Boosting Open CSPs *

Santiago Macho González[1] Carlos Ansótegui[2]
Pedro Meseguer[1]

[1] Institut d'Investigació en Intel.ligència Artificial
Consejo Superior de Investigaciones Científicas
Campus UAB, E-08193 Bellaterra, Catalonia, Spain.
smacho@iiia.csic.es pedro@iiia.csic.es
[2] Universitat de Lleida (UdL)
Jaume II, 69, 25001 Lleida, Spain.
carlos@diei.udl.es

Abstract. In previous work, a new approach called Open CSP (OCSP) was defined as a way of integrate information gathering and problem solving. Instead of collecting all variable values before CSP resolution starts, OCSP asks for values dynamically as required by the solving process, starting from possibly empty domains. This strategy permits to handle unbounded domains keeping completeness. However, current OCSP algorithms show a poor performance. For instance, the FO-Search algorithm uses a Backtracking and needs to solve the new problem from scratch every time a new value is acquired. In this paper we improve the original algorithm for the OCSP model. Our contribution is two-fold: we incorporate local consistency and we avoid solving subproblems already explored in previous steps. Moreover, these two contributions guarantee the completeness of the algorithm and they do not increase the number of values needed for finding a solution. We provide experimental results than confirm a significant speed-up on the original approach.

1 Problem-solving in Open Environments

The increasing desire to automate problem-solving for scenarios that are distributed over a network of agents can be addressed with existing tools, first collecting all the options and constraints in an information gathering phase, and second solving the resulting problem using a centralized constraint solver. This conventional approach of collecting values from all servers and then running a CSP solver has been implemented in many distributed information systems [2, 1, 3]. However, it is very inefficient because it asks for more values than the strictly needed to find a solution, and it does not work with unbounded number of values.

In the real world, choices (values) and constraints are collected from different sources. With the increasing use of the Internet, those classical CSP problems

* Research partially supported by TIN2004-07933-C03-03, TIN2005-09312-C03-01 and TIC2003-00950 funded by the *Ministerio de Educación y Ciencia*.

could be defined in an *open-world* environment. Imagine you want to configure a PC using web data sources. Querying all the possible PC parts in all data sources on the web is just not feasible. We are interested in querying the minimum amount of information until finding a solution. Since the classical CSP approach (querying all values before search starts) is not applicable here, the new *Open CSP* approach [6] was proposed. Solving an *Open CSP* implies obtaining values for the variables, one by one. If the collected information does not allow to solve the problem, new values are requested. The process stops when a solution is found.

Although there are several models which in principle are suitable for open environments such as the *Iterative CSP* model [14] and the *Dynamic CSP* model [15], our model deals with a different problem. These approaches assume bounded domains while *Open CSP* assumes unbounded domains (domains with a possibly unlimited number of values). The ability to handle unbounded domains poses an interesting challenge when designing algorithms. For this reason, we do not include experiments comparing these approaches. See [5] for a detailed relation among these models.

Several algorithms for solving *Open CSPs* were proposed in [6]. These algorithms have a poor performance due to the lack of local consistency and that to the fact they solve from scratch a problem every time new values are acquired. Local consistency allows CSP algorithms to be very powerful, thus in this paper we present a new algorithm called FCO-Search that uses local consistency. We also show how the Factoring Out Failure strategy [4] can be used to avoid solving a problem from scratch.

This paper is organized as follows: firstly a brief description of the *Open CSP* model is given, followed by an explanation of the FO-Search algorithm. In the next sections we discuss how to improve this algorithm by incorporating local consistency and by avoiding to solve from scratch every instance of the OCSP. Then, we describe the FCO-Search algorithm which incorporates the mentioned improvements and finally the benefits of our approach are shown.

2 Open Constraint Satisfaction Problems

In Figure 1 we show the important elements of an open setting. The problem-solving process is modeled abstractly as the solution of a CSP. The choices that make up domains and permitted tuples of the CSP are distributed throughout an unbounded network of information servers $IS_1, IS_2, ...$, and accessed through a mediator [7]. For the purpose of this paper, we assume that this technology allows the CSP solver to obtain additional domain values:

– Using the $more(x_i, \ldots, (x_i, x_j), \ldots)$ message, it can request the mediator to gather more values for these variables. In this paper we assume that this method returns just one new value every time it is called.
– Using $options(x_i, \ldots,)$ and $options((x_i, x_j), \ldots)$ messages, the mediator informs the CSP solver of additional domain values or constraint tuples found in the network.

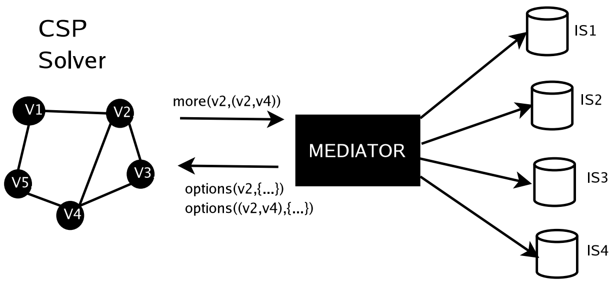


Fig. 1. *Elements of an open constraint satisfaction problem.*

– When there are no more values to be found, the mediator returns $nomore(x_i, \ldots)$.

Formally, an *Open CSP*(OCSP) [6] is a possibly infinite sequence $\langle$CSP(0), CSP(1), $\ldots\rangle$ of CSP instances. An instance CSP(i) is the tuple $\langle X, D(i), C(i)\rangle$ where,

– $X = \{x_1, x_2, ..., x_n\}$ is a set of n variables.
– $D(i) = \{D_1(i), D_2(i), ..., D_n(i)\}$ is the set of domains for CSP(i) where variable v_k takes values in $D_k(i)$. Initially domains are empty, $D_k(0) = \emptyset$, and they grow monotonically with i, $D_k(i) \subseteq D_k(i+1)$ for all k.
– $C(i) = \{c_1(i), c_2(i), \ldots, c_r(i)\}$ is a set of r constraints. A constraint $c(i)$ involves a sequence of variables $var(c(i)) = \langle v_p, \ldots, v_q\rangle$ denominated its scope. The extension of $c(i)$ is the relation $rel(c(i))$ defined on $var(c(i))$, formed by the permitted value tuples on the constraint scope. Initially, relations are empty, $rel(c_k(0)) = \emptyset$, and they grow monotonically, $rel(c_k(i)) \subseteq rel(c_k(i+1))$ for all k.

A *solution* is a set of value assignments involving all variables such that for some i, each value belongs to the corresponding domain in $D(i)$ and all value combinations are allowed by the constraints $C(i)$ of $CSP(i)$. Solving an OCSP requires an integration of search and information gathering. It starts from a state where all domains are empty, and the first action is to find values that fill the domains and allow the search to start. As long as the available information does not include enough values to make the CSP solvable, the problem solver initiates further information gathering requests to obtain additional values. The process stops as soon as a solution is found. Thus, with this OCSP model, we are solving a satisfiability problem, but also we are interested in optimizing the number of queries needed to find a solution.

The definition of an OCSP assumes that all constraints are binary. For experimentation, we assume that only variable domains change over time. We made these assumptions for the simplicity of the algorithms. They are not strong restrictions because using the **hidden variable encoding** method explained in [12, 13], any problem whose constraints are non binary or that are incrementally

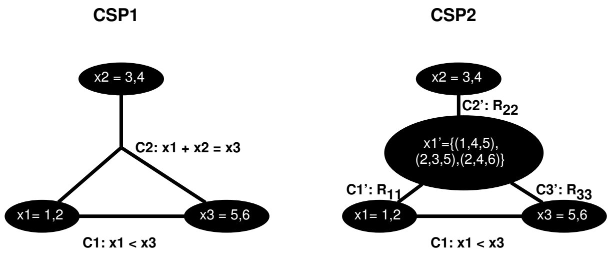


Fig. 2. *Hidden variable encoding of a non-binary CSP.*

discovered could be turned into a variable which has as values the tuples allowed by the constraints. These new variables are linked to variables involved in the original problem by new binary constraints that enforce equality between the variable values and the corresponding elements of the tuple. Figure 2 shows an example of the hidden variable encoding for a non binary CSP.

3 The FO-Search algorithm

The idea behind the FO-Search algorithm is that new values have to be gathered only when the current instance $CSP(i)$ has no solution. In that case, it usually contains a subproblem that already has no solution, and $CSP(i)$ could be made solvable only by creating a solution to that subproblem. Information gathering thus should focus on the variables of this subproblem.

An *Unsolvable Subproblem* of size *k* is a set of variables $S = \{x_{s1}, x_{s2}, \ldots, x_{sk}\}$ such that there is no value assignment $x_{s1} \in D_{s1}, \ldots, x_{sk} \in D_{sk}$ (where D_{si} represents a domain) that satisfies all constraints between these variables. If any subset $S' \subset S$ is solvable, we call this set of variables *Minimal Unsolvable Subproblem.*

Note that any strategy that does not assure the selection of a variable that belongs to a *Minimal Unsolvable Subproblem* may lead us to an incomplete algorithm. Actually, this is the key point of working with unbounded domains. Think for example about an strategy of selecting the most constrained variable. This variable is not forced to belong to a *Minimal Unsolvable Subproblem*, thus, adding new values to this variable may loop infinitely without solving the unsolvable subproblems that made inconsistent the instance. Although it seems difficult to choose a correct variable, the following result provides a method to identify a variable that belongs to a *Minimal Unsolvable Subproblem.*

Proposition 1. Let a CSP be explored by a failed backtrack search algorithm (BT) with static variable ordering $(x_1, ..., x_n)$, and let x_k be the deepest node reached in the search with inconsistency detected at x_k. Then x_k, called the *failed variable*, is part of every unsolvable subproblem of the CSP involving variables in the set $\{x_1..x_k\}$.

Using this result (proved in [6]), a failed CSP search allows us to identify the failed variable, for which an additional value should be collected. When there are no additional values for this variable, the mediator returns a `nomore` message, and other variables are then considered.

The resulting algorithm **FO-search** (failure-driven open search) is shown in Algorithm 1. It makes the assumption that variables are ordered by the index

Algorithm 1 *The* **FO-search** *algorithm.*

```
 1: procedure FO-search(X(i),D(i),C(i))
 2: $i \leftarrow 1, k \leftarrow 1$
 3: repeat {backtrack search}
 4:   if $exhausted(d_i)$ then {backtrack}
 5:     $reset - values(d_i)$, $i \leftarrow i - 1$
 6:   else
 7:     $k \leftarrow max(k, i), x_i \leftarrow nextvalue(d_i)$
 8:     if $consistent(\{x_1..x_i\})$ then {extend assignment}
 9:       $i \leftarrow i + 1$
10:     end if
11:     if $i > n$ then
12:       return $\{x_1, ..., x_n\}$ as a solution
13:     end if
14:   end if
15: until $i = 0$
16: if $e_k = CLOSED$ then
17:   if $(\forall i \in 1..k-1) e_k = CLOSED$ then
18:     return failure
19:   end if
20: else
21:   $nv \leftarrow$ more$(x_k)$
22:   if $nv =$ nomore$(x_k)$ then
23:     $e_k \leftarrow CLOSED$
24:   end if
25:   $d_k \leftarrow nv \cup d_k$
26: end if
27: reorder variables so that $x_k$ becomes $x_1$ (relative order of others remains the same)
28: FO-search(X(i),D(i),C(i)) {search again}
```

i, and uses the array $E = \{e_1, .., e_n\}$ to indicate whether the domain for the corresponding variable is completely known (CLOSED). The algorithm assumes that no constraint propagation is used, although the chronological backtracking can be replaced with backjumping techniques (jumping directly to the last constraint violation) to make it more efficient.

In [6] it is shown that if the current instance $CSP(i)$ contains a minimal unsolvable subproblem, the FO-Search algorithm (algorithm 1) is complete even in the presence of unbounded domains. The main drawbacks of the algorithm are: (i) it does not use local consistency (ii) every instance $CSP(i)$ is solved from scratch. In next sections we will study how the FO-Search algorithm could be improved with these suitable properties.

4 Local consistency for OCSPs

Given an unsolvable CSP instance and a static ordering of its variables $o = x_1, \ldots, x_n$, the *failed variable* along the ordering o is the deepest variable in the search tree developed by BT that detects inconsistency, following the ordering

o . Different variable orderings may identify different failed variables, all being equally acceptable.

As shown in previous section, the unbounded domains of the OCSP model could lead us to incomplete algorithms if we do not choose the correct variable. The interest for finding a failed variable x_k of an unsolvable problem in the OCSP context is clear: x_k belongs to a minimal unsolvable subproblem, for which more values are required in order to make it solvable (otherwise the whole problem will continue being unsolvable). Variable x_k is the natural candidate to get more values, to extend this minimal unsolvable subproblem.

Local consistency has been shown to be essential in constraint satisfaction to increase the solving performance. We think that local consistency should plays a similar role in OCSP. With this aim, we explore how the popular Forward Checking (FC) algorithm [11] can be used in the OCSP context.

In order to identify a failed variable when having local consistency in OCSPs, we provide the next proposition.

Proposition 2. Let an unsolvable CSP be explored by a FC algorithm with static variable ordering $o = (x_1, ..., x_n)$, and let x_k be the deepest variable in the search tree for which an empty domain was detected. Then there exists an ordering for which x_k is the failed variable in the BT algorithm.
Proof. Let us build the search tree that BT will traverse following the static ordering o. We know that BT will not visit any branch below x_k level (otherwise, BT will find as consistent an assignment that FC detected as inconsistent). BT may fail before x_k level at some branches, but it will not go below that level in any branch. It may happen:

1. There is at least one branch for which BT reaches x_k level.
2. In all branches, BT fails before reaching x_k level.

If 1, x_k is the failed variable for o. If 2, however, x_k is not the failed variable for o since BT never reaches it in any branch. Assuming that x_j is the deepest inconsistent variable found by BT along the ordering o, we construct the ordering o' that is equal to o but where x_j and x_k exchange places. Along this new ordering o', BT finds x_k as the failed variable. To see this, it is enough to realize that FC never instantiates more than $x_1, \dots, x_{j-1}$ variables (otherwise FC would have instantiated an inconsistent assignment, something impossible [10]) and instantiating these variables is enough to detect inconsistency on x_k. BT following ordering o' will find x_k as the failed variable. Therefore, there is an ordering for which x_k is the failed variable. $\Box$

It is important to point that this failed variable could be different from the failed variable obtained in Proposition 1. Therefore, we cannot assure that this failed variable is part of every unsolvable subproblem of the CSP involving variables in the set $\{x_1..x_k\}$.

Using proposition 2, we are able to identify a failed variable even when having local consistency. This property will be useful to create a complete algorithm which uses local consistency for solving OCSP.

5 Using Factoring Out Failure in OCSPs

Beside local consistency, we can improve the FO-Search algorithm avoiding to re-explore the search space already explored. To achieve this, two solutions could be performed:

– Reuse the no-goods found in the earlier search. This no-good recording method is explained in [8, 9]. It seems that this is the most obvious solution, but is not practical because new acquired values may invalidate the no-goods inferred in previous searches.
– Using the technique of decomposing a CSP into subproblems proposed by Freuder and Hubbe [4] called *Factoring Out Failure*.

The decomposition process and the algorithm called *Factoring Out Failure* are described in [4]. The algorithm extracts unsolvable subproblems from a CSP, thus limiting search effort to smaller and possible solvable subproblems. This idea seems to apply well to OCSP: we can decompose the new instance $CSP(i)$ obtained after collecting new values, into the old problem just searched $CSP(i-1)$ (which is known to be unsolvable) and a new one based on the values just obtained. This extraction method is shown in Algorithm 2. We have to be careful here, because limiting search to the new found values may cause incompleteness of the algorithm for solving the OCSP.

Algorithm 2 The Extract procedure

```
procedure Extract ($CSP(i-1), CSP(i), decomposition$)
begin
repeat
    Pick a variable, $x_k \in CSP(i)$ whose domain does not match in $CSP(i-1)$.
    Divide $CSP(i)$ into two subproblems $CSP_1$ and $CSP_2$ that differ only in that the domain of
    $x_k$ matches the first subproblem $CSP_1$ while the remaining values of $x_k$ matches the second
    subproblem $CSP_2$.
    $CSP(i) \leftarrow CSP_1; decomposition \leftarrow decomposition \cup CSP_2$
    Apply Extract to the updated problem $CSP(i)$ and $decomposition$ with the same subproblem
    $CSP(i-1)$.
until $CSP(i) = CSP(i-1)$
return $decomposition$;
end Extract
```

To obtain a complete algorithm, we study the relation between the failed variables of two consecutive instances $CSP(i-1)$ and $CSP(i)$.

Let be $CSP(i) = CSP(i-1) \cup CSP_{nv}$ where:

$$CSP_{nv} =< X, D'_{nv}, C(i) >= \begin{cases} D'_{nv}(x_k) = D(x_k) \cup nv & x_k \text{ failed variable in } CSP(i-1) \\ D'_{nv}(x_i) = D(x_i) & \forall x_i \neq x_k \end{cases} \quad (1)$$

Let say we have solved instance $CSP(i-1)$ with variable order $x_0, \ldots, x_k, \ldots, x_n$. Assume that it was found unsolvable with x_k as failed variable. Thus, following the FCO-Search algorithm, instance $CSP(i)$ will be solved with variable order

$x_k, x_0, \ldots, x_j, \ldots, x_{k-1}, \ldots, x_n$. Let say that problem CSP_{nv} was solved using the same variable order $x_k, x_0, \ldots, x_j, \ldots, x_{k-1}, \ldots, x_n$, and it was found unsolvable with x_j as failed variable. Comparing both failed variables, x_k and x_j, we can identify one of these situations:

- The depth level of variable x_j in CSP_{nv} is greater than or equal to the depth level of variable x_k in $CSP(i-1)$. Note that solving instance $CSP(i)$ with variable order $x_k, x_0, \ldots, x_j, \ldots, x_{k-1}, \ldots, x_n$ has as possible candidates for failed variable the set $\{x_0, \ldots, x_{k-1}\}$ where x_{k-1} is the deepest variable. Also note that the depth level of variable x_{k-1} in the variable order $x_k, x_0, \ldots, x_j, \ldots, x_{k-1}, \ldots, x_n$ is the same as x_k in the variable order $x_0, \ldots, x_k, \ldots, x_n$ used for solving instance $CSP(i-1)$. Thus, we can conclude that x_j is the failed variable of instance $CSP(i)$, because it is deeper that the deepest possible candidate x_{k-1} which has the same depth level that x_k in instance $CSP(i-1)$. Note that in this situation, we just need to solve CSP_{nv} in order to know the failed variable of instance $CSP(i)$ or found a possible solution.
- The depth level of variable x_j in CSP_{nv} is lower than the depth level of variable x_k in instance $CSP(i-1)$. As before, solving instance $CSP(i)$ has as possible candidates for failed variable the set $\{x_0, \ldots, x_{k-1}\}$ where x_{k-1} is the deepest variable. In this case, the set of failed variables is $\{x_0, \ldots, x_{k-1}\}$ $\cup$ $\{x_j\}$. Therefore, we have to solve $CSP(i)$ from scratch with the new variable ordering $x_k, x_0, \ldots, x_j, \ldots, x_{k-1}, \ldots, x_n$ to know which is the failed variable.

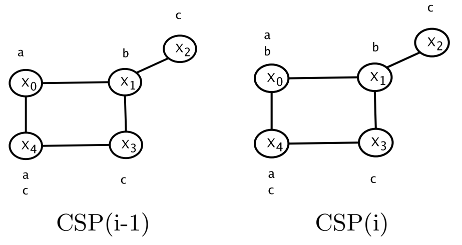


Fig. 3. An Open graph coloring example.

Figures 3 and 4 show an example of solving instance $CSP(i)$ using the *Factoring Out Failure* algorithm. Figure 3 (left) shows an unsolvable instance $CSP(i-1)$ of a graph coloring OCSP with variable ordering $\{x_1, x_2, x_3, x_4, x_0\}$. In the example, x_0 is the failed variable, thus following the FO-Search algorithm we collect a new value $x_0 = b$ obtaining the instance $CSP(i)$ as shown in Figure 3(right) with a new variable ordering $\{x_0, x_1, x_2, x_3, x_4\}$. The FO-Search algorithm will solve again from scratch instance $CSP(i)$ although we know that instance $CSP(i-1)$ is an unsolvable subset of $CSP(i)$. Before solve $CSP(i)$, we

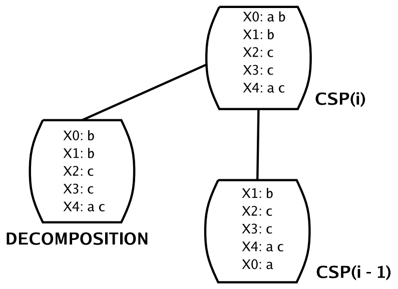


Fig. 4. An example of instance $CSP(i)$ decomposition .

can use the Algorithm 2 to extract the known unsolvable subproblem $CSP(i-1)$ and then focusing on solving the decomposition subproblem obtained [1] (for more details of the *Factoring Out Failure* method, please refer to [4]). When we extract the subproblem $CSP(i-1)$ from problem $CSP(i)$ we obtain a new subproblem (called *DECOMPOSITION* in figure 4) which is smaller and probably easy to solve than instance $CSP(i)$.

6 The FCO-Search algorithm

Based on the previous FO-Search algorithm, we developed a new algorithm with the same properties (new values have to be gathered only when the current instance is unsolvable) but including local consistency and using the Factoring Out Failure decomposition. We call the obtained algorithm FCO-Search.

The FCSearch function, implements the classical FC algorithm. In line 7, it assigns a new value to variable x_i and starts the propagation (lines 8-14). Lines 15-17 check if there is a variable x_j which domain was exhausted by the previous assignment of variable x_i, recording the deepest variable with empty domain. It returns either a solution or a failed variable if instance $CSP(i)$ is unsolvable.

The initial call of the FCO-Search (Algorithm 3) is $FCO{-}Search(CSP(0), x_0)$ where $CSP(0)$ is the initial instance and x_0 is the first variable in the variable order given by this initial instance. The FCO-Search algorithm has two parameters: $< X(i), D(i), C(i) >$ as instance $CSP(i)$ and x'_k as the failed variable of instance $CSP(i-1)$. In line 2 we use the *Extract* procedure to extract the unsolvable instance $CSP(i-1)$ from instance $CSP(i)$. From this decomposition, we obtain the subproblem $< X_d, D_d, C_d >$ which is solved by the FCSearch function in line 3. If there is no solution, then it compares in line 7 if the failed variable x_k of problem $CSP(i)$ is deeper than the previous failed variable x'_k of instance $CSP(i-1)$. When this condition is true, we know that x_k is the failed variable of instance $CSP(i)$. Otherwise we need to solve $CSP(i-1)$ to calculate the new failed variable (line 8). Once the new failed variable is calculated, we call the function **more** in line 15 and a new value is added. Line 21 reorders variables

[1] Problem $CSP(i)$ differs from problem $CSP(i-1)$ only in the new queried value.

```
1: function FCSearch(X,D,C)
2: $i \leftarrow 1, k \leftarrow 1$
3: repeat {main loop}
4:   if $exhausted(d_i)$ then
5:     $i \leftarrow i - 1$
6:   else
7:     $x_i \leftarrow nextvalue(d_i)$
8:     for all $x_j \in (x_{i+1}, \ldots, x_n)$ do
9:       for all $a \in d_j$ do
10:        if $\neg consistent(x_1, \ldots, x_i, x_{i+1}, \ldots, x_n)$ then
11:          $d_j \leftarrow d_j - a$
12:        end if
13:      end for
14:    end for
15:    if $\{\exists x_j \in (x_{i+1}, \ldots, x_n) \mid exhausted(d_j)\}$ then {backtrack}
16:      reset each $x_j \in (x_{i+1}, \ldots, x_n)$ to value before $x_i$ was set
17:      $k \leftarrow max(k, j), i \leftarrow i - 1$
18:    else
19:      $i \leftarrow i + 1$
20:    end if
21:    if $i > n$ then
22:      return $\{x_1, ..., x_n\}$ as a solution
23:    end if
24:  end if
25: until $i = 0$
26: return $x_k$ as failed variable
27: end FCSearch
```

so that x_k becomes the first variable and the relative order of other variables remains the same. Finally line 22 calls recursively algorithm FCO-Search with instance $CSP(i)$ and with the new failed variable.

Proposition 3. Algorithm FCO-Search is complete.

Proof. We know that the FO-Search algorithm is complete [6], where search is performed by BT. We will show that the FCO-Search algorithm, where BT is replaced by FC, is also complete. Let x_k be the failed variable found by FC, and x_j the failed variable found by BT, both along the ordering o. Either (i) $x_j = x_k$ or (ii) x_j appears before x_k in o. In (i) the FCO-Search algorithm behaves like FO-search, so it is obviously complete. Then, let us assume (ii). In this case, there is an ordering o' equal to o but with x_j and x_k exchanging places. Along o' BT would have found x_k as failed variable. We show that FCO-Search with ordering o behaves like FO-Search with ordering o'. First, both algorithms ask for one more value for x_k and put it as the first variable, forming the ordering $x_k, x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_j, \ldots, x_n$. We know that the subset $x_k, x_1, \ldots, x_{j-1}$ formed a unsatisfiable subproblem in the previous iteration, but we do not know if the new value of x_k has made it solvable. If it is solvable, then the algorithm will continue looking for the next unsatisfiable subproblem (if any). If not, FCO-Search will find a new failed variable x_p in the sequence $x_k, x_1, \ldots, x_{j-1}$ using the FC algorithm. Obviously, x_p appears before or it is equal to x_{j-1}. Since BT found a consistent instantiation from x_1 until x_{j-1}, in this subset BT cannot find a constraint that will stop search before reaching x_p. So BT would find the same failed variable as FC. Therefore, FCO-Search will behave exactly as FO-search, until finding a solution for that unsatisfiable subproblem.

Algorithm 3 *The* **FCO-Search** *algorithm.*

```
procedure FCO-Search($< X(i), D(i), C(i) >, x'_k$)
$< X_d, D_d, C_d >= Extract(< X(i), D(i), C(i) >, < X(i-1), D(i-1), C(i-1) >, \emptyset)$
if $solution(FCSearch(X_d, D_d, C_d))$ then
  return $\{x_1, ..., x_n\}$ as a solution
end if
$x_k$ = returned $x_k$ by the FCSearch function
if $x'_k > x_k$ then
  $x_k = FCSearch(X(i-1), D(i-1), C(i-1))$
end if
if $e_k = CLOSED$ then
  if $(\forall i \in 1..k-1) e_k = CLOSED$ then
    return failure
  end if
else
  $nv \leftarrow$ more($x_k$)
  if $nv =$ nomore($x_k$) then
    $e_k \leftarrow CLOSED$
  end if
  $d_k \leftarrow nv \cup d_k$
end if
reorder variables so that $x_k$ becomes $x_1$ (relative order of others remains the same)
FCO-Search($X(i), D(i), C(i), x_k$)
end FCO-Search
```

In both cases (i) and (ii), FCO-search behaves like FO-search with ordering (i) o, or (ii) o'. Since FO-Search is complete, FCO-Search is complete. □

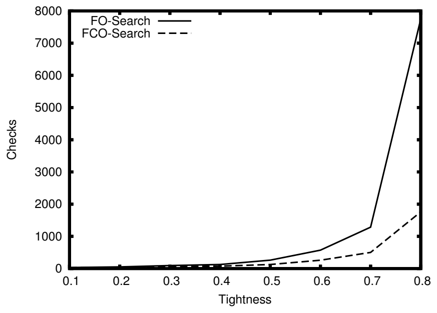
7 Experiments

We compared the performance of the new FCO-Search algorithm against the FO-Search algorithm [2] on solvable random OCSPs. We performed several experiments.

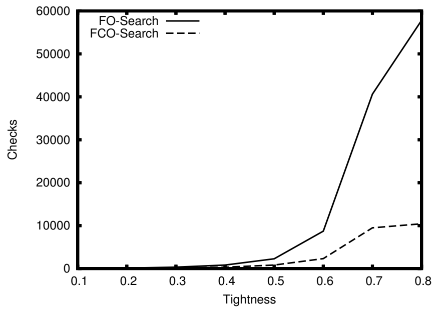
As a first experiment, we compared the performance of the algorithms until a solution is found when density and tightness change. At this point we want to emphasize that our experimental results are done with finite domain random classes (following the B random model) in order to allow researches to reproduce the experiments. For this experiment we generated 1000 random OCSPs with 7 variables and with a domain size of 10 values. Figures 5 (a)(b) compare the number of checks needed to find a solution for the OCSP when (a) density = 0.2 (b) density = 0.8 and tightness moves from 0.1 to 0.8. In Figure 5 (c) density = 0.8 but tightness moves from 0.1 to 0.6 due to the difficulty of generating solvable problems when tightness > 0.6 .

Figure 5 shows an important improvement of the FCO-Search algorithm over the FO-Search in any case. Combining local consistency and avoid solving from scratch the same problem reduces dramatically the number of constraint checks for hard problems producing a substantial improvement in the performance of the proposed algorithm.

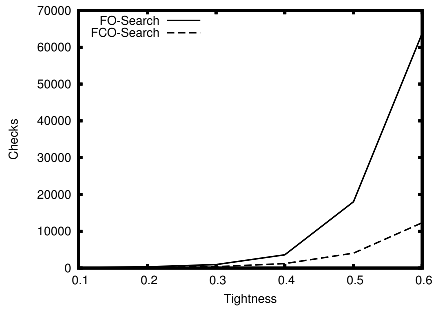
[2] The FCO-Search algorithm is compared against the backtracking version of the FO-Search without backjumping.



(*a*)



(*b*)



(*c*)

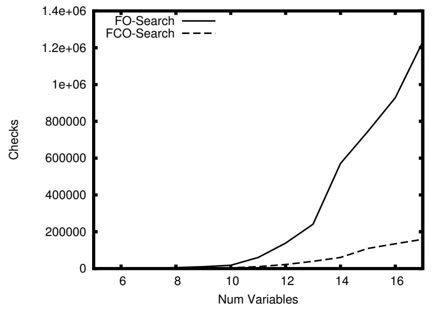
Fig. 5. Comparison of the number of checks when tightness increases. Top: (a) when density = 0.2, (b) when density = 0.5. Bottom: (c) when density = 0.8

In the second experiment, we compared the algorithms in two aspects: the number of accesses to information sources and the number of constraint checks until a solution for the OCSP is found [3]. We generated 100000 random OCSPs with between 5 to 17 variables and a domain size of 10 values, forcing the graph to be solvable and at least connected and at most complete with random density and tightness.

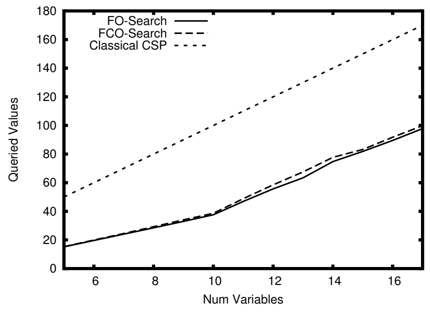
Figure 6(a) shows the number of checks against the number of variables, studying the performance of the algorithms when increasing the number of variables. The benefits are very important (it is "likely" to get at least an order of magnitude for bigger instances) because the FCO-Search algorithm incorporates local consistency and avoids redoing the same solving process every time a new value is added (as explained in previous section).

Figure 6(a) compares the number of queries against the number of variables. In this figure, we include the *Classical CSP* approach which decouples information gathering and problem solving. This approach first queries all values for all variables and then solves the problem. We decided to include it because it establishes an upper bound for the performance of the algorithms. As mentioned

[3] These constraint checks are not hidden in the FCO-Search algorithm.



(a)



(b)

Fig. 6. (a) Comparison of the number of checks vs the number of variables when domain size = 10 (b) Comparison of the number of values queried vs the number of variables when domain size = 10.

before, it could be possible that both algorithms FO-Search and FCO-Search do not select the same failed variable in every instance $CSP(i)$, so the number of queried values could vary. Empirical results show that the number of queries is nearly the same for both algorithms, having a slightly better performance the FO-Search algorithm. As explained before, this is due to the property that the FO-Search algorithm finds a failed variable x_k which participates in all unsolvable subproblems $\{x_1, \ldots, x_k\}$, while the FCO-Search algorithm just finds a component that may does not have this property. Thus, the selected variable by the FO-Search algorithm gets a new value that could solve several unsolvable subproblems at the same time, while the selected variable by the FCO-Search can not assure this property. Despite this difference, empirical results show that both algorithms have a similar performance.

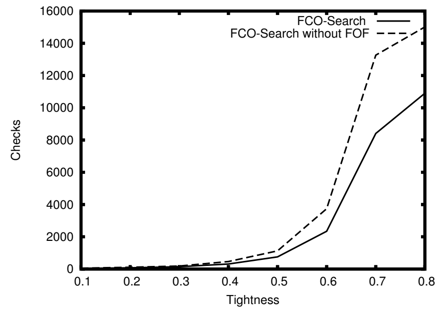


Fig. 7. Comparison of the number of checks when the FCO-Search algorithm uses Factoring out Failure and without it. (density = 0.5)

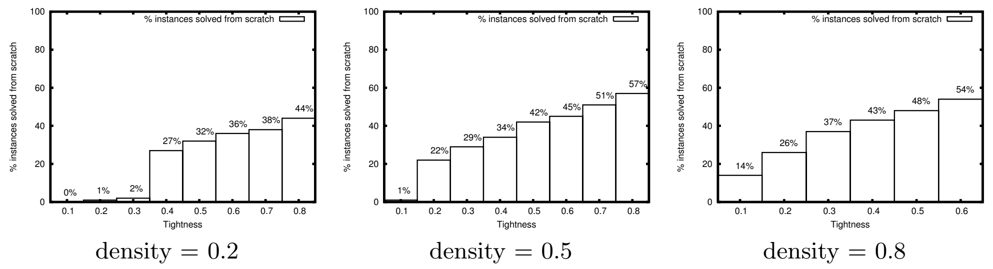


Fig. 8. Number of times instances are solved from scratch when density and tightness increases.

The benefits of our new approach come from a combination of applying local consistency and avoid solving instances from scratch. We are interested in the influence of these improvements separately. Figure 7 compares the number of checks when the FCO-Search algorithm uses the Factoring out Failure method and when it does not use it. For this particular experiment, we generated 1000 random OCSPs with 7 variables with domain size 10 and with a *density* = 0.5. In figure 8, we studied the percentage of instances solved from scratch for every random OCSP. For these instances, solving the obtained decomposition subproblem is not enough for finding the next failed variable. As expected, figure 8 shows that when problems become harder, around 60% of instances are solved from scratch. For these hard problems, if we compare Figure 7 and Figure 8, we can see that local consistency bring us more benefits, although the contribution of the Factoring out Failure method is also significant.

8 Conclusions

The performance of algorithms for solving OCSPs is very poor because they neither use local consistency nor avoid solving subproblems already explored in previous step. We have studied how we can incorporate local consistency while keeping the completeness of the algorithm by finding a failed variable. We also have shown how we can use the idea of the Factoring out Failure method proposed by Freuder [4] to avoid redoing the previous work. Based on these two techniques, we have developed a new algorithm called *FCO-Search* for solving OCSPs. The results described in last section show a significant speed-up in the number of checks compared with the previous *FO-Search* algorithm while the number of queried values remains nearly the same, even when the problem becomes hard to solve.

As future work, we are studying how to incorporate dynamic variable ordering into our algorithm. This new improvement poses a new challenge when calculating the failed variable, but we suspect it will provide a promising improvement for OCSPs solvers.

References

1. Marian Nodine and Jerry Fowler and Tomasz Ksiezyk and Brad Perry and Malcolm Taylor and Amy Unruh : Active Information Gathering in InfoSleuth. International Journal of Cooperative Information Systems **9** (2000) 3-28
2. Genesereth M. and R. Keller and A. M. Duschka: Infomaster: An Information Integration System. Proceedings of 1997 ACM SIGMOD Conference, May 1997
3. Alon Y. Levy and Anand Rajaraman and Joann J. Ordille: Querying Heterogeneous Information Sources Using Source Descriptions. Proceedings of the Twenty-second International Conference on Very Large Databases, VLDB Endowment, Saratoga, Calif, Bombay, India (1996) 251–262
4. Eugene C. Freuder and Paul D. Hubbe: Extracting Constraint Satisfaction Subproblems. IJCAI - 1995, 548-557.
5. S. Macho-Gonzalez and Pedro Meseguer. Open, Interactive and Dynamic CSP. Changes'05 International Workshop on Constraint Solving under Change and Uncertainty (CP'05).
6. Boi Faltings and Santiago Macho-Gonzalez: Open Constraint Programming. Artificial Intelligence 161, (2005) 181–208.
7. Gio Wiederhold and Michael R. Genesereth: The Conceptual Basis for Mediation Services. IEEE Expert volume 12 (5) 38–47 (1997).
8. Yuejun Jiang and Thomas Richards and Barry Richards: No-good backmarking with min-conflicts repair in constraint satisfaction and optimization. Proceedings of Principles and Practice of Constraint Programming 94, (1994) 21–39.
9. Thomas Schiex and Gérard Verfaillie: Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. International Journal on Artificial Intelligence Tools, volume 3 (2) 187–207 (1994).
10. Grzegorz Kondrak and Peter van Beek: A theoretical evaluation of selected backtracking algorithms. Artificial Intelligence 89, (1997) 365–387.
11. Haralick and Elliot: Increasing tree search efficiency for constraint-satisfaction problems. Artificial Intelligence 14 (3) 263-313, (1980).
12. F. Rossi and C. Petrie and V. Dhar: In the equivalence of constraint satisfaction problems. Proc. ECAI-90 550–556 (1990).
13. Kostas Stergiou and Toby Walsh : Encodings of Non-binary Constraint Satisfaction Problems. Proceedings of AAAI/IAAI 163-168 (1999).
14. Evelina Lamma and Paola Mello and Michela Milano and Rita Cucchiara and Marco Gavanelli and Massimo Piccardi: Constraint Propagation and Value Acquisition: Why we should do it Interactively. Proceeding of IJCAI'99 468-477 1999.
15. Thomas Schiex and Gérard Verfaillie: Nogood Recording for Static and Dynamic Constraint Satisfaction Problem, International Journal of Artificial Intelligence Tools (3)2 187–207, (1994).