Local Consistency on Conjunctions of Constraints

Christian Bessière[1] and **Jean-Charles Régin**[2]

Abstract. Constraint networks are used more and more to solve combinatorial problems in real-life applications. As pointed out in [BR97], this success requires dealing with non-binary constraints, which are widely needed in real world constraint solvers. Since arc consistency is a fundamental piece of reasoning that seems to be of great help during search for solutions, it is important to have efficient arc consistency algorithms for non-binary constraints. In [BR97], a first step has been done, resulting in a general schema for arc consistency on any kind of constraints, and in instantiations of the schema to handle the more general non-binary constraints, i.e., those given as a predicate or as a set of allowed/forbidden combinations of values. In this paper, we extend that work by proposing an efficient way to compute local consistency on conjunctions of constraints (called *conjunctive consistency*). We then use this notion to improve arc consistency processing on two practical cases, *embedding* constraints, which can be found in any network, and *conjunctive* constraints, which are often present in problems modelization.

1 Introduction

The constraint satisfaction problem (CSP) involves finding a solution to a constraint network, i.e. finding values for problem variables subject to constraints. As the general task of solving a CSP is NP-hard, many researchers have concentrated on improving the efficiency of finding a solution in a constraint network. In particular, arc consistency has caught many researchers' attention.

However, results about CSP solving and arc consistency are almost always limited to *binary* constraint networks, as justified by the fact that any non-binary constraint network can be translated into an equivalent binary one with additional variables [JJNV89, RPD90, Dec90]. But, in practical cases, it is often inconceivable to translate a non-binary constraint into an equivalent set of binary ones with the techniques cited above because of the computational and memory cost.

Hence, in [BR97], a general framework has been proposed, in which one can incorporate any specific algorithm to calculate arc consistency on any given kind of constraint, including non-binary ones, as is widely required in real-world constraint solvers. This framework, `GAC-schema`, is powerful enough to take into account multidirectionality [BFR95, BR97], and is based on the notion of single support [Bes94], which is essential for efficiency. Bessière and Régin also provided three instantiations of the schema to efficiently handle constraints defined by a list of allowed tuples, by a list of forbidden tuples, and most notably, by any predicate. The usefulness of these algorithms has already been shown on some practical problems [Rég98].

However, `GAC-schema` and its current instantiations, although general enough to be able to deal with constraints defined by any predicate, are incapable of using specific knowledge (if available) that underlies the constraints it handles. Suppose that in a given constraint network you have four variables, x_1, x_2, x_3, and x_4, each representing a piece of a logo to design, which are constrained by the two constraints $C(x_1, x_2, x_3, x_4)$ and $x_1 = x_4$. Imagine that C must ensure that the combination of colors taken by x_1, x_2, x_3, x_4 is accepted by the designer. Using `GAC-schema` and its instantiation to predicates to achieve arc consistency on C, you will guarantee never asking the designer the same question twice and never asking him something you can deduce from the previous answers. But you will not use the knowledge contained in $x_1 = x_4$. Thus if the variables take their values in the 7 rainbow colors, you will pick your questions in a space of 7^4 possibilities instead of 7^3 when using $x_1 = x_4$. More important, when performing arc consistency on C and $x_1 = x_4$ separately, as in `GAC-schema`, you may miss some local inconsistencies that would be detected when using $x_1 = x_4$ during the processing of C. The use of $x_1 = x_4$ when processing C would permit achieving a local consistency stronger than arc consistency. (Suppose one of the requirements of the designer is that all the pieces must have different colors. Arc consistency, and so `GAC-schema`, will not remove any value, but a local consistency involving $x_1 = x_4$ when processing C would find an inconsistency.)

There is a second case in which achieving a local consistency that involves several constraints would be useful. Indeed, in practical applications, it sometimes appears that a constraint C between a set of variables is defined by the user as a conjunction of several subconstraints, each involving a subset of the variables of C. The usual way of encoding that is to consider the subconstraints as classical constraints of the network, the constraint C being just some higher level knowledge used to help the modelization of the problem, but not present in the network. As a result, there is some global knowledge present during the modelization, which is missing in the network. If we achieve arc consistency, some inconsistencies can be missed, that would be detected if C was handled as a single global constraint.

Related work Relational arc consistency is another local consistency which has been proposed for non-binary constraints [vBD95]. Unfortunately, it is very expensive when

[1] LIRMM–CNRS (UMR 5506), 161 rue Ada, 34392 Montpellier, France, Email: bessiere@lirmm.fr
[2] ILOG S.A., Sophia Antipolis, 06560 Valbonne, France, Email: regin@ilog.fr

ECAI 98 Workshop Programme.

used on practical problems with non-binary constraints of great arity since relational arc consistency on a constraint C implies global consistency on the network restricted to the variables of C. Therefore, enforcing relational arc consistency produces many new constraints that have to be represented in extension since their semantics has few chances to be easy to extract.

In this paper, after a recall on the `GAC-schema`, we define a new local consistency, which is the extension of arc consistency to several (a conjunction of) constraints taken globally[3]. We call it *conjunctive consistency.* (Multi-arc consistency could have been an adequate name too but it has a different meaning in graph theory.) Afterwards, we propose an algorithm based on the `GAC-schema` to compute conjunctive consistency on a conjunction of constraints. Then, we show how conjunctive consistency can be directly applied to fit the two special cases which motivated that work. The first of these two cases is local consistency on an *embedding* constraint, i.e., a constraint involving a superset of the variables of other constraints (the constraint $C(x_1, x_2, x_3, x_4)$ in the example above is an embedding constraint). The second is arc consistency on constraints defined as conjunctions of subconstraints. Finally, some short experiments show the behavior of the method on a sample problem.

2 Preliminary Definitions

Constraint network. A finite *constraint network* $\mathcal{P} = (X, \mathcal{D}, \mathcal{C})$ is defined as a set of n *variables* $X = \{x_1, \ldots, x_n\}$, a set of *domains* $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ where $D(x_i)$ is the finite set of possible *values* for variable x_i, and a set $\mathcal{C}$ of *constraints* between variables.

A total ordering $<_d$ can be defined on $D(x_i), \forall x_i \in X$, without loss of generality.

Constraint. A constraint C on the ordered set of variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ is a subset of the Cartesian product $D(x_{i_1}) \times \cdots \times D(x_{i_r})$ that specifies the *allowed* combinations of values for the variables $x_{i_1}, \ldots, x_{i_r}$. An instantiation of the variables in $X(C)$ is called a *tuple on* $X(C)$. Two tuples τ and τ' on $X(C)$ can be ordered by the natural lexicographic order $\prec_{lo}$ in which $\tau \prec_{lo} \tau'$ iff $\exists k / \tau[1..k-1] = \tau'[1..k-1]$ and $\tau[k] <_d \tau'[k]$ ($\tau[1..k]$ being the prefix of size k of τ, and $\tau[k]$ the k^{th} value of τ). The tuples on $X(C)$ not allowed by C are called the *forbidden* tuples of C. Verifying whether a given tuple τ is allowed by C or not is called a *constraint check.* $|X(C)|$ is the *arity* of C.

As stated in [BR97], a constraint C involving the subset of variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ can be defined by the set of allowed/forbidden tuples given in extension. It can also be represented by any Boolean function (predicate) f_C defined on $D(x_{i_1}) \times \cdots \times D(x_{i_r})$. Less generally, but of great use in practical applications, it can be represented by a conjunction of subconstraints, namely a conjunctive constraint.

Notation. A value a for a variable x is often denoted by (x, a). $\mathsf{var}(C, i)$ represents the ith variable of $X(C)$, while $\mathsf{index}(C, x)$ is the position of variable x in $X(C)$.

[3] Even if this local consistency seems to be a "global" one, we still use the word "local", which is consistent here with its meaning in the CSP community, the word global having a completely different meaning when we speak about consistencies.

Algorithm 1: function `propagation`

```
propagation (C: constraint; x: variable; a: value;
             in out deletionStream: list): boolean
  for each τ ∈ S_C(x,a) do
    for each (z,c) ∈ τ do remove τ from S_C(z,c);
    for each (y,b) ∈ S(τ) do
      remove (y,b) from S(τ);
      if b ∈ D(y) then
1       σ ← seekInferableSupport(C, y, b);
2       if σ ≠ nil then add (y,b) in S(σ) ;
        else
          σ ← seekNextSupport(C, y, b, last_C(y,b));
          if σ ≠ nil then
            add (y,b) in S(σ) ;
            last_C(y,b) ← σ ;
            for k from 1 to |X(C)| do
              add σ in S_C(var(C,k), σ[k]) ;
          else
            remove b from D(y) ;
            if D(y) = ∅ then return false ;
            add (y,b) in deletionStream ;
  return true ;
```

Arc consistency. Let $\mathcal{P} = (X, \mathcal{D}, \mathcal{C})$ be a constraint network, C a constraint in $\mathcal{C}$. A tuple τ on $X(C)$ is *valid* if $\forall (x, a) \in \tau, a \in D(x)$; otherwise, it is *rejected.* A value $a \in D(x)$ is *consistent with* C iff $x \notin X(C)$, or $\exists \tau$ allowed by C, such that $a = \tau[\mathsf{index}(C, x)]$ and τ is valid. (τ is then called a *support* for (x, a) on C.) C is *arc consistent* iff $\forall x_i \in X(C), D(x_i) \neq \emptyset$ and $\forall a \in D(x_i), a$ is consistent with C. $\mathcal{P}$ is *arc consistent* iff all the constraints of $\mathcal{C}$ are arc consistent. We achieve arc consistency in $\mathcal{P}$ by removing every value not consistent with at least one constraint in $\mathcal{C}$.

3 The `GAC-schema`

Let us review the `GAC-schema` presented in [BR97] since the algorithms we will propose in the following sections are instantiations of that schema.

`GAC-schema` has been built to take into account the *multidirectionality* inherent in any constraint when achieving arc consistency. Multidirectionality is the fact that if τ is a support for a value (x, a) on a constraint C, then it is also a support on C for all the other values composing it.

The function `propagation` is given in Algorithm 1. The program including it must create and initialize the data structures (S_C, S, and $last_C$), and call `propagation`$(C, x, a, \mathsf{deletionStream})$ for each constraint C involving x, each time a value (x, a) is removed from $D(x)$, in order to propagate the consequences of this deletion. The $\mathsf{deletionStream}$ is updated to contain the deleted values not yet propagated. The search for support (function `seekNextSupport`) is instantiated differently depending on the type of the constraint involved. S_C, S, and $last_C$ must be initialized in a way such that:

- $S_C(x, a)$ contains all the allowed tuples τ that are the current support for some value, and such that $\tau[\mathsf{index}(C, x)] = a$.
- $S(\tau)$ contains all values for which τ is the current support.

Algorithm 2: function `seekInferableSupport`

`seekInferableSupport` (C: constraint; y: variable; b: value)**:** tuple

while $S_C(y, b) \neq \emptyset$ **do**
  $\sigma \leftarrow first(S_C(y, b))$;
  if σ *is valid* **then return** σ /* σ is a support */ ;
  else remove σ from $S_C(y, b)$;
return nil ;

- $last_C(y, b)$ is the last tuple returned by `seekNextSupport` as a support for (y, b) if `seekNextSupport`$(C, y, b, -)$ has already been called; **nil** otherwise. There is an ordering on the tuples, which is proper to `seekNextSupport`, and $last_C(y, b)$ gives the point where `seekNextSupport` will have to restart the search for a support for (y, b) on C at the next call.

We say that `GAC-schema` takes into account multidirectionality because it never checks whether a tuple is a support for a value when it has already been unsuccessfully checked for another value (use of $last_C$ in function `seekNextSupport`), and never looks for a support for a value on a constraint C when a tuple supporting this value has already been checked (use of S_C in function `seekInferableSupport`–see Algorithm 2).

We now review the instantiation of `GAC-schema` for constraints given as a predicate, because the implementation of conjunctive consistency filtering will be based on it.

When a constraint C is defined by a predicate for which no particular semantics are known, it is necessary to define an ordering on the tuples that will be followed when looking for a support for a value (y, b) on C. By means of the function `nextTuple` (Algorithm 3), the lexicographic order $\prec_{lo}$ defined previously is used to examine the valid tuples on $X(C)$.

Algorithm 3: function `nextTuple`

`nextTuple` (C: constraint; y: variable; b: value; τ: tuple; k: index)**:** (tuple, index)

/* **returns** (σ, k') where:
- σ is the smallest valid tuple such that $\sigma[\mathsf{index}(C, y)] = b$, $\tau \prec_{lo} \sigma$ and $\sigma[1..k] \neq \tau[1..k]$
- k' is such that $\sigma[1..k'-1] = \tau[1..k'-1]$ and $\sigma[k'] \neq \tau[k']$

otherwise **returns** (**nil**,–) */

The function `seekNextSupport` (not given here, but which can be found in [BR97]) returns the smallest support for (y, b) on C greater than τ. It is called when we know that there does not exist any tuple already checked that supports (y, b) on C (lines 1-2 of `propagation`). To deal completely with multidirectionality requires checking only valid tuples that have never been checked, which we call *candidates*. This is the main subtlety of `seekNextSupport`, and it is done by means of the function `seekCandidateTuple`(C, y, b, σ, k) (see Algorithm 5 in [BR97]). It returns the smallest (w.r.t.$\prec_{lo}$) candidate greater than or equal to σ, with value b at $\mathsf{index}(C, y)$. The following property illustrates the behavior of `seekCandidateTuple`.

Property 1 *Let σ be a valid tuple on $X(C)$. If $\exists x \in X(C)$ such that $\sigma \prec_{lo} last_C(x, \sigma[\mathsf{index}(C, x)])$ then σ has already been checked as forbidden by C, and so is not a candidate.*

Proof. Suppose σ is allowed by C. Then, σ would be a support for $(x, \sigma[\mathsf{index}(C, x)])$. Now, $\sigma \prec_{lo} last_C(x, \sigma[\mathsf{index}(C, x)])$, which leads to a contradiction.

Remark. Notice that a valid tuple $\sigma = last_C(x, a)$ for some (x, a) cannot be considered by `seekCandidateTuple` $(C, y, b, \sigma, -)$ because `seekInferableSupport` would have found that σ is a support for (y, b) before calling `seekNextSupport`.

4 Conjunctive consistency

In this section, we define what we call conjunctive consistency, and present an instantiation of `GAC-schema` (improperly named here) that achieves conjunctive consistency.

Definition 1 (Conjunctive consistency) *A set of constraints $Conj = \{C_1, \ldots, C_k\}$ on the ordered set of variables $X(Conj) = (x_{i_1}, \ldots, x_{i_r})$ (with $\{x_{i_1}, \ldots, x_{i_r}\} = \bigcup_{C_j \in Conj} X(C_j)$) is* conjunctive consistent *iff $\forall x_i \in X(Conj), D(x_i) \neq \emptyset$, and $\forall a \in D(x_i)\ \exists$ a tuple τ on $X(Conj)$* supporting *(x_i, a), i.e., which satisfies all the constraints in $Conj$.*

Property 2 *A set of constraints $Conj$ is conjunctive consistent iff the subnetwork $(X(Conj), \mathcal{D}_{X(Conj)}, Conj)$ is variable completable [Fre91] ($\mathcal{D}_{X(Conj)}$ being the set of current domains of the variables in $X(Conj)$).*

Definition 2 (Conjunctive consistency of a network) *Let $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a constraint network, and $\mathcal{C_J} = \{Conj_1, Conj_2, \ldots\}$ be a covering of $\mathcal{C}$. $\mathcal{P}$ is* conjunctive consistent *w.r.t. $\mathcal{C_J}$ iff $\forall Conj_i \in \mathcal{C_J}$, $Conj_i$ is conjunctive consistent.*

Achieving conjunctive consistency could be done by means of any search algorithm since looking for a support for a value (x, a) in a set $Conj$ of constraints corresponds to looking for a solution in the subnetwork $(X(Conj), \mathcal{D}_{X(Conj)}, Conj \cup \{x = a\})$. However, our final goal is to apply conjunctive consistency to two special kinds of constraints (see Sections 5 and 6) in which there are many common variables between the constraints (i.e., $X(Conj)$ is equal or close to $X(C)$ for some $C \in Conj$). So, using the property of multidirectionality on $X(Conj)$ should not imply a too big overhead compared to multidirectionality on $X(C)$, which in any case is used for arc consistency. We will then propose an algorithm that uses multidirectionality during the search for support because we think it can save much useless effort.

In order to use `GAC-schema` to achieve conjunctive consistency on a set $Conj$ of constraints, the parameter C representing a constraint in the algorithms implementing `GAC-schema` (Algorithms 1, 2, and 3) must be replaced by $Conj$ the set of constraints involved, and $X(C)$ by $X(Conj)$.

4.1 A First Algorithm

The first algorithm we propose to implement the function `seekNextSupport` needed for conjunctive consistency is closely related to `seekCandidateTuple`. (In fact, the only thing done in the `seekNextSupport` for predicates is to call `seekCandidateTuple` and to test whether a candidate is allowed by C or not.) An invariant of `seekCandidateTuple`(C, y, b, σ, k) is that σ is consistent with property 1 on the $k-1$ first variables of $X(C)$. `seekCandidateTuple` proceeds step by step, verifying that σ does

Algorithm 4: function seekNextSupport

```
seekNextSupport (Conj: set of const; y: variable; b:
                 value; σ: tuple): tuple
  begin
      /* steps to the next valid tuple */;
      if σ ≠ nil then
          (σ, dummy) ← nextTuple(Conj, y, b, σ, |X(Conj)|);
      else
          for i from 1 to |X(Conj)| do
              σ[i] ← first(D(var(Conj, i))) ;
          σ[index(Conj, y)] ← b ;
      k ← 1;
      while (σ ≠ nil) and (k ≤ |X(Conj)|) do
          /* σ is candidate till index k − 1 */;
          k_new ← |X(Conj)| + 1;
          if last_Conj(var(Conj, k), σ[k]) ≠ nil then
1             λ ← last_Conj(var(Conj, k), σ[k]) ;
              split ← 1 ;
              while σ[split] = λ[split] do
                  split ← split + 1 ;
2             if σ[split] < λ[split] then
                  if split < k then
                      (σ, k_new)                              ←
                      nextTuple(Conj, y, b, σ, k) ;
                  else
                      (σ, k_new)                              ←
                      nextTuple(Conj, y, b, λ, |X(Conj)|) ;
          if k_new ≤ k then  k ← k_new;
          else
3             if check(Conj, y, b, σ[1..k]) then  k ← k + 1;
4             else  (σ, k) ← nextTuple(Conj, y, b, σ, k) ;
      return σ ;
  end
```

not violate property 1 on $\mathsf{var}(C, k)$ before going to $k+1$, making a kind of chronological backtracking (BT) otherwise. `seekCandidateTuple` is in fact very similar to BT, checking the values of σ w.r.t. property 1 one by one, thus completely using multidirectionality, and only terminating when a candidate is found, i.e., an instantiation of $X(C)$ satisfying the "constraint" of property 1.

By seeing `seekCandidateTuple` in this way, it is easy to fit it to conjunctive consistency. When looking for a support for a value (y, b) on a set of constraints $Conj$, the instantiation/tuple which is sought is now a tuple on $X(Conj)$ satisfying both property 1 (i.e., being a candidate for $Conj$) and the constraints of $Conj$. It is a candidate consistent with $Conj$, so a support. At each step k of the function `seekCandidateTuple` for predicates, we have to check whether $\sigma[1..k]$ is consistent with $Conj$ in addition to verifying property 1. This is presented in the function `seekNextSupport` (see Algorithm 4).

`seekNextSupport`$(Conj, y, b, \sigma)$ returns the smallest (w.r.t.-$\prec_{lo}$) support greater than or equal to σ, with value b at $\mathsf{index}(Conj, y)$. Like `seekCandidateTuple` for predicates, `seekNextSupport` verifies for each index whether σ is greater than $last_{Conj}(\mathsf{var}(Conj, k), \sigma[k])$, (denoted by λ) (lines 1 to 2). In addition, it has to check whether or not $\sigma[1..k]$ is consistent with $Conj$ (lines 3 to 4).

The function `check` performs the same work as that done at a given level of the search in any backtrack-search procedure, i.e., checking whether the current instantiation is consistent or not. Indeed, the only requirement is that `check`$(Conj, y, b, \sigma[1..k])$ returns true iff $\sigma[1..k]$ is a "possible" partial solution to the subnetwork $(X(Conj), \mathcal{D}_{X(Conj)}, \{y = b\} \cup Conj)$.

It must be pointed out that what a "possible" partial solution is depends on which algorithm is used in `check`. A function `check` containing some look ahead, such as forward checking (FC), can detect that $\sigma[1..k]$ is inconsistent while another one, only checking backwards, such as BT, would need a few more steps (k growing) to find the same inconsistency.

4.2 Improvement

In the technique to achieve conjunctive consistency we presented in the previous subsection, no matter which look ahead algorithm is used in `check`, the verification of property 1 is done independently of `check`, and remains based on BT. It is a pity when you use an algorithm with some look ahead inside the function `check`, to be obliged to remain based on BT to test property 1. Satisfying property 1 is, in fact, a particular constraint that guarantees that multidirectionality is used completely during the search for support in $Conj$. The only reason it is processed independently, in BT behavior, is that property 1 states what a non candidate tuple/instantiation is, but does not state what an "impossible" value (i.e., a value that cannot belong to any candidate) is. As in BT, a value has to be added in a partial instantiation to detect it as non candidate by property 1. The following property gets rid of this drawback.

Property 3 *Let $highest(D(x_i))$ denote the highest value in $D(x_i)$ w.r.t. $<_d$. Let $X(Conj) = (x_{i_1}, \ldots, x_{i_r})$. Let $k \in [1..r]$. Let $a \in D(x_{i_k})$. If $\exists s < k$ such that $\forall j \in [1..s-1], highest(D(x_{i_j})) = last_{Conj}(x_{i_k}, a)[j]$, and $highest(D(x_{i_s})) <_d last_{Conj}(x_{i_k}, a)[s]$ then (x_{i_k}, a) cannot belong to any candidate on $Conj$ w.r.t. the current domains $D(x_{i_1}), \ldots, D(x_{i_r})$.*

Proof. Suppose (x_{i_k}, a) belongs to a candidate σ. By hypothesis, $\sigma \in D(x_{i_1}) \times \cdots \times D(x_{i_r})$ implies that $\sigma <_{lo} last_{Conj}(x_{i_k}, a)$, which is impossible for a candidate.

With property 3 it is now possible to predict that a value (x_{i_k}, a) cannot belong to any candidate in $D(x_{i_1}) \times \cdots \times D(x_{i_r})$ without instantiating all the variables from x_{i_1} to x_{i_k}. Therefore, it is possible to build a `propagation` procedure specific to the constraint of satisfying multidirectionality. This constraint, say M_{Conj}, with its own propagation scheme, can be considered as a constraint added to $Conj$, and dealt with as such. And since M_{Conj} ensures that multidirectionality is used completely, we can get rid of the special processing of multidirectionality in `seekNextSupport`. The new function `seekNextSupport2` can be any look ahead search algorithm such as FC or MAC [SF94]. The specification of `seekNextSupport2`$(Conj, y, b, \sigma)$ is that it must return a solution to the subnetwork $(X(Conj), D_{X(Conj)}, \{y = b\} \cup Conj \cup M_{Conj})$, knowing that σ is the smallest (w.r.t. $<_{lo}$) possible solution[4]. A solution to this subnetwork is a candi-

[4] Notice that starting a search algorithm with an instantiation for which we know there is no solution before it has already been studied in [Van90].

date and a support for (y, b), and if the ordering $\prec_{lo}$ has been followed, we know that this is also the smallest support for (y, b). Following the ordering $\prec_{lo}$ (i.e., using the ordering on $X(Conj)$ to instantiate variables and $<_d$ to choose values) is also required by the constraint M_{Conj} (see property 3). The `propagation` algorithm associated with M_{Conj} is given in Appendix. It is the exact application of property 3, propagating deletions only when a value that was the highest of its domain is deleted. The following example illustrates the advantage of `seekNextSupport2` against `seekNextSupport`.

Example. Let us take the same problem as in the introduction, but with a different designer, whose requirements are represented by $C'(x_1, x_2, x_3, x_4)$, and with the constraint $x_2 < x_3$. Suppose we are enforcing conjunctive consistency on $C_g = \{C', x_2 < x_3\}$, and at some moment during the conjunctive consistency process on C_g $D(x_1) = D(x_2) = D(x_3) = D(x_4) = \{a, b, c, d\}$ (the first four colors of the rainbow), and $last_{C_g}(x_4, c) = (d, c, d, c)$. Suppose (x_3, c) has lost its current support. When looking for a new support for (x_3, c), we will seek consistent candidates in the space $D(x_1) \times D(x_2) \times \{c\} \times D(x_4)$. If some look ahead is used in `seekNextSupport` and `seekNextSupport2` (arc consistency for example), `seekNextSupport2` will immediately detect (i.e., without any search) that (x_4, c) cannot belong to a consistent candidate for (x_3, c) because arc consistency removes (x_2, d) and (x_2, c) thanks to the constraint $x_2 < x_3$, and then removes (x_4, c) thanks to the constraint M_{C_g} (because $highest(D(x_1)) = last_{C_g}(x_4, c)[1]$ and $highest(D(x_2)) <_d last_{C_g}(x_4, c)[2]$). In `seekNextSupport`, (x_4, c) is not removed by the look ahead used in `check` (constraint M_{C_g} not used), and thus is unsuccessfully tried several times during the search for a support. Furthermore, other deletions or inconsistency detections implied by the propagation of the deletion of (x_4, c) are missed by `seekNextSupport` [5].

4.3 Analysis

First, let us report the worst-case space complexity of the algorithms we proposed to achieve conjunctive consistency on a conjunction $Conj$ of constraints. In the case of `seekNextSupport`, the only data structures used are those of `GAC-schema`. The worst-case space complexity is then in $O(|X(Conj)|^2 \cdot d)$, where d is the size of the largest domain (see [BR97]). When we use `seekNextSupport2`, some additional data structures are used to implement the propagation procedure of the multidirectionality constraint M_{Conj}. But their size is less than $|X(Conj)|^2 \cdot d$ (see the Appendix), so the space complexity remains in $O(|X(Conj)|^2 \cdot d)$. The worst-case time complexity is bounded above by the complexity of the search procedure used to search for supports (i.e., a polynomial in $|X(Conj)|$).

The ordering of the variables that is used in the algorithms we presented must be a static variable ordering (SVO). This SVO is required to deal with the constraint of multidirectionality. The ordering $<_{lo}$ on the tuples is defined w.r.t. the ordering on $X(Conj)$. With the possibility of using look ahead search algorithms such as FC or MAC to search for supports on $X(Conj)$, it can seem a strong limitation to use them with a SVO. But we have to keep in mind that this restriction holds only for the subnetwork defined on $X(Conj)$ by the constraints in $Conj$. Furthermore, "maximum degree" is a SVO which usually shows relatively good performances on structured networks, and it can be used to define the ordering of $X(Conj)$ on $Conj$. Nevertheless, in cases where a SVO in the search for support would be a too strong limitation, we can relax the constraint of using complete multidirectionality by forgetting property 1 (or 3), then replacing `seekNextSupport` (or `seekNextSupport2`) by any search algorithm with dynamic variable ordering (DVO) (but without the constraint M_{Conj}). The counterpart is that only one side of multidirectionality is used, the one performed in `seekInferableSupport` (i.e., never seeking a support for a value when a tuple supporting it has already been checked). We cannot guarantee that we never check whether a tuple is a support for a value when it has already been unsuccessfully checked for another value.

5 Embedding Constraints

We have an algorithm to compute conjunctive consistency on any set of constraints. We will now discuss its adaptation to embedding constraints.

Definition 3 (Embedding constraint) *A constraint C on $X(C)$ is an* embedding constraint *iff $\exists c \in \mathcal{C} / X(c) \subset X(C)$. c is a constraint underlying C. The set of all the constraints underlying C in the network is denoted by $Under(C)$.*

As shown in the example of Section 1, when a constraint C is an embedding constraint, and when arc consistency is enforced on C by the instantiation of `GAC-schema` for predicates[6], the use of the knowledge of the underlying constraints has two advantages. The first one is to avoid considering, in the embedding constraint, combinations of values that are forbidden by some underlying constraint, thus improving time efficiency. The second advantage is that the local consistency obtained, conjunctive consistency, is stronger than arc consistency since several constraints are considered globally. It is thus less expensive and more pruningful to apply conjunctive consistency on $C \cup Under(C)$ than arc consistency separately on C and the constraints underlying C.

6 Conjunctive Constraints

In problem modelization, a conjunctive constraint is a set of several subconstraints each involving a part of a particular set of variables on which they are circumscribed and on which their conjunction defines a global constraint. Encoding such a constraint is not difficult. It is sufficient to add all the subconstraints in the network to completely define the conjunctive constraint. However, we loose some global knowledge on the constraint, and an algorithm achieving arc consistency on each subconstraint taken separately will not remove as many values as if the whole set is taken globally. On the other hand, encoding the conjunctive constraint as a single predicate that

[5] Keep in mind that these deletions, performed during the search for a consistent candidate for (x_3, c) are done just in this context. This does not mean they do not belong to the final domains after conjunctive consistency enforcement on C_g. It only means that they won't belong to a support of (x_3, c).

[6] In a *positive* embedding constraint (i.e., defined by the set of allowed tuples explicitly given), it would be irrelevant to try to save unsuccessful constraint checks by using multidirectionality or underlying constraints since the tuples tested are all successful ones.

returns true iff all the subconstraints (not necessarily added in the network) are satisfied permits to keep the global knowledge of the constraint. But the underlying knowledge is lost, and then, as for embedding constraints, arc consistency can become much more expensive. However, thanks to the following property, we can both use the underlying knowledge of the subconstraints and save the global knowledge of the conjunction by using conjunctive consistency instead of arc consistency.

Property 4 *Let C be a conjunctive constraint such that $C \equiv C_1 \wedge C_2 \wedge \ldots \wedge C_k$, $\{C_i\}_{\forall i \in [1..k]}$ being the set of subconstraints defining C. Achieving arc consistency on C is equivalent to achieving conjunctive consistency on $\{C_i\}_{\forall i}$.*

7 Experimental results

We report in this section the first results we obtained with our improved algorithm `seekNextSupport2` on a problem used in cryptography. It consists of a square with r rows and r columns, each row i, resp. each column j, being associated with an integer $rsum(i)$, resp. $csum(j)$. The goal is to find a key, i.e., to put the integers 1 to r^2 in the cells of the square in such a way that the sum of the values affected to row i is equal to $rsum(i)$, and the sum of the values affected to column j is equal to $csum(j)$, $\forall i, j \in [1..r]$. (We dealt only with satisfiable instances.)

The associated constraint network involves r^2 variables representing the r^2 cells. Each variable takes its values in the domain $\{1..r^2\}$. The standard encoding (denoted by stand) involves one constraint, C_S, on each row and each column, to guarantee that the sums are respected, and an *alldiff* constraint [Rég94] on all the variables. In our new encoding (denoted by conj), we replace each constraint C_S constraining the sum of a row or a column, by a constraint $C_{S\neq}$, which is the conjunction of C_S and of an *alldiff* constraint $C_{\neq}$ involving the same r variables as C_S. Thanks to property 4, enforcing arc consistency on $C_{S\neq}$ is done by achieving conjunctive consistency on $\{C_S, C_{\neq}\}$. The *alldiff* constraint involving all the variables is kept[7].

The results presented in the following are obtained with a MAC search procedure with the "minimum domain" DVO. The procedure used to search for supports inside the arc consistency process of a conjunctive constraint (i.e., `seekNextSupport2`) is also a MAC procedure, which we call the "internal" one. It uses a lexicographic[8] SVO. These MAC procedures are those of ILOG Solver [ILO98] without any special features except those presented in this paper. We can point out that the experimentation is presented as the comparison of behavior of MAC on two different encodings of the same problem (one with and one without conjunctive constraints). Another completely equivalent way would have been to say that we compare two different algorithms, one maintaining arc consistency, and one maintaining conjunctive consistency, on the same network, where constraints C_S and $C_{\neq}$ are grouped by pairs for the second algorithm.

Table 1 contains the results we obtained on the 6×6 square.

	# backtracks$_{(+internal)}$		cpu time (sec.)	
	stand	conj	stand	conj
average	75,548	$281_{(+24,276)}$	26.2	6.7
hardest	1,623,557	$2{,}598_{(+110,264)}$	520	42.0
easiest	3	$0_{(+8,713)}$	0.03	2.5

Table 1. 6×6 square: stand vs conj (50 instances tested). $k_{(+m)}$ means that the search for solution needed k backtracks, to which we can add m backtracks performed in the internal MAC procedure.

Two interesting comments can be extracted from this table. The first one, not surprising, is that dealing globally with conjunctions of constraints (i.e., filtering more) pays off when problems become hard to solve. A look at the hardest problem illustrates that. On the 7×7 squares it is even more true since we could not solve them with the stand encoding.

The second comment is about backtracks in the internal search procedure. We see that their cost in cpu time is similar to the cost of classical ones (around 3000 backtracks/sec.). But, thanks to conjunctive constraints, they are circumscribed into these constraints, and seem to perform more useful partial searches since the total number of backtracks necessary in conj (internal + classical, i.e., 281+24,276 in average) is significantly smaller than the number of backtracks needed in stand (75,548 in average). We can notice that on the easiest problem, we still need 8,713 internal backtracks to enforce arc consistency in conjunctive constraints when only 3 backtracks were sufficient to find a solution in stand, showing that conj is not interesting on too easy problems.

8 Conclusion

Following the intuition that "the more globally you process filterings, the more able you are to tackle combinatorial explosion", we have proposed a filtering technique in order to deal globally with conjunctions of constraints, either because one of them embeds other constraints in the network, or because it is natural to define a constraint as a conjunction of others. The local consistency we achieve is stronger than arc consistency applied separately to each constraint of the conjunction. It could seem to be too costly to enforce such a local consistency. However, our first experiments show that as soon as a problem is difficult enough, the cost of globally filtering conjunctions of constraints is outweighed by the reduction of the search space it involves.

REFERENCES

[Bes94] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

[BFR95] C. Bessière, E. C. Freuder, and J. C. Régin. Using inference to reduce arc consistency computation. In *Proceedings IJCAI'95*, pages 592–598, Montréal, Canada, 1995.

[BR97] C. Bessière and J.C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings IJCAI'97*, pages 398–404, Nagoya, Japan, 1997.

[7] Notice that adding *alldiff* constraints on all the rows and columns in the stand encoding has no effect since they are included in the *alldiff* involving the r^2 variables and are treated independently of the C_S constraints. Notice too that what we name stand is not at all bad compared to the encoding with a complete graph of binary constraints stating that each pair of variables must take different values.

[8] The use of the "max degree" SVO is useless here since all the variables belong to the same number of constraints.

[Dec90] R. Dechter. On the expressiveness of networks with hidden variables. In *Proceedings AAAI'90*, pages 556–562, Boston MA, 1990.

[Fre91] E.C. Freuder. Completable representations of constraint satisfaction problems. In *Proceedings KR'91*, pages 186–195, Cambridge MA, 1991.

[ILO98] ILOG. *User's manual.* ILOG Solver, 4.3 edition, 1998.

[JJNV89] P. Janssen, P. Jégou, B. Nouguier, and M. C. Vilarem. A filtering process for general constraint-satisfaction problems: Achieving pairewise-consistency using an associated binary representation. In *Proceedings of the IEEE Workshop on Tools for Artificial Intelligence*, pages 420–427, Fairfax VA, 1989.

[Rég94] J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings AAAI'94*, pages 362–367, Seattle WA, 1994.

[Rég98] J.C. Régin. Modeling and solving sports league scheduling with constraint programming. In *Proceedings INFORMS*, Montreal, Canada, Spring 1998.

[RPD90] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proceedings ECAI'90*, pages 550–556, Stockholm, Sweden, 1990.

[SF94] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings PPCP'94*, Seattle WA, 1994.

[Van90] P. Van Hentenryck. Incremental constraint satisfaction in logic programming. In *Proceedings ICLP'90*, pages 189–202, Jerusalem, Israel, 1990.

[vBD95] P. van Beek and R. Dechter. On the minimality and global consistency of row-convex constraint networks. *Journal of the ACM*, 42(3):543–561, 1995.

Appendix

We give here a brief description of the `propagation` algorithm associated with the constraint of multidirectionality M_C of a constraint or a conjunction of constraints C. (The fact that one or several constraints are involved does not change the algorithm.)

We need first to define what we call the *breakpoint* of a value w.r.t. C, and for the "internal" domains on $X(C)$. What we call internal domains, are the domains during the search for support for a value (y, b) in the subnetwork $(X(C), \mathcal{D}_{X(C)}, \{y = b\} \cup Under(C))$. To avoid confusion, let us denote $D^{in}(x)$, the domain of x during the current search for support, as opposed to $D(x)$ the domain of x when that search for support began.

Definition 4 (breakpoint) *Let* $y \in X(C)$. $breakpoint_C(y, b) = k$ *iff* $\forall i \in [1..k-1]$, $highest(D^{in}(var(C, i))) = last_C(y, b)[i]$, *and* $highest(D^{in}(var(C, k))) \neq last_C(y, b)[k]$.

$breakpoint_C(y, b)$ can change in only one of the two following cases:

1. $last_C(y, b)$ is changed
2. the highest value of a domain is removed

The first case is easy to deal with. For the second, it is necessary to introduce a new data structure in order to determine when some breakpoints have changed. When a highest value (x, a) is removed from $D^{in}(x)$, the values (y, b) such that $last_C(y, b)$ contains (x, a) must be reachable. Therefore, we introduce the following list:

$L_C(x, a) = \{(y, b) \text{ s.t. } last_C(y, b) \text{ contains } (x, a)\}$

When the $last_C$ of an element is changed, the breakpoint value of this element is recomputed by calling the function `ComputeBreakpoint`.

Algorithm 5: function `ComputeBreakpoint`

```
ComputeBreakpoint (C: constraint; y: variable; b: value;
                   start: integer): integer
  k ← start ;
  while last_C(y,b)[k] = highest(D^in(var(C,k))) or k =
  index(C,y) do
  |  k ← k + 1 ;
  |_ if k > |X(C)| then  return k ;
  return k ;
```

We can first define a simple filtering algorithm that only deals with instantiated variables. Due to the definition of the breakpoint value, when a variable y is instantiated with a value b, all the variables that precede $breakpoint_C(y, b)$ w.r.t the constraint ordering can be instantiated to the maximal value of their domain. In order to avoid recomputing some parts of the search tree, the values less than $last_C(y, b)$ of the variable corresponding to the breakpoint can be deleted (see function `InstantiatedVar`).

Algorithm 6: function `InstantiatedVar`

```
InstantiatedVar (C: constraint; y: variable; b: value;
                 start: integer)
  k ← breakpoint_C(y,b) ;
  for i ∈ [start..k − 1] do
  |_ instantiate var(C,i) to highest(D^in(var(C,i))) ;
  if k ≤ |X(C)| then
  |  for a=first(D^in(var(C,k))) to last_C(y,b)[k] − 1 do
  |  |_ remove a from D^in(var(C,k)) ;
  |_
```

Now, we can present the `propagation` algorithm associated with M_C, which is called only when the highest value a of a domain $D^{in}(x)$ is deleted.

Algorithm 7: function `Propagation`

```
propagation (M_C: constraint; x: variable; a: value)
  for a' ∈ [(highest(D^in(x)) + 1)..a] ∩ D(x) do
  |  for (y,b) ∈ L_C(x,a') do
  |  |  if breakpoint_C(y,b) ≥ index(C,x) then
  |  |  |_ remove b from D^in(y) ;
  |_ |_
  for (y,b) ∈ L_C(x, highest(D^in(x))) do
  |  if breakpoint_C(y,b) = index(C,x) then
  |  |  k ← ComputeBreakpoint(y, b, index(C,x)) ;
  |  |  if k ≤ |X(C)| and highest(D^in(var(C,k))) <_d
  |  |  last_C(y,b)[k] then
  |  |  |  remove b from D^in(y) ;
  |  |  else
  |  |  |  breakpoint_C(y,b) ← k ;
  |  |  |  if y is instantiated then
  |  |  |  |_ InstantiatedVar (y, b, index(C,x)) ;
  |_ |_ |_
```