,q))$ for $q \in V(I_i)$
1	$[1,2]$	x_2, x_3	ϕ	x_1	5	$[6,7]$	13	$13 + 1 * (\frac{q-5}{1})^2 - \frac{q^2}{3}$
2	$[2,3]$	x_3	ϕ	x_1, x_2	3	$[7,9]$	9	$9 + 2 * (\frac{q-3}{2})^2 - \frac{q^2}{3}$
3	$[3,6]$	ϕ	x_1	x_2, x_3	3	$[9,15]$	9	$9 + 2 * (\frac{q-3}{2})^2 - \frac{q^2}{3}$
4	$[6,9]$	ϕ	x_1, x_2	x_3	9	$[15,18]$	45	$45 + 1 * (\frac{q-9}{1})^2 - \frac{q^2}{3}$

Each function $opt(\Pi_1(X,q))$ for $q \in V(I_i)$ is plotted on the following graphics. Clearly the minimum is reached for $q = 9$ in this example.



5 Propagation from X and σ to μ

As already explained at the beginning of Section 4, the convexity property of the problem of variance minimization (see Theorem 2) with given mean allows us to design an efficient propagator from X and σ to μ. All the values for the mean leading to a minimum standard deviation larger than the upper bound $\sigma^{\max}$ can be filtered. Thanks to the convexity property, all inconsistent values for the mean will be filtered by computing only two values for the mean such that the upper bound $\sigma^{\max}$ is reached. All the values for the mean not between these two computed values are inconsistent.

We now explain more precisely the narrowing of μ with help of Figure 3. The function $\underline{\sigma}(\mu)$ depicted on figure 3 is the function $\sqrt{opt(\Pi_1(X,q))/n}$ with $\mu = q/n$. Naturally this function has the same properties than the function $opt(\Pi_1(X,q))$. The constraint $\sigma \leq \sigma^{\max}$ is represented by a vertical line crossing $\underline{\sigma}(\mu)$ in two points. The projection of these two points on the mean axis gives the two values μ_1, μ_2 for the mean such that the minimum standard deviation is equal to the upper bound of σ. All mean values outside the interval $[\mu_1, \mu_2]$ are inconsistent and can be filtered.

It is possible that the maximum standard deviation is so large that it does not constraint the mean. In this case $\mu_1 = \underline{S}/n$ and $\mu_2 = \overline{S}/n$ and we have simply a propagation from X to μ.

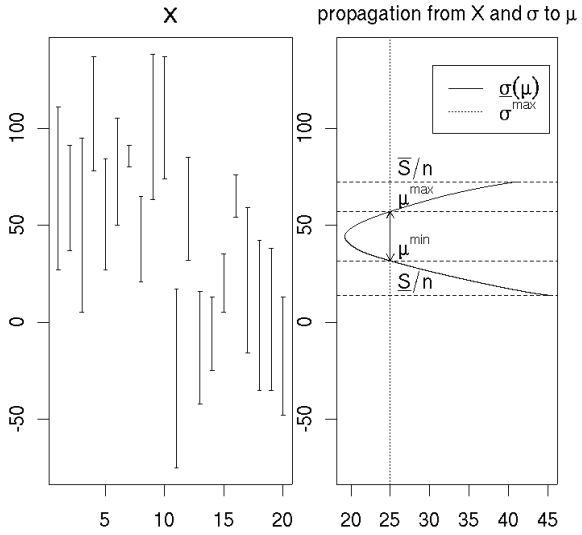


Fig. 3. Propagation from X and σ to μ.

In the remaining of this section we explain how the two values μ_1, μ_2 are found and finally we give the resulting filtering algorithm for μ

As already said μ_1, μ_2 are the projection on the mean axis of the two cross points of $\sqrt{opt(\Pi_1(X,q))/n}$ with $\sigma^{\max}$ (see Figure 3). These two cross points are obtained by considering each interval $V(I_k)$ in turn. It is possible to find two values $n.\mu_1 = q_1 \leq q_2 = n.\mu_2$ for q such that $opt(\Pi_1(X,q_1)) = opt(\Pi_1(X,q_2)) = \pi_1^{\max} = n(\sigma^{\max})^2$ and $\forall q \in [q_1, q_2], opt(\Pi_1(X,q)) \leq \pi_1^{\max}$. The two values q_1, q_2 are found as follows. For every value of q: $opt(\Pi_1(q)) = C + \sum_{x_i \in M(I_k)} \left(\frac{q - ES(I_k)}{m}\right)^2 - \frac{q^2}{n}$ where $C = \sum_{x_i \in L(I_k)} (x_i^{\max})^2 + \sum_{x_i \in R(I_k)} (x_i^{\min})^2$. Then, q_1 and q_2 are the solutions of the second degree equation $aq^2 + 2bq + c$ where $a = (1/m - 1/n)$, $b = -ES(I_k)/m$ and $c = C + (1/m).ES(I_k)^2 - \pi_1^{\max}$. If $q_1 = (-b - \sqrt{b^2 - ac})/a \in V(I_k)$ then $\mu_1 = q_1/n$ is a lower bound of the permitted interval for μ. If $q_2 = (-b + \sqrt{b^2 - ac})/a \in V(I_k)$ then $\mu_2 = q_2/n$ is the upper bound of the permitted interval for μ. Else there is no bounds in $V(I_k)$. The resulting Algorithm 2 narrows the interval μ if possible.

Example 5 (Filtering of μ).

Variables and domains are from Example 1. We search the permitted values for μ under the constraint $\pi_1^{\max} = 8$. Clearly, if we look at the figure of Example 4, we can deduce that $q^{\min} = 6$ but the upper bound $q^{\max}$ must be computed. All we know by looking at the figure is that $q^{\max} \in V(I_4)$ because the curve $opt(\Pi_1(X,q))$ intersects $\pi_1^{\max} = 8$ in this interval. We can take the expression of $opt(\Pi_1(X,q))$ on the interval $V(I_4)$ (see Example 4) and compute the value $q^{\max}$ such that $opt(\Pi_1(X,q^{\max})) = \pi_1^{\max} = 8$. We have the equation $45 + 1 *$

Algorithm: MeanPruning
Result: narrowing of μ
set $\mu^{min} \geq \underline{S}/n$
set $\mu^{max} \leq \overline{S}/n$
for $1 \leq k \leq |\mathcal{I}(X)|$ **do**
  $q_1 = (-b - \sqrt{b^2 - ac})/a$
  if $q_1 \in V(I_k)$ **then**
    set $\mu^{min} \geq q_1/n$
    break
  end
end
for $|\mathcal{I}(X)| \geq k \geq 1$ **do**
  $q_2 = (-b + \sqrt{b^2 - ac})/a$
  if $q_2 \in V(I_k)$ **then**
    set $\mu^{max} \leq q_2/n$
    break
  end
end

Algorithm 2: MeanPruning

$(\frac{q^{\max}-9}{1})^2 - \frac{(q^{\max})^2}{3} = 8$ and we find $q^{\max} \approx 15.79$. A bound consistent interval for the mean is thus $[6/3, 15.79/3] = [2, 3.74]$.

6 Narrowing of σ

The propagation from X and μ to $\sigma^{\min}$ is detailed in [2]. We propose to study the propagation from X and μ to $\sigma^{\max}$.

The decreasing of the upper bound of σ requires to compute the maximal variance on X with a given mean. This can be shown to be a convex maximization problem (NP-hard in general [3]). Even the relaxed problem without the sum constraint remains a convex maximization problem but it is easier to design an upper bound on it because of a known characterization of the optimal solution with respect to the extrema of the domains. We describe in this section a quadratic running time algorithm (with respect to the number of variables) to find an upper bound on the variance.

The maximization problem we want to solve is:

Definition 4 (Maximization of the variance on X). *Let $X = \{x_1, x_2, ..., x_n\}$ be a set of finite-domain (discrete) variables. We denote by $\Pi_2(X)$ the problem: $\max \sum_{i=1}^{n}(x_i - \sum_{j=1}^{n} x_j/n)^2$. We denote by $opt(\Pi_2(X))$ the optimal value for the problem.*

Since $opt(\Pi_2(X)) = \sum_i x_i^2 - \left(\sum_i x_i\right)^2/n$, an upper bound $\overline{opt}(\Pi_2(X))$ can be computed using the bound values $x_i^{\max}$ (resp. $x_i^{\min}$) in the first (resp. second) sum. This upper bound can be used to narrow the interval σ by setting $n.\sigma^2 \leq \overline{opt}(\Pi_2(X))$.

Example 6 (Upper bound). We consider the same variables and domains as in Example 1. We have $X = \{x_1, x_2, x_3\}$ with $I_D(x_1) = [1, 3]$, $I_D(x_2) = [2, 6]$ and $I_D(x_3) = [3, 9]$. $\overline{opt}(\Pi_2(X)) = (3^2 + 6^2 + 9^2) - (1 + 2 + 3)^2/3 = 114$.

The following lemma gives a property on an optimal assignment for the variance maximization problem. We will use this property to improve the upper bound in $O(n^2)$.

Lemma 2 (Optimal solution to $\Pi_2(X)$). *Any optimal solution to $\Pi_2(X)$ must be an assignment on the extrema of the domains i.e. on $x^{\max}$ or $x^{\min}$.*

Proof (Proof of Lemma 2). It is sufficient to show that starting from an arbitrary assignment and choosing an arbitrary variable $x_i > \sum_j x_j/n$, assigning a greater value to x_i i.e. $x_i \leftarrow x_i + d$ will increase the variance on X. The previous variance was $\sigma^2 = \frac{1}{n}\sum_j x_j^2 - \frac{1}{n^2}(\sum_j x_j)^2$ the variance with the modified variable is $\sigma'^2 = \frac{1}{n}\sum_j x_j^2 + \frac{1}{n}(d^2 + 2dx_i) - \frac{1}{n^2}(\sum_j x_j)^2 - \frac{1}{n2}(d^2 + 2\sum_j(d.x_j))$. The result is $\sigma'^2 = \sigma^2 + \frac{1}{n}(d^2 + 2dx_i) - \frac{1}{n^2}(d^2 + 2d\sum_j(x_j)) > \sigma^2 + \frac{1}{n}(d^2 + 2dx_i) - \frac{1}{n^2}(d^2 + 2dnx_i) = \sigma^2 + \frac{1}{n}d^2 - \frac{1}{n^2}d^2$ with $\sigma'^2 > \sigma^2$. The same result holds by symmetry for a variable $x_i < \sum_j x_j/n$ if it is decreased $x_i \leftarrow x_i - d$. □

As already explained, an upper bound for $\overline{opt}(\Pi_2(X))$ can be computed using the values and $x_i^{\max}$ (resp. $x_i^{\min}$) in the first (resp. second) sum of $\sum_i x_i^2 - \left(\sum_i x_i\right)^2/n$. With Lemma 2, it is possible to improve this bound. In each case where the lower-bound using an extrema is larger than the upper-bound using the other extrema, the optimal assignment corresponds to the first extrema. If for one variable, the extrema assignment can be found, then we can use this extrema value in the first and in the second sum to decrease the upper bound. If all the extrema assignment could be found the upper bound would be optimal (equal to the maximum variance). There are 2^n possible extrema assignments on X. We suggest an $O(n^2)$ algorithm to deduce as much extrema assignments as possible.

We now detail the method to deduce the correct extrema assignment of some variables. We denote $\underline{\mu} = \underline{S}(X)/n$ and $\overline{\mu} = \overline{S}(X)/n$. For some variables the optimal assignment can be deduced immediately. Indeed if $x^{\min} > \overline{\mu}$, an optimal solution to $\Pi_2(X)$ is such that $x = x^{\max}$. The case $x^{\max} < \underline{\mu}$ is symmetrical. There are additional cases where extrema assignment can be deduced. Note that if x would be assigned to $x^{\min}$, the upper bound for μ would become $\overline{\mu}^* = \overline{\mu} - \frac{x^{\max} - x^{\min}}{n}$.

In the example on the left of Figure 4, an optimal solution would assign $x = x^{\max}$ because the lower bound on the distance of $x^{\max}$ to μ is greater than the upper bound on the distance of $x^{\min}$ to $\overline{\mu}^*$. More generally, in each case where the lower-bound using an extrema is larger than the upper-bound using the other extrema, the optimal assignment corresponds to the first extrema.

Assigning a variable x to $x^{\min}$ will decrease $\overline{\mu}$ and assigning a variable x to $x^{\max}$ will increase $\overline{\mu}$ resulting possibly in a larger set of variables for which an optimal assignment can be deduced. All such extrema can be found in $O(n^2)$.

Example 7 (Deducing extrema assignment). We consider the same variables and domains as in Example 1. We have $X = \{x_1, x_2, x_3\}$ with $I_D(x_1) = [1, 3]$, $I_D(x_2) = [2, 6]$ and $I_D(x_3) = [3, 9]$. We have $\underline{\mu} = 2$ and $\overline{\mu} = 6$.

- x_3: If we assign x_3 to 9 then we have $\underline{\mu} = 4, \overline{\mu} = 6$ and the smallest distance from 9 to μ is $9 - 6 = 3$. If we assign x_3 to 3 then we have $\underline{\mu} = 2, \overline{\mu} = 4$ and the largest distance from 3 to μ is 1. We are sure that the correct extrema assignment for x_3 is 9 because whichever the assignment on other variables is, the distance to μ (and thus the variance also) will always be greater with x_3 assigned to 9. The new values for the bounds on the mean are now $\underline{\mu} = 4, \overline{\mu} = 6$.
- x_2: A similar argument as for x_3 leads to the conclusion that the extrema assignment on x_2 is 2.
- x_1: Since the distance to the mean is always larger with x_1 assigned to 1 because $x_1^{\max} = 3 < \underline{\mu} = 4$ we are sure that it is the correct extrema assignment.

Example 8 (Upper bound with extrema assignments). The extrema assignment computed in Example 7 can be used to compute $\overline{opt}(\Pi_2(X))$. In this example, all the extrema assignments could be deduced. Consequently we have $\overline{opt}(\Pi_2(X)) = opt(\Pi_2(X)) = (1-4)^2 + (2-4)^2 + (9-4)^2 = 38$.

For the example on the right of Figure 4 with 50 variables, the algorithm find the optimal solution i.e. $\overline{opt}(\Pi_2(X)) = opt(\Pi_2(X))$. The deduced extrema are indicated with a $\oplus$. The worst case for propagating on σ would correspond to all variables with an identical domain.

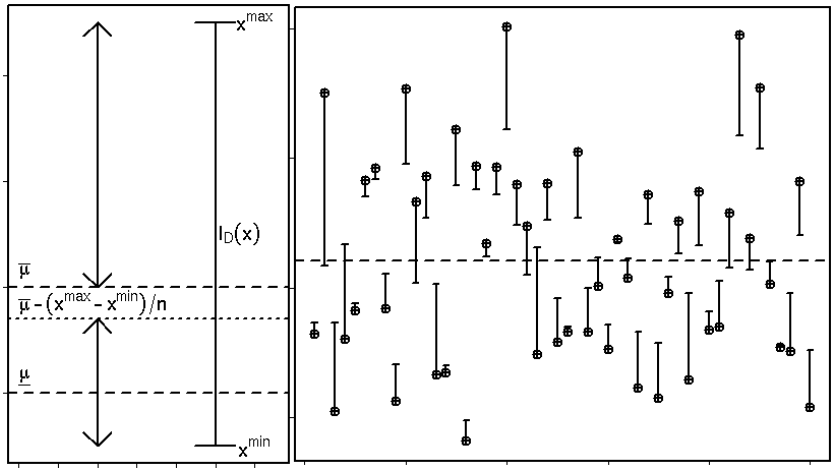


Fig. 4. Left figure: $x = x^{\max}$ because the lower bound on the distance from $x^{\max}$ to μ is smaller than the upper bound on the distance from $x^{\min}$ to μ. Right figure: $\overline{opt}(\Pi_2(X)) = opt(\Pi_2(X))$. The deduced extrema are indicated with a $\oplus$

7 Conclusion

In this paper we have considered a constraint dealing with statistics: the Spread constraint. This constraint and some filtering algorithms associated with it have been proposed by [2]. First, we have shown that simpler filtering algorithms with the same efficiency can be designed. Then, we have studied the main problem on which the constraint is based, that is the minimization of the standard deviation, and we have proved that this problem has a unique optimal value. From this result, we have proposed for the first time an algorithm reducing the values of the mean from the variables and the standard deviation. At last, we have shown that the computation of the maximal value of the standard deviation is NP-hard and we have given an algorithm to compute an upper bound of that value.

References

1. *Problem 30 of CSPLIB (www.csplib.org).*
2. Jean-Charles Regin Gilles Pesant. Spread: A balancing constraint based on statistics. *Lecture Notes in Computer Science*, 3709:460–474, 2005.
3. Lieven Vandenberghe Stephen Boyd. *Convex Optimization.* Cambridge University Press, 2004.

A new Filtering Algorithm for the Graph Isomorphism Problem

Sbastien Sorlin and Christine Solnon

LIRIS, CNRS UMR 5205, bât. Nautibus, University of Lyon I
43 Bd du 11 novembre, 69622 Villeurbanne cedex, France
{sebastien.sorlin,christine.solnon}@liris.cnrs.fr

Abstract. The graph isomorphism problem consists in deciding if two given graphs have an identical structure. This problem may be modeled as a constraint satisfaction problem in a very straightforward way, so that one can use constraint programming to solve it. However, constraint programming is a generic tool that may be less efficient than dedicated algorithms which take advantage of the global semantic of the original problem to reduce the search space.
Hence, we have introduced in [1] a global constraint dedicated to graph isomorphism problems, and we have defined a partial consistency —the label-consistency— that exploits all edges of the graphs in a global way to narrow variable domains. This filtering algorithm is very powerful in the sense that, for many instances, achieving it allows one to either detect an inconsistency, or reduce variable domains to singletons so that the global consistency can be easily checked. However, achieving the label-consistency implies the computation of the shortest path between every pair of vertices of the graphs, which is rather time consuming.
We propose in this article a new partial consistency for the graph isomorphism problem and an associated filtering algorithm. We experimentally show that this algorithm narrow the variable domains as strongly as our previous label-consistency, but is an order faster, so that it makes constraint programming competitive with *Nauty*, the fastest known algorithm for graph isomorphism problem.

1 Introduction

Graphs provide a rich mean for modeling structured objects and they are widely used in real-life applications to represent, *e.g.*, molecules, images, or networks. In many of these applications, one has to compare graphs to decide if their structures are identical. This problem is known as the Graph Isomorphism Problem (GIP).

More formally, a *graph* is defined by a pair (V, E) such that V is a finite set of vertices and $E \subseteq V \times V$ is a set of edges. In this paper, we consider graphs without self-loops, i.e., $\forall (u, v) \in E,\ u \neq v$. Two graphs $G = (V, E)$ and $G' = (V', E')$ are *isomorphic* if there exists a bijective function $f : V \rightarrow V'$ such that $(u, v) \in E$ if and only if $(f(u), f(v)) \in E'$. We shall say that f is

an *isomorphism function*. The GIP consists in deciding if two given graphs are isomorphic.

There exists many dedicated algorithms for solving GIPs such as, e.g., [2–4]. These algorithms are often very efficient (eventhough their worst case complexities are exponential). However, such dedicated algorithms can hardly be used to solve more specific problems, such as isomorphism problems with additional constraints, or larger problems that include GIPs.

An attractive alternative to these dedicated algorithms is to use Constraint Programming (CP), which provides a generic framework for solving any kind of Constraint Satisfaction Problems (CSPs). Indeed, GIPs can be transformed into CSPs in a very straightforward way [5], so that one can use generic constraint solvers to solve them. However, when transforming a GIP into a CSP, the global semantic of the problem is lost and replaced by a set of binary constraints. As a consequence, using CP to solve GIPs may be less efficient than using dedicated algorithms which have a global view of the problem.

In order to allow constraint solvers to handle GIPs in a global way so that they can solve them efficiently without loosing CP's flexibility, we have introduced in [1] a global constraint dedicated to graph isomorphism problems (the *gip* constraint), and we have defined a partial consistency —the label-consistency— and an associated filtering algorithm that exploits all edges of the graphs in a global way to narrow variable domains. This filtering algorithm is very powerful in the sense that, for many instances, achieving it allows one to either detect an inconsistency, or reduce variable domains to singletons so that the global consistency can be easily checked. However, achieving the label-consistency implies the computation of the shortest path between every pair of vertices of the graphs, which is time expensive.

Motivation and outline of the paper. The goal of this paper is to define another partial consistency for the global constraint *gip*: the iterated local label consistency (ILL-consistency). This one is based on an iterated relabelling of the graph vertices and does not need to compute the distance matrix of the graphs. As a consequence, achieving this consistency is less time expensive than achieving the label-consistency.

Section 2 recalls some complexity results for GIPs and an overview of existing approaches for solving these problems. We also recall the definition of the global constraint *gip* and the label-consistency proposed in [1]. In section 3, we introduce the iterated local label consistency (ILL-consistency), a partial consistency for the *gip* constraint based on a relabelling technic of the graph vertices from the direct neighborhood of the vertices. Section 4 experimentally compares label-consistency, ILL-consistency and *Nauty*, the fastest known algorithm for graph isomorphism problem.

2 Approaches for solving graph isomorphism problems

Complexity. The theoretical complexity of the GIP is not exactly stated: the problem is in NP but it is not known to be in P nor to be NP-complete [6] and its own complexity class, *isomorphism-complete*, has been defined. However, some topological restrictions on graphs (e.g., planar graphs [7], trees [8] or bounded valence graphs [9]) make this problem solvable in a polynomial time.

Dedicated algorithms. To solve a GIP, one has to find a one to one mapping between the vertices of the two graphs. The search space composed of all possible mappings can be explored in a "Branch and Cut" way: at each node of the search tree, some graph properties (such as edges distribution, vertices neighbourhood) can be used to prune the search space [4, 2]. This kind of approach is rather efficient and can be used to solve GIPs up to a thousand or so vertices very quickly (in less than one second).

[3] proposes another rather dual approach, which has been originally used to detect graph automorphisms (*i.e.*, non trivial isomorphisms between a graph and itself). The idea is to compute for each vertex v_i a unique label that characterizes the relationships between v_i and the other vertices of the graph, so that two vertices are assigned with a same label if and only if they can be mapped by an isomorphism function. This approach is implemented in the system *Nauty* which is, to our knowledge, the most efficient solver for the graph isomorphism problem. *Nauty* compute a canonical representation of a graph: two graphs have the same representation with Nauty if and only if they are isomorphic. The time needed to solve a GIP with *Nauty* is comparable to "Branch and Cut" methods but *Nauty* is often the quickest for large graphs [10].

Hence dedicated algorithms are very efficient to solve GIPs in practice, even-though their worst case complexities are exponential. However, they are not suited for solving more specific problems, such as GIPs with additional constraints. In particular, vertices and edges of graphs may be associated with labels that characterize them, and one may be interested in looking for isomorphisms that satisfy additional constraints on these labels. This is the case, *e.g.*, in [11] where graphs are used to represent molecules, or in computer aided design (CAD) applications where graphs are used to represent design objects [12].

Constraint Programming. CP is a generic tool for solving constraint satisfaction problems (CSPs), and it can be used to solve GIPs. A *CSP* [13] is defined by a triple (X, D, C) such that :

- X is a finite set of variables,
- D is a function that maps every variable $x_i \in X$ to its domain $D(x_i)$, *i.e.*, the finite set of values that can be assigned to x_i,
- C is a set of constraints, *i.e.*, relations between some variables which restrict the set of values that can be assigned simultaneously to these variables.

Binary CSPs only have binary constraints, *i.e.*, each constraint involves two variables exactly. We shall note $C(x_i, x_j)$ the binary constraint holding between the two variables x_i and x_j, and we shall define this constraint by the set of couples $(v_i, v_j) \in D(x_i) \times D(x_j)$ that satisfy the constraint.

Solving a CSP (X, D, C) involves finding a complete assignment, which assigns a value $v_i \in D(x_i)$ to every variable $x_i \in X$, such that all the constraints in C are satisfied.

CSPs can be solved in a generic way by using constraint programming languages (such as CHOCO [14], Ilog solver [15], or CHIP [16]), *i.e.*, programming languages that integrate algorithms for solving CSPs. These algorithms (called constraint solvers) are often based on a systematic exploration of the search space, until either a solution is found, or the problem is proven to have no solution. In order to reduce the search space, this kind of complete approach is combined with filtering technics that narrow variables domains with respect to some partial consistencies such as Arc-Consistency [13, 17, 18].

Using CP to solve GIPs. Graph isomorphism problems can be formulated as CSPs in a very straightforward way, so that one can use CP languages to solve them [19, 11]. Given two graphs $G = (V, E)$ and $G' = (V', E')$, we define the CSP (X, D, C) such that :

- a variable x_u is associated with each vertex $u \in V$, *i.e.*, $X = \{x_u / u \in V\}$,
- the domain of each variable x_u is the set of vertices of G' that have the same number of adjacent vertices than u, *i.e.*,

$$D(x_u) = \{u' \in V' \ / \ |\{(u, v) \in E\}| = |\{(u', v') \in E'\}|\}$$

- there is a binary constraint between every pair of different variables $(x_u, x_v) \in X^2$, denoted by $C_{edge}(x_u, x_v)$. This constraint expresses the fact that the vertices of G' that are assigned to x_u and x_v must be connected by an edge in G' if and only if the two vertices u and v are connected by an edge in G, *i.e.*,

$$\begin{array}{ll} \text{if } (u, v) \in E, & C_{edge}(x_u, x_v) = E' \\ \text{otherwise} & C_{edge}(x_u, x_v) = \{(u', v') \in V'^2 \mid u' \neq v' \text{ and } (u', v') \notin E'\} \end{array}$$

Once a GIP has been formulated as a CSP, one can use constraint programming to solve it in a generic way, and additional constraints, such as constraints on vertex and edge labels, may be added very easily.

Global constraint and label-consistency. When formulating a GIP into a CSP, the global semantic of the problem is decomposed into a set of binary "edge" constraints, each of them expressing locally the necessity either to maintain or to forbid one edge. As a consequence, using CP to solve GIPs will often be less efficient than using a dedicated algorithm.

To improve the solution process of CSPs associated with GIPs, one can add an *allDiff* global constraint, in order to constrain all variables to be assigned to

different vertices [11]. This constraint is redundant as each binary edge constraint only contains couples of different vertices, so that it will not be possible to assign a same vertex to two different variables. This global constraint allows a constraint solver to prune the search space more efficiently, and therefore to solve GIPs quicker.

However, even with *allDiff* global constraint, CP does not appear to be competitive with dedicated algorithms because most of the global semantic of the problem is still lost. Hence, we have introduced in [1] the global constraint *gip* to define a graph isomorphism problem. We have also defined a partial consistency —the label consistency— that strongly reduces the search space. This partial consistency is based on a labelling of the graph vertices based on the number of vertices at a given distance. We have shown that this partial consistency is very powerful and achieving it generally allows to either detect an inconsistency, or reduce variable domains to singletons so that the global consistency can be easily checked. However, achieving the label-consistency implies the computation of the shortest path between every pair of vertices of the graphs and as a consequence it is time expensive.

3 ILL-consistency

We introduce in this section another filtering algorithm for the graph isomorphism problem global constraint. The main idea of this filtering is to label every vertex with respect to its relationships with the other vertices of the graph. This labelling is "isomorphic-consistent" —in the sense that two vertices that may be associated by an isomorphism function necessary have a same label— so that it can be used to narrow the domains of the variables. These labels are built iteratively: starting from an empty label, each label is extended by considering the labels of its adjacent vertices. This labelling extension is iterated until a fixed point is reached. This fixed point corresponds to a new partial consistency for the graph isomorphism problem.

3.1 Isomorphic-consistent labelling functions

Definition. A *labelling function* is a function denoted by α that, given a graph $G = (V, E)$ and a vertex $v \in V$, returns a label $\alpha_G(v)$. This label does not depend on the names of the vertices but only on the relation defined by E between v and the other vertices of the graph. We shall note $image(\alpha_G)$ the set of labels returned by α for the vertices of a given graph G.

Definition. A labelling function α is *isomorphic-consistent* if for every pair of isomorphic graphs $G = (V, E)$ and $G' = (V', E')$, and for every isomorphism function f between G and G', the vertices that are matched by f have the same labels, *i.e.*, $\forall v \in V, \alpha_G(v) = \alpha_{G'}(f(v))$.

Example. Let us define the labelling function that labels each vertex by its degree, *i.e.*,

$$\forall v \in V, \alpha_{(V,E)}(v) = |\{u \in V, (u,v) \in E\}|$$

This labelling function is isomorphic-consistent as isomorphism functions only match vertices that have a same number of adjacent vertices.

An isomorphic-consistent labelling function can be used to narrow the domains of the variables of a CSP associated with a GIP: the domain of every variable x_u associated with a vertex u can be narrowed to the set of vertices that have the same label than u. We shall say that a labelling function α is stronger than another labelling function α' if it allows a stronger narrowing (or an equivalent narrowing), i.e., if

$$\forall (u,v) \in V^2, \alpha'_G(u) \neq \alpha'_G(v) \Rightarrow \alpha_G(u) \neq \alpha_G(v)$$

3.2 Isomorphic-consistent local relabelling function

We propose to iteratively strengthen an isomorphic-consistent labelling function: at each step, the label of every vertex v is extended with a set of couples (k,l) such that k is the number of vertices that are adjacent to v and that are labelled with l.

Definition. Given a graph $G = (V,E)$ and a labelling function α^i_G for this graph, we define the new labelling function $\alpha^{i+1}_G : V \rightarrow image(\alpha^i_G) \times \wp(\mathbb{N}^* \times image(\alpha^i_G))$ as follows:

$$\forall v \in V, \alpha^{i+1}_G(v) = \alpha^i_G(v) \cdot \{(k,l)\ ,\ k \in \mathbb{N}, l \in image(\alpha^i_G),\\ k = |\{u \in V, (v,u) \in E \wedge \alpha^i_G(u) = l \wedge k > 0\}|\}$$

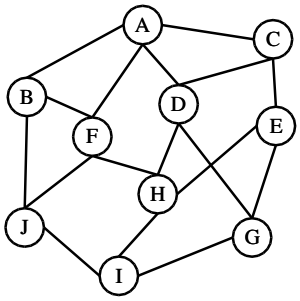


Fig. 1. A graph $G = (V,E)$

Example. For the graph $G = (V, E)$ displayed in figure 1, and the labelling function α^0_G that associates the same empty label $\emptyset$ to every vertex of G, we have $\alpha^1_G(A) = \emptyset \cdot \{(4, \emptyset)\}$ because vertex A is labelled by $\emptyset$ and has four adjacent vertices that are all labelled with $\emptyset$ whereas $\alpha^1_G(B) = \emptyset \cdot \{(3, \emptyset)\}$ because vertex B is labelled by $\emptyset$ and has three adjacent vertices that are all labelled with $\emptyset$.

Theorem 1. *Given an isomorphic-consistent labelling function α^i, the labelling function α^{i+1} is also an isomorphic-consistent labelling function.*

Proof. If α^i is an isomorphic-consistent labelling function then, given the definition of an isomorphic-consistent labelling function, for any pair of isomorphic graphs $G = (V, E)$ and $G' = (V', E')$ and for any isomorphism function f between G and G', $\forall u \in V, \alpha^i_G(u) = \alpha^i_{G'}(f(u))$. Furthermore, as f is an isomorphism function, $\forall (u, v) \in V^2, (u, v) \in E \Leftrightarrow (f(u), f(v)) \in E'$. As a consequence, $\forall u \in V, \forall l \in image(\alpha^i_G), |\{v/(u, v) \in E \wedge \alpha^i_G(v) = l\}| = |\{v'/(f(u), v') \in E' \wedge \alpha^i_{G'}(v') = l\}|$ (because f is a bijective function) and $\forall u \in V, \alpha^{i+1}_G(u) = \alpha^{i+1}_{G'}(f(u))$. The property holds.

A direct consequence of the theorem 1 is that the function α^{i+1} can be used to extend the labels of the vertices of two graphs G and G' without changing the isomorphism properties between G and G'.

Theorem 2. *Given a graph $G = (V, E)$ and a labelling function α^i, the function α^{i+1} is stronger than α^i, i.e.,*

$$\forall (u, v) \in V^2, \alpha^i_G(u) \neq \alpha^i_G(v) \Rightarrow \alpha^{i+1}_G(u) \neq \alpha^{i+1}_G(v)$$

Proof. Straightforward from the fact that each label $\alpha^i(u)$ is a prefix of the label $\alpha^{i+1}(u)$.

A direct consequence of theorem 2 is that, when relabelling the vertices of two graphs G and G' with the function α^{i+1}, the domain of each variable x_v of the CSP corresponding to a GIP between G and G' always has a size inferior or equal than the domain of x_v when the vertices are only labelled by α^i. In other words, the function α^{i+1} can filter the variable domains.

3.3 Iterative local labelling

Relabelling the graph vertices with the function α^{i+1} can introduce more different labels and as a consequence can reduce the variable domains. One can propagate these domain reductions by iterating this relabelling step until a fixed point is reached, *i.e.*, until the number of different labels is not any longer increased.

Starting from an initial isomorphic-consistent labelling α^0, we define a sequence α^1, α^2, α^3, ... of labelling functions such that each step k of this sequence corresponds to a relabelling of the vertices from the labels given at the step $k-1$. Theorem 1 states that each labelling function α^k is isomorphic-consistent, whereas theorem 2 states that each labelling function α^{k+1} is stronger than the previous one α^k. Finally, theorem 3 shows that this sequence reaches a fixed point.

Theorem 3. *Given a graph* $G = (V, E)$*, if* $\exists k \in \mathbb{N}^*$ *such that* $\forall (u, v) \in V^2, \alpha^k_G(u) = \alpha^k_G(v) \Rightarrow \alpha^{k+1}_G(u) = \alpha^{k+1}_G(v)$ *then,* $\forall j \geq k, \forall (u, v) \in V^2, \alpha^k_G(u) = \alpha^k_G(v) \Rightarrow \alpha^j_G(u) = \alpha^j_G(v)$.

Proof. Given its definition, we can see that the function α^{i+1} does not use the labels given by α^i themselves but only an operator of equality between these labels. As a consequence, when a relabelling of the vertices does not change the equality properties between the vertex labels, any further relabelling cannot change these equality properties any more.

Roughly speaking, the theorem 3 shows that, when a step of the sequence α^k does not increase the number of different vertex labels, a fixed point is reached and the relabelling process can be stopped.

Finally, we can trivially show that this fixed point is reached in at most $|V|$ steps.

3.4 ILL-consistency and associated filtering algorithm

We propose to use the sequence of isomorphic-consistent labelling functions defined previously to narrow the domains of the variables of a CSP associated with a GIP. We define the initial labelling function α^0 as the function that associates the same label $\emptyset$ to each vertex. Starting from this initial labelling function, we can then compute a sequence of stronger labelling functions until a fixed point is reached. The last labelling function can be used to define a new partial consistency for a global constraint for the graph isomorphism problem.

Let us recall the syntax proposed in [1] for this global constraint: it is defined by the relation $gip(V, E, V', E', L)$ where

- V and V' are 2 sets of values such that $|V| = |V'|$,
- $E \subseteq V \times V$ is a set of pairs of values from V,
- $E' \subseteq V' \times V'$ is a set of pairs of values from V',
- L is a set of couples which associates one different variable of the CSP to each different value of V, i.e., L is a set of $|V|$ couples of the form (x_u, u) where x_u is a variable of the CSP and u is a value of V, and such that for any pair of different couples (x_u, u) and (x_v, v) of L, both x_u and x_v are different variables and $u \neq v$.

Semantically, the global constraint $gip(V, E, V', E', L)$ is consistent if and only if there exists an isomorphism function $f : V \rightarrow V'$ such that for each couple $(x_u, u) \in L$ there exists a value $u' \in D(x_u)$ so that $u' = f(u)$.

Definition. The global constraint $gip(V, E, V', E', L)$ corresponding to a graph isomorphism problem between $G = (V, E)$ and $G' = (V', E')$ is iterated-local-label consistent (ILL-consistent) if and only if:

$$\forall (x_u, u) \in L,\ \forall u' \in D(x_u), \forall k \in \mathbb{N}\ ,\ \alpha^k_G(u) = \alpha^k_{G'}(u')$$

where α^0 is the labelling function that associates the same label $\emptyset$ to each vertex.

To make the *gip* constraint ILL-consistent, we just have to compute the sequence α^1, α^2, ... of labelling functions for each graph G and G' until a fixed point is reached and to remove from the domain of each variable x_u associated to a vertex $u \in V$ the values $u' \in D(x_u)$ such that $\alpha^k_G(u) \neq \alpha^k_{G'}(u')$.

Note that this process may be stopped before reaching the fixed point. We can use every new labelling function α^i to narrow the domains and, when all the variable domains are reduced to a singleton or when a variable domain becomes empty, the global consistency of the constraint *gip* can be easily checked.

At each step of the sequence, the vertex labels become larger and comparing such labels can be costly in time and in memory. However, one can easily show that these labels can be renamed after each relabelling step, provided that the same name is associated with identical labels in the two graphs. As a consequence, at the end of each relabelling step, labels are renamed with unique integers in order to keep the cost in memory and in time constant at each step of the sequence.

When using appropriate data structures, and provided that labels can be compared in constant time, each relabelling step for a graph $G = (V, E)$ has a time complexity of $\mathcal{O}(|E|)$: for each vertex, one has to look at the labels of the vertices that are adjacent to it. Renaming the labels at each step can be done in a time proportional to the size of the longest label of $image(\alpha)$ (*i.e.*, in the worst case $|E|$). As a consequence, as achieving the ILL-consistency needs at most $|V|$ relabelling steps (in the worst case), the maximum time complexity of our filtering algorithm is $\mathcal{O}(|V| \times |E|)$, the same than for the filtering based on labels [1]. However, we show in section 4 that the average complexity of establishing label-consistency is much more expensive than establishing ILL-consistency.

3.5 Complete example

We propose here a complete example of our relabelling procedure on the graph G of the figure 1. At each step of the sequence, the vertices are renammed by labels l_i. For reason of space, we shall note $\alpha^k_G(S)$ when $\forall (u, v) \in S^2, \alpha^k_G(u) = \alpha^k_G(v)$.

At step 0:

$$\alpha^0_G(\{A, B, C, D, E, F, G, H, I, J\}) = \emptyset$$

The next three relabelling steps successively define α^1, α^2 and α^3 as follows.

$$\begin{array}{lll}
\alpha^1_G(\{A,D,F,H\}) & = & \emptyset.\{(4,\emptyset)\} \Rightarrow l_{1,1} \\
\alpha^1_G(\{B,C,E,G,I,J\}) & = & \emptyset.\{(3,\emptyset)\} \Rightarrow l_{1,2}
\end{array}$$

$$\begin{array}{lll}
\alpha^2_G(\{A,D,F,H\}) & = & l_{1,1}.\{(2,l_{1,1}),(2,l_{1,2})\} \Rightarrow l_{2,1} \\
\alpha^2_G(\{B,C\}) & = & l_{1,2}.\{(1,l_{1,2}),(2,l_{1,1})\} \Rightarrow l_{2,2} \\
\alpha^2_G(\{E,G,I,J\}) & = & l_{1,2}.\{(1,l_{1,1}),(2,l_{1,2})\} \Rightarrow l_{2,3}
\end{array}$$

$$\begin{array}{lll}
\alpha^3_G(A) & = & l_{2,1}.\{(2,l_{2,1}),(2,l_{2,2})\} \Rightarrow l_{3,1} \\
\alpha^3_G(\{B,C\}) & = & l_{2,2}.\{(2,l_{2,1}),(1,l_{2,3})\} \Rightarrow l_{3,2} \\
\alpha^3_G(\{D,F\}) & = & l_{2,1}.\{(1,l_{2,2}),(1,l_{2,3}),(2,l_{2,1})\} \Rightarrow l_{3,3} \\
\alpha^3_G(\{E,J\}) & = & l_{2,3}.\{(1,l_{2,1}),(1,l_{2,2}),(1,l_{2,3})\} \Rightarrow l_{3,4} \\
\alpha^3_G(\{G,I\}) & = & l_{2,3}.\{(1,l_{2,1}),(2,l_{2,3})\} \Rightarrow l_{3,5} \\
\alpha^3_G(H) & = & l_{2,1}.\{(2,l_{2,1}),(2,l_{2,3})\} \Rightarrow l_{3,6}
\end{array}$$

We note that, at step 3, two vertices (A and H) have unique labels. As a consequence, we do not need to relabel these vertices during the next steps. The vertex A (resp. H) keeps the label $l_{3,1}$ (resp. $l_{3,6}$).

$$\begin{array}{lll}
\alpha^4_G(\{B,C\}) & = & l_{3,2}.\{(1,l_{3,1}),(1,l_{3,3}),(1,l_{3,4})\} \Rightarrow l_{4,1} \\
\alpha^4_G(D) & = & l_{3,3}.\{(1,l_{3,1}),(1,l_{3,2}),(1,l_{3,5}),(1,l_{3,6})\} \Rightarrow l_{4,2} \\
\alpha^4_G(E) & = & l_{3,4}.\{(1,l_{3,2}),(1,l_{3,5}),(1,l_{3,6})\} \Rightarrow l_{4,3} \\
\alpha^4_G(F) & = & l_{3,3}.\{(1,l_{3,1}),(1,l_{3,2}),(1,l_{3,4}),(1,l_{3,6})\} \Rightarrow l_{4,4} \\
\alpha^4_G(G) & = & l_{3,5}.\{(1,l_{3,3}),(1,l_{3,4}),(1,l_{3,5})\} \Rightarrow l_{4,5} \\
\alpha^4_G(I) & = & l_{3,5}.\{(1,l_{3,4}),(1,l_{3,5}),(1,l_{3,6})\} \Rightarrow l_{4,6} \\
\alpha^4_G(J) & = & l_{3,4}.\{(1,l_{3,2}),(1,l_{3,3}),(1,l_{3,5})\} \Rightarrow l_{4,7}
\end{array}$$

At step 4, only two vertices (B and C) share the same label.

$$\begin{array}{l}
\alpha^5_G(B) = l_{4,1}.\{(1,l_{3,1}),(1,l_{4,4}),(1,l_{4,7})\} \Rightarrow l_{5,1} \\
\alpha^5_G(C) = l_{4,1}.\{(1,l_{3,1}),(1,l_{4,2}),(1,l_{4,3})\} \Rightarrow l_{5,2}
\end{array}$$

From the step 5, all the vertices have different labels. As a consequence, any graph isomorphism problem involving the graph G of the figure 1 will be solved by our filtering technic.

3.6 Propagation during a search tree process

ILL-filtering does not always reduce every domain to a singleton so that it may be necessary to explore the search space. For example, if all the vertices of the graph have the same degree (*i.e.*, the same number of neighbours), our filtering algorithm is totaly inefficient. Some GIP may have more than one solution (when the graphs are automorph) and as a consequence, some variable domains are not singletons.

When ILL-filtering does not reduce the domain of each variable to a singleton, one has to explore the search space composed of all possible assignments by constructing a search tree. At each node of this search tree, the domain of one variable is splitted into smaller parts, and then filtering technics are applied to

narrow variable domains with respect to some local consistencies. These filtering technics iteratively use constraints to propagate the domain reduction of one variable to other variable domains until either a domain becomes empty (the node can be cut), or a fixed-point is reached (a solution is found or the node must be splitted).

To propagate the domain reductions implied by a search tree assignment, a first possibility is to use the set of C_{edge} constraints as defined in section 2. However, we can still use our filtering method to propagate more strongly the domain reductions. Indeed, assigning a value to a variable corresponds to giving the same unique label to the two corresponding vertices. As a consequence, we can use this new label to restart the relabelling process until it reaches another fixed point.

4 Comparative experimental results

In this section, we compare the efficiency of the label-consistency introduced in [1] and our new ILL-consistency on randomly generated graphs. We also compare these results with the results of *Nauty*, the best known algorithm dedicated to the graph isomorphism problem.

Nauty is a complete algorithm: it always solves a graph isomorphism problem. On the contrary, label-consistency and ILL-consistency are only partial consistencies. However, when labeling vertices by using these consistencies, if there is as many vertex labels than vertices, each variable domain of a GIP involving G becomes a singleton and the global consistency of the CSP is trivially checked.

As a consequence, we choose the following experimental protocol: for each considered graph $G = (V, E)$, we compute the vertex labels with the label-consistency of [1] and the vertex labels with the sequence α until reaching its fixed point. We then count the number of different vertex labels: if there is $|V|$ different labels, any GIP involving G will be perfectly filtered and the problem will be trivially solved.

Note that we only consider non automorphic graphs, *i.e.*, the only existing isomorphism function between G and itself is the identity function. As a consequence, for each considered graph $G = (V, E)$, our algorithm perfectly filters the GIP involving G if and only if the number of different vertex labels is equal to $|V|$.

We consider randomly generated graphs from the Foggia et al.'s benchmark [20]. However, as the number of vertices of the graphs of this benchmark is limited to 1000, we have also generated bigger graphs with a Nauty tool (*genrang*). As a consequence, we consider graphs having between 200 and 10000 vertices and three different edge density: 1%, 5% and 10% (the three densities proposed by [20]). For each size of graphs and each density, the given results are the average results on 100 graphs.

In order to compare the influence of the edge density, we also generate a set of graphs having 1000 vertices and an edge density varying from 1% to 50% (1% by 1%, 100 graphs for each density). We do not test with graphs that have higher

densities because, for the three considered algorithms, it is then more interesting to consider the complementary graph.

4.1 Number of labels

Nauty is a complete algorithm. As a consequence, it always found a perfect filtering (*i.e.*, a different label for each vertex).

On the contrary, label-consistency and ILL-consistency are only partial consistencies and do not find systematicaly a unique label for each vertex. However, except for little graphs with a low density (less than 400 vertices with an edge density of 1%), these two partial consistencies have actually found a perfect labelling of the vertices. As a consequence, label-consistency and ILL-consistency always solve the GIP involving the graphs having more than 400 vertices. Furthermore, the ILL-consistency is obtained at step 2 for all graphs having more than 800 vertices.

$\|V\|$	$\|L_\alpha\|$	T_α	k	$\|L_{label}\|$	T_{label}
200	199,64	0	3,40	191,92	0,01
400	400,00	0	2,88	399,87	0,07
600	600,00	0	2,14	600,00	0,19
800	800,00	0,01	2,01	800,00	0,36

Table 1. Results for the little graphs having a density of 1%. Each line successively displays: the number of vertices $|V|$ of the graphs, the average number of labels $|L_\alpha|$ (resp. $|L_{label}|$) obtained by the ILL-consistency (resp. *label*-consistency), the time needed T_α (resp. T_{label}) in seconds to establish this consistency and k, the average number of steps needed to reach the fix point of the sequence α.

Results for the little graphs having a density of 1% are given into table 1. For some of these graphs, the labeling process does not give an unique label to each vertex. However, the average number of different labels shows an extremly strong reduction of the variable domains. These results also show that ILL-consistency filter the variable domains more strongly than label-consistency and is less expensive to compute.

4.2 Execution time

We compare here the time needed to compute our filtering and to execute Nauty. All tests have be done on a PC at 1,7Ghz and 512MB of RAM running Linux (kernel 2.6).

The first graph of the figure 2 shows that, for the three algorithms, the execution time increases when the density of the graphs increases. The general behavior of the three algorithms is the same whatever the edge density is.

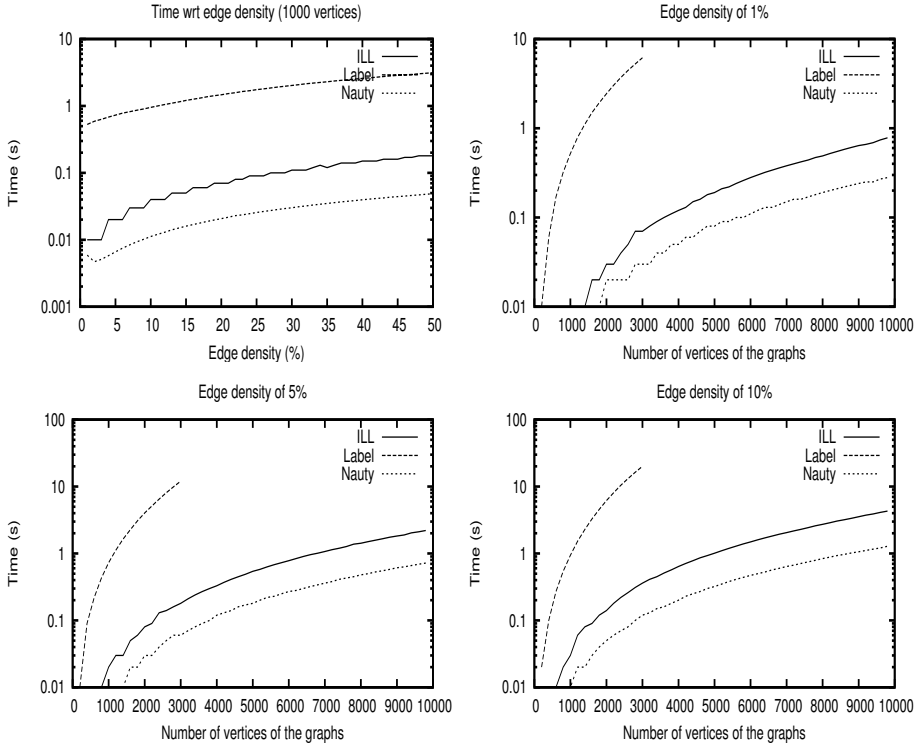


Fig. 2. Execution time w.r.t. edge density for graphs having 1000 vertices and w.r.t. graph size for graph having a density of 1%, 5% or 10%.

The label-consistency is clearly more expensive than the two others filtering technics: we had to interrupt the tests of label-consistency for graphs having up more than 3000 vertices. To establish the label-consistency and the ILL-consistency both have the same worst case complexity. However, as the fix point of the ILL-consistency is reached in only k steps with $k << |V|$, ILL-consistency is generally one order less expensive than label-consistency.

If we compare the ILL-consistency to Nauty, we can show that these two algorithms have a very similar behavior. However, Nauty is generally 3 times as fast as ILL-consistency and is still the best algorithm for graph isomorphism problems. Finally, with 512MB of RAM, our algorithm based on the ILL-consistency begins to swap with graphs having up to 9800 vertices.

5 Conclusion

We have introduced in this paper a new partial-consistency (the ILL-consistency) for the global constraint *gip* defining graph isomorphism problems. This ILL-consistency is based on the computation, for each vertex u, of a sequence of

labels which characterizes the relationship between u and its neighbours that can be viewed as a vertex invariant.

We compare ILL-consistency and label-consistency based on distances between each couple of vertices of the graphs proposed in [1]. These two consistencies are very efficient in the sense where, on randomly generated graphs having up to 400 vertices, achieving them will allow a constraint solver to either detect an inconsistency, or reduce variable domains to singletons so that the global consistency can be easily checked.

These two consistencies have both a theoritical worst case complexity of $\mathcal{O}(|V|\times|E|)$ operations for graphs having $|V|$ vertices and $|E|$ edges. However our experimental results show that ILL-consistency is faster and tigher than label-consistency. Comparing to Nauty, ILL-consistency appear to be competitive. However, Nauty is still 3 times faster than it and is still the fastest algorithm known for graph isomorphism problems.

Our experimentations show that ILL-consistency is strong enough to solve GIP with non automorph randomly generated graphs. Further work will concern the integration of our filtering algorithm into a constraint solver (such as CHOCO [14]), in order to experimentally validate and evaluate it on different kinds of graphs.

References

1. Sorlin, S., Solnon, C.: A global constraint for graph isomorphism problems. In: the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR 2004), Springer-Verlag (2004) 287–301
2. Ullman, J.: An algorithm for subgraph isomorphism. Journal of the Association of Computing Machinery **23**(1) (1976) 31–42
3. McKay, B.: Practical graph isomorphism. Congressus Numerantium **30** (1981) 45–87
4. Cordella, L., Foggia, P., Sansone, C., Vento, M.: An improved algorithm for matching large graphs. In: 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen (2001) 149–159
5. McGregor, J.: Relational consistency algorithms and their applications in finding subgraph and graph isomorphisms. Information Science **19** (1979) 229–250
6. Fortin, S.: The graph isomorphism problem. Technical report, Dept of Computing Science, Univ. Alberta, Edmonton, Alberta, Canada (1996)
7. Hopcroft, J., Wong, J.: Linear time algorithm for isomorphism of planar graphs. 6^{th} Annu. ACM Symp. theory of Comput. (1974) 172–184
8. Aho, A., Hopcroft, J., Ullman, J.: The design and analysis of computer algorithms. Addison Wesley (1974)
9. Luks, E.: Isomorphism of graphs of bounded valence can be tested in polynomial time. Journal of Computer System Science (1982) 42–65
10. Foggia, P., Sansone, C., Vento, M.: A performance comparison of five algorithms for graph isomorphism. In: 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen (2001) 188–199
11. Régin, J.: Dveloppement d'Outils Algorithmiques pour l'Intelligence Artificielle. Application la Chimie Organique. PhD thesis, Univ. Montpellier II (1995)

12. Champin, P.A., Solnon, C.: Measuring the similarity of labeled graphs. In: 5th International Conference on Case-Based Reasoning (ICCBR 2003). Volume Lecture Notes in Artificial Intelligence Nu. 2689 - Springer-Verlag. (2003) 80–95
13. Tsang, E.: Foundations of Constraint Satisfaction. Academic Press (1993)
14. Laburthe, F., the OCRE project team: CHOCO: implementing a CP kernel. In: Proc. of the CP'2000 workshop on techniques for implementing constraint programming systems, Singapore. (2000)
15. ILOG,S.A.: ILOG Solver 5.0 User's Manual and Reference Manual. (2000)
16. Aggoun, A., Beldiceanu, N.: Extending CHIP in order to solve complex and scheduling and placement problems. In: Actes des Journes Francophones de Programmation et Logique, Lille, France. (1992)
17. Mohr, R., Henderson, T.: Arc and path consistency revisited. Artificial Intelligence **28** (1986) 65–74
18. Bessière, C., Régin, J.: Refining the basic constraint propagation algorithm. In Nebel, B., ed.: Proceedings of the seventeenth International Conference on Artificial Intelligence (IJCAI-01), San Francisco, CA, Morgan Kaufmann Publishers, Inc. (2001) 309–315
19. Garey, M., Johnson, D.: Computers and Intractability : A Guide to The Theory of NP-Completness. W.H. Freeman, San Francisco (1979)
20. Foggia, P., Sansone, C., Vento, M.: A database of graphs for isomorphism and subgraph isomorphism benchmarking. 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition (2001) 176 –187