Nogood-FC for Solving Partitionable Constraint Satisfaction Problems*

Montserrat Abril, Miguel A. Salido, Federico Barber

Dpt. of Information Systems and Computation, Technical University of Valencia
Camino de Vera s/n, 46022, Valencia, Spain
{mabril, msalido, fbarber}@dsic.upv.es

Abstract. Many real problems can be naturally modelled as constraint satisfaction problems (CSPs). However, some of these problems are of a distributed nature, which requires problems of this kind to be modelled as distributed constraint satisfaction problems (DCSPs). In this work, we present a distributed model for solving CSPs. Our technique carries out a partition over the constraint network using a graph partitioning software; after partitioning, each sub-CSP is arranged into a *DFS-tree CSP structure* that is used as a hierarchy of communication by our distributed algorithm. We show that our distributed algorithm outperforms well-known centralized algorithms solving partitionable CSPs.

keywords: Distributed Constraint Satisfaction Problems, Graph Partition, Distributed Algorithms.

1 Introduction

One of the research areas in computer science that has gained increasing interest in recent years is constraint satisfaction, mainly because many problems can be modelled as constraint satisfaction problems (CSPs) and solved using specific constraint satisfaction techniques. These include problems from fields such as artificial intelligence, operational research, information systems, databases, etc. Most of these problems can be naturally modelled as CSPs. However, some of these problems are of a distributed nature either for reasons of security, for privacy requirements, or for physical distribution of objects, variables or constraints. Therefore, problems of this kind must be modelled as distributed constraint satisfaction problems (DCSPs), where the set of variables and constraints are distributed among a set of *agents*. These agents, as distributed solvers, are in charge of solving their own sub-problem and must coordinate themselves with the rest of the *agents* in order to reach a solution to the global problem [5].

* This work has been partially supported by the research projects TIN2004-06354-C02- 01 (Min. de Educacion y Ciencia, Spain-FEDER), FOM- 70022/T05 (Min. de Fomento, Spain), GV/2007/274 (Generalidad Valenciana) and by the Future and Emerging Technologies Unit of EC (IST priority - 6th FP), under contract no. FP6-021235-2 (project ARRIVAL).

On the other hand, constraint satisfaction problems can also model large real problems, which generally imply models with a great number of variables and constraints. These problems often have clear, smaller sub-problems. These problems can be handled as a whole only at overwhelming computational cost. This leads to problem partition so that they can subsequently be solved more effectively by means of algorithms that solve DCSPs. In this work, we divide a problem using graph partitioning techniques. Many researchers are currently working on graph partitioning ([1], [9]). The main objective of graph partitioning is to divide the graph into a set of regions so that each region has roughly the same number of nodes and so that the sum of all the edges connecting different regions is minimized. Graph partitioning can also be applied to constraint satisfaction problems. Thus, we can use graph partitioning when dealing with large-scale CSPs to distribute the problem into a set of sub-CSPs. For instance, we can divide a CSP into several sub-CSPs so that constraints among variables of each sub-CSP are minimized.

In the literature of constraint satisfaction, the need to handle DCSP emerged at the beginning of the 90s. However, most researchers who work in this field focus their attention on algorithms in which each *agent* handles a single variable [11]. Even though these algorithms can be transformed so that each *agent* handles multiple variables [7], none of the resulting algorithms scales up well as the size of the problems increases, due to space requirements and/or computacional cost. Therefore, the resolution of real problems with algorithms of this type, in practice, is not viable.

In this paper, we present new algorithms for the resolution of distributed constraint satisfaction problems that are able to handle multiple variables by *agent*. These algorithms handle the data obtained by means of communication among the *agents* in order to obtain greater efficiency during the resolution process. In addition, their space requirements are minimum, which is why they are able to handle large, local sub-problems.

In the following section, we summarize some definitions about CSPs. A method for translating a CSP into a *DFS-tree CSP structure* is presented in section 3. Our distributed algorithm for solving *DFS-tree CSP structures* (depth first search tree CSP structures) is presented in section 4. In section 5, we propose a new algorithm to carry out *intra-agent* search. An evaluation of our methods over random problems is presented in section 6. Finally, we summarizes our conclusions in section 7.

2 Constraint Satisfaction Problems

In this section, we present some basic definitions related to CSPs, which will be convenient for our purposes and will unify works from the constraint satisfaction community. Then, we present three ways for solving a CSP: as a centralized problem, as a partitionable problem and as a distributed problem.

*A **CSP*** consists of a set of variables $X = \{x_1, ..., x_n\}$; each variable $x_i \in X$ has a set D_i of possible values (its domain); and a finite collection of constraints $C = \{c_1, ..., c_p\}$ that restricts the variable values.

A **solution** to a CSP is an assignment of values to all the variables so that all constraints are satisfied.

*A **binary constraint network*** is a network in which every constraint involves at most two variables. In this case, the network can be associated with a constraint graph, where each node represents a variable and the arcs connect nodes whose variables are explicitly constrained [2]. In this paper, we assume a binary constraint network.

*A **DFS-tree CSP structure*** is a tree whose nodes are composed by subproblems, where each subproblem is a CSP (sub-CSPs) (see Figure 2). Each node of the *DFS-tree CSP structure* is a ***DFS-node*** and each individual and atomic node of each sub-CSP is a ***single-node***. Each *single-node* represents a variable, and each *DFS-node* is made up of one or several *single-nodes.* Each constraint between two *single-nodes* of different *DFS-nodes* is called ***inter-constraint***. Each constraint between two *single-nodes* of the same *DFS-node* is called ***intra-constraint***.

Partition : A partition of a set C is a set of disjoint subsets of C whose union is C. The subsets are called the blocks of the partition.

Distributed CSP: A distributed CSP (DCSP) is a CSP in which the variables and constraints are distributed among automated *agents* [11]. Each *agent* attempts to determine the values of its variables satisfying its *intra-constraints.* Furthermore, there are *inter-constraints* and *agents* must be coordinated among them because their value assignment must also satisfy *inter-constraints.*

2.1 Partitioning of CSPs

There are many ways to solve a CSP. However, these problems can be classified into three categories: centralized problems, distributed problems, and partitionable problems.

- A CSP is a *centralized CSP* when there are no privacy/security rules between parts of the problem, and all knowledge about the problem can be gathered into one process. It is commonly recognized that centralized CSPs must be solved by centralized CSP solvers. Many problems are represented as typical examples to be modelled as a centralized CSP and solved using constraint programming techniques. Some typical examples are: sudoku, n-queens, map-coloring.
- A CSP is a *distributed CSP* when variables, domains and constraints of the underlying network are distributed among *agents.* This distribution is mainly carried out due to security and/or privacy factors: constraints may be strategic information that should not be revealed to competitors; a failure

of one *agent* can be less critical and other *agents* might be able to find a solution without the failed *agent*. Examples of such systems are sensor networks, meeting scheduling, web-based applications, etc.
- A CSP is a *partitionable CSP* when the global problem can be divided into smaller problems (sub-problems) which must be coordinated to find the solution to the global problem. In this case, the set of variables and/or constraints can have a distributed spacial structure. Therefore, the search space of a CSP can be divided into several regions, and a solution is found by using parallel computing (see Figure 1).

Given these three categories, we can conclude that a distributed CSP cannot be solved by using centralized techniques. However, a centralized CSP can be solved by using distributed techniques if the CSP is decomposed previously.

Real problems usually imply models with a great number of variables and constraints, causing dense networks. Thus, it could be an advantage to divide problems of this kind into several simpler interconnected sub-problems which can be more easily solved.

In the following example, we show that a centralized CSP could be partitioned into several sub-problems in order to obtain simpler sub-CSPs. This way, we can apply a distributed technique to solve the partitioned CSP.

The map coloring problem is a typically centralized problem. The goal of a map coloring problem is to color a map so that regions sharing a common border have different colors. Let's suppose that each country of Europe must be colored. Figure 1 (1) shows a colored portion of Europe. This problem can be solved by a centralized CSP solver. However, if the problem is to color each region of each country of Europe (Spain, Figure 1(3); France, Figure 1(4)), it is easy to realize that the problem can be partitioned into a set of sub-problems, grouped by clusters. This problem can be solved as a distributed problem, even when the problem is not inherently distributed.

A map coloring problem can be rephrased in a non-directed graph. Here, every region of the map is replaced by a vertex of the graph, and two vertices are connected by an edge if and only if the two regions share a border segment. In our problem of coloring the regions of each country of Europe, it can be observed that the corresponding graph maintains clusters representing each country (Spain, Figure 1(3); France, Figure 1(4)). Thus, the problem can be solved in a distributed way.

3 How to Decompose a binary CSP into a DFS-Tree CSP structure

Given any binary CSP, it can be translated into a *DFS-tree CSP structure*. However, there exist many ways to decompose a graph into a *DFS-tree CSP structure*. Depending on the user requirements, it may be desirable to obtain balanced *DFS-nodes*, that is, each *DFS-node* maintains roughly the same number of *single-nodes*; or it may be desirable to obtains *DFS-nodes* in such a way that the number of edges connecting two *DFS-nodes* is minimized.

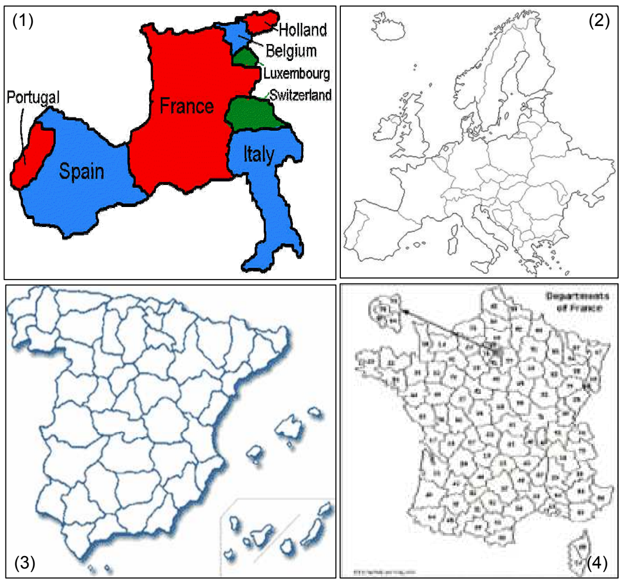

Fig. 1. Map coloring of Europe.

We present a proposal that is focused on decomposing the problem by using graph partitioning techniques. Specifically, the problem decomposition is carried out by means of a graph partitioning software called METIS [6]. METIS provides two programs, *pmetis* and *kmetis*, for partitioning an unstructured graph into k roughly equal partitions, such that the number of edges connecting nodes in different partitions is minimized.

The first step to obtain a *DFS-tree CSP structure* is to divide the problem. We use METIS to decompose a CSP into several sub-CSPs so that *inter-constraints* among variables of each sub-CSP are minimized. Each *DFS-node* will be composed by a sub-CSP.

```
Algorithm DFSStructure(G,v)
Input: Graph G, originally all nodes are unvisited. Start DFS-node v of G
Output: DFS-Tree CSP structure
process(v); /* put DFS-node v into DFS-tree CSP structure      */
mark v as visited;
forall DFS-node i adjacent^1 to v not visited do
    DFSStructure(G,i);
end
/* (1) DFS-node i is adjacent to DFS-node v if at least one
   inter-constraint exists between i and v.                    */
```
Algorithm 1: DFSStructure Algorithm.

The next step is to build the *DFS-tree CSP structure* with r *DFS-nodes* in order to be studied by *agents*. The number of agents (r) is obtained by using the formulae given in [10]. The *DFS-tree CSP structure* is used as a hierarchy to communicate messages between *DFS-nodes*. The *DFS-tree CSP structure* is built by using Algorithm 1. The nodes and edges of graph G are, respectively, the *DFS-nodes* and *inter-constraints* obtained after the CSP decomposition. The root *DFS-node* is obtained by selecting the tightest *DFS-node*, in the sense that this sub-CSP maintains a higher number of *single-nodes*. The DFSStructure algorithm then simply puts *DFS-node* v into the *DFS-tree CSP structure* (*process(v)*), initializes a set of markers so we can tell which vertices are visited, chooses a new *DFS-node* i, and recursively calls DFSStructure(i). If a *DFS-node* has several adjacent *DFS-nodes*, it would be equally correct to choose them in any order, but it is very important to delay the test to determine whether a *DFS-node* is visited until the recursive calls for previous *DFS-nodes* are finished.

Figure 2 shows an example of CSP generated by a random generator module $generate_R(C, n, k, p, q)$[1], where C is the constraint network; n is the number of variables in network; k is the number of values in each of the domains; p is the probability of a non-trivial edge; q is the probability of an allowable pair in a constraint. This figure represents the constraint network $< C, 20, 50, 0.1, 0.1 >$. We can observe that this problem can be divided into several cluster (Figure 2-left) and it can be converted into a *DFS-tree CSP structure* (Figure 2-right).

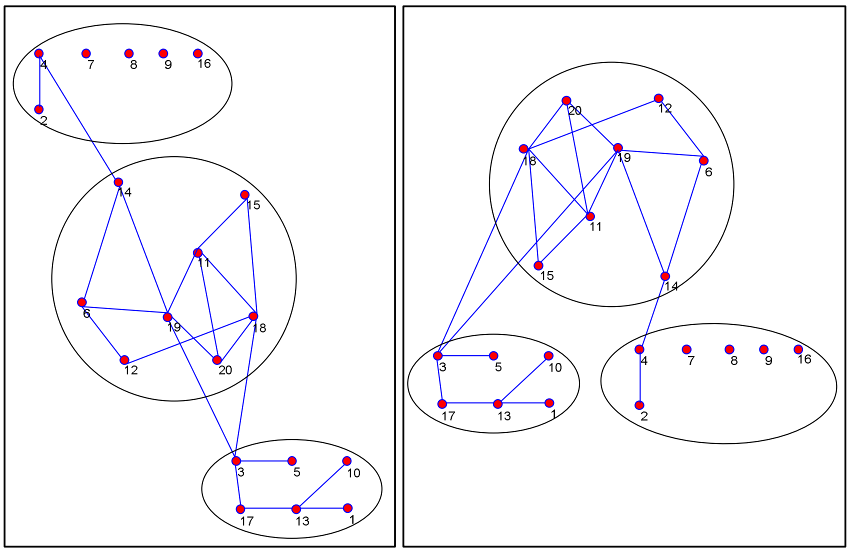

Fig. 2. Left: Decomposed Problem. Right: DFS-Tree CSP structure.

[1] A library of routines for experimenting with different techniques for solving binary CSPs is available at http://ai.uwaterloo.ca/∼vanbeek/software/software.html

4 The DFS-Tree Search Algorithm: DTS

In section 3, we presented a method for structuring a binary CSP into a *DFS-Tree CSP structure*. In this section, we show the DFS-Tree Search Algorithm (DTS) for solving *DFS-Tree CSP structures*.

The DTS algorithm can be considered as a distributed and asynchronous technique. In the specialized literature, there are many works about distributed CSPs. In [11], Yokoo et al. present a formalization and algorithms for solving distributed CSPs. These algorithms can be classified as either centralized methods, synchronous or asynchronous backtracking [11].

The DTS algorithm is committed to solving the *DFS-tree CSP structure* in a *Depth-First Search Tree* (*DFS Tree*) where the root *DFS-node* is composed by the tightest sub-CSP. In the literature, *DFS trees* have already been investigated as a means to boost search [3]. Due to the relative independence of nodes lying in different branches of the *DFS tree*, it is possible to perform search in parallel on these independent branches.

Once the variables are divided and arranged into a *DFS-tree CSP structure*, the problem can be considered as a distributed CSP, where a group of *agents* manages each sub-CSP with its variables (*single-nodes*) and its constraints (edges). Each *agent* is in charge of solving its own sub-CSP by means of a search. Each subproblem is composed by its CSP subject to the variable assignment generated by the ancestor *agents* in the *DFS-tree CSP structure*. In this way, *DFS-tree CSP structure* induces a tree-based structure of solver *agents*.

Thus, the root *agent* works on its subproblem (root *DFS-node*). If the root *agent* finds a solution, then it sends the consistent partial state to its children *agents* in the *DFS-tree CSP structure*, and all children work concurrently to solve their specific subproblems knowing consistent partial states assigned by the root *agent*. When a child *agent* finds a consistent partial state, it again sends this partial state to its children and so on. Finally, leaf *agents* try to find a solution to its own subproblems. If each leaf *agent* finds a consistent partial state, it sends an OK message to its parent *agent*. When all leaf *agents* answer with OK messages to their parents, a solution to the entire problem is found. When a child *agent* does not find a solution, it sends a Nogood message to the parent *agent*. The Nogood message contains the variables that empty the variable domains of the child *agent*. When the parent *agent* receives a Nogood message, it stops the search of the children and it tries to find a new solution taking into account the Nogood *information* and so on. Depending on the management of this information, the search space is differently pruned. If a parent *agent* finds a new solution, it will start the same process again sending this new solution to its children. Each *agent* works in the same way with its children in the *DFS-tree CSP structure*. However, if the root *agent* does not find a solution, the DTS algorithm returns *no solution found*.

At the top of Figure 3, we show our technique for translating a CSP into a *DFS-tree CSP structure*. Figure 3 also shows an example of DTS algorithm execution. The root *agent* (a_1) starts the search process finding a partial solution. Then, it sends this partial solution to its children. The brother *agents*

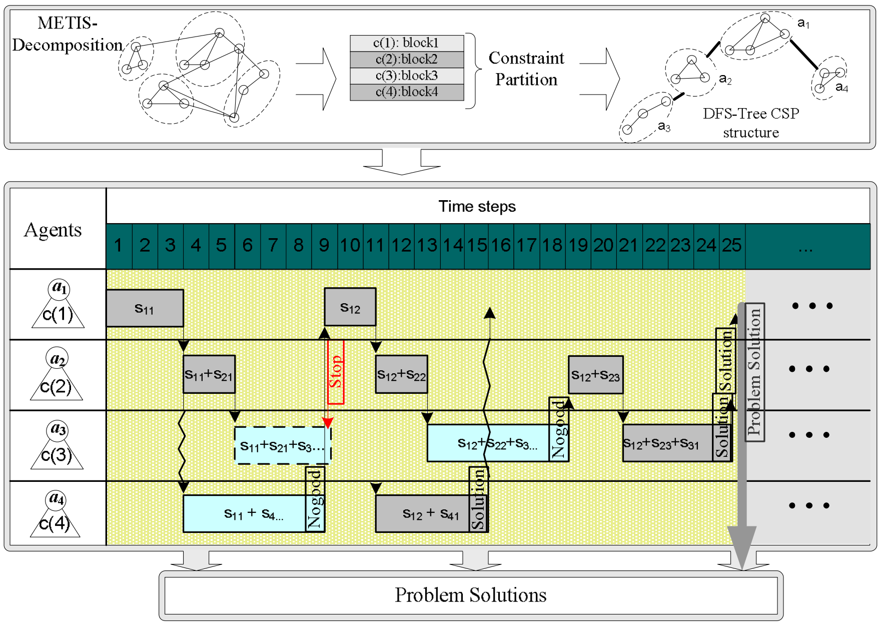

Fig. 3. Partitioning Technique and DTS Algorithm Execution.

are committed to concurrently finding the partial solutions of their subproblem. Each *agent* sends the partial problem solutions to its children *agents*. A problem solution is found when all leaf *agents* find their partial solutions. For example, (state $s_{12} + s_{41}$) + (state $s_{12} + s_{23} + s_{31}$) is a problem solution. The concurrence can be seen in *Time step* 4, in which *agents* a_2 and a_4 are concurrently working. *Agent* a_4 sends a Nogood message to its parent (*agent* a_1) in step 9 because it does not find a partial solution. Then, *agent* a_1 stops the search process of all its children, and it finds a new partial solution which is again sent to its children. Now, *agent* a_4 finds its partial solution, and *agent* a_2 works with its child, *agent* a_3, to find their partial problem solution. When *agent* a_3 finds its partial solution, a solution to the global problem will be found. This happens in *Time step* 25.

5 *Intra-agent* search of *partial_solutions*

In section 4, we show the distributed algorithm DTS for solving CSPs and we point out that each *agent* is in charge of solving its own sub-CSP by means of search techniques. *Agents* are free to select their own search methods in order to find *partial_solutions*.

In this section, we propose a new algorithm to search *partial_solutions*: *Nogood-FC* algorithm. The core of *Nogood-FC* is the well-known Forward Checking (FC) algorithm [3]. Furthermore, our algorithm is based on *Nogood_message*, which allows us to prune the search space.

```
Algorithm Nogood-FC($G$, Nogood_message)
Input: Constraint Graph $G$={$X, E$} where E is a set of constraints and
       $X$={$x_1, ..., x_n$} is a set of variables ordered according to
       $d$=($x_1, ..., x_n$); Nogood message: set of inconsistent variables.
Output: partial_solution

  if Nogood_message==∅ then
      solution ← FC($x_1$); /* search $1^{st}$ partial_solution starting
          Forward Checking algorithm from variable $x_1$.          */
  else
      solution ← current solution;
      if $\exists$ $x_j \in X$: $x_j \in$ Nogood_message and
      $\nexists$ $x_t \in X$:($x_t \in$ Nogood_message $\wedge$ $t > j$) then

          while (solution $\neq$ NO_SOLUTION) and
          ($\nexists$ $x_i \in$ Nogood_message: $x_i$ changed value($x_i$)) do

              if $\exists$ $x_s \in X$: $x_s \in$ Nogood_message and
              $\nexists$ $x_t \in X$:($x_t \in$ Nogood_message $\wedge$ $s < t < j$) then

                  set value($x_j$) restricted by $x_s$;
              end
              foreach $x_i \in X$ / $j \leq i < n$ do
                  reset effects of assignment $x_i \leftarrow value(x_i)$;
              end
          end

          solution ← FC($x_j$); /* search next partial_solution
              starting FC algorithm from variable $x_j$.           */
      end
  end
  return solution
```

Algorithm 2: *Nogood-FC* Algorithm.

First, *Nogood-FC* uses Nogood *information* to prune the search space jumping to the variable (x_j) involved in the *Nogood_message* within the lowest level in the search tree. Furthermore, if there is another variable (x_s) involved in the *Nogood_message* within the second lowest level in the search tree, then *Nogood-FC* will cancel the current value of x_j until x_s changes its current value.

Algorithm 2 shows the pseudo-code of *Nogood-FC.* This algorithm has the same behavior of FC when it searches the first solution. However, the search of the next solutions is different because it first prunes some solutions that are inconsistent with the *Nogood_message*, and later it searches for a new solution using FC. Therefore, the search space is pruned and *partial_solutions* can be deleted. However Nogood-FC does not prune any valid global solution of the whole problem. However, it does eliminate *partial_solutions* that cannot be included in a *global_solution* since they are inconsistent with the variables of other/s sub-CSP/s.

Next, we show an example of *Nogood-FC* execution. It is based on the DCSP shown in Figure 4. This figure shows a CSP divided into two sub-problems.

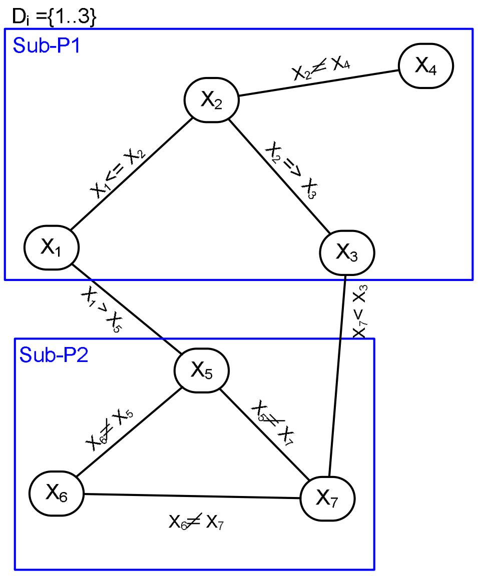

Fig. 4. Example of DCSP.

According to the DTS and *Nogood-FC* algorithms, the process starts with the search for the first *partial_solution* in sub-problem 1. In this case (first solution), the *Nogood-FC* algorithm works in the same way as the FC algorithm. It finds the first *partial_solution* included in the set of valid *partial_solutions* which are shown in Figure 5: (X_1=1, X_2=1, X_3=1, X_4=2). This first *partial_solution* is inconsistent with sub-problem 2 since assignment X_1=1 empties the domain of X_5. Therefore, sub-problem 2 sends the *Nogood* message: ($X_1 = 1$).

When the agent that owns sub-problem 1 gets the *Nogood* message ($X_1 = 1$), it eliminates value 1 of D_1, and it searches for a new *partial_solution*. The *Nogood-FC* algorithm detects the pruning of all solutions with X_1=1 (see Figure 6). Therefore, it undoes the effects of X_3, X_2 and X_1 assignments, and it executes FC(X_1) to find a new *partial_solution*: (X_1=2, X_2=2, X_3=1, X_4=1). Again, this *partial_solution* is inconsistent with sub-problem 2 since assignment X_3=1 empties the domain of X_7. Therefore, sub-problem 2 sends the *Nogood* message: ($X_3 = 1$).

The *Nogood* message ($X_3 = 1$) causes the pruning of all solutions with X_3=1 (see Figure 7). Now, the *Nogood-FC* algorithm undoes the effects of X_3 assign-

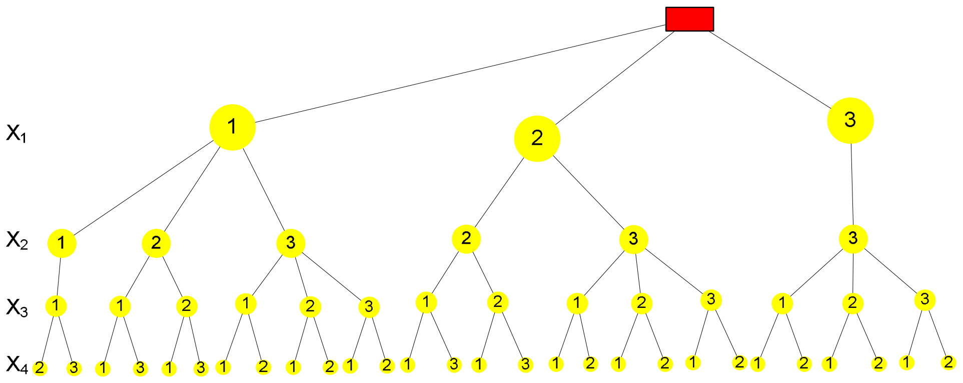

Fig. 5. Set of valid *partial_solutions* of Sub-P1 (Figure 4).

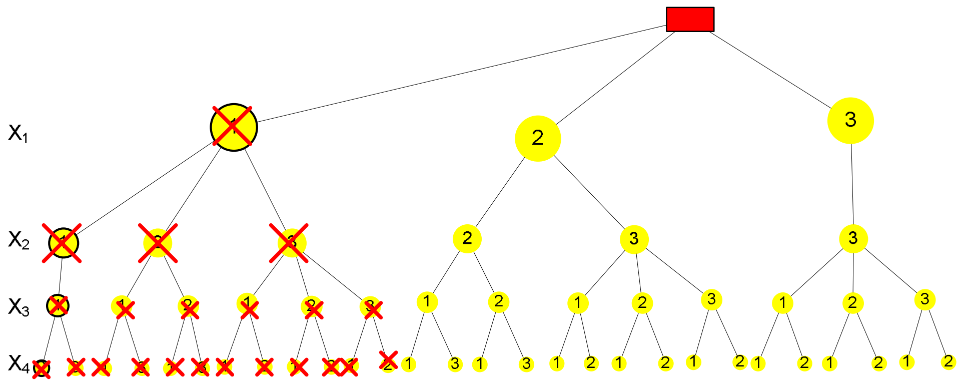

Fig. 6. *Nogood-FC* pruning with *Nogood*-message (X_1=1).

ment, and it executes FC(X_3) to find a new *partial_solution*: (X_1=2, X_2=2, X_3=2, X_4=1). This new *partial_solution* is again inconsistent since assignments X_1=2 and $X_3 = 2$ do not allow the algorithm to find any consistent values in domains D_5 and D_7 that satisfy constraint $X_5 \neq X_7$. Therefore, the agent that owns sub-problem 2 sends the *Nogood* message: (X_1=2,$X_3 = 2$).

The information of the Nogood message (X_1=2,$X_3 = 2$) causes the *Nogood-FC* algorithm to set X_3 restricted by X_1. Thus, the value of X_3 will not be 2 until X_1 changes its current value. Therefore, the *partial_solutions* shown in Figure 8 are pruned. Then, the *Nogood-FC* algorithm undoes the effects of the X_3 assignment, and it executes FC(X_3) to find a new *partial_solution*: (X_1=2, X_2=3, X_3=3, X_4=1). Finally, this *partial_solution* is consistent with sub-problem 2.

Figure 8 shows that thanks to the *Nogood* message, the *Nogood-FC* algorithm has pruned 17 *partial_solutions* of sub-problem 1 which are inconsistent with sub-problem 2.

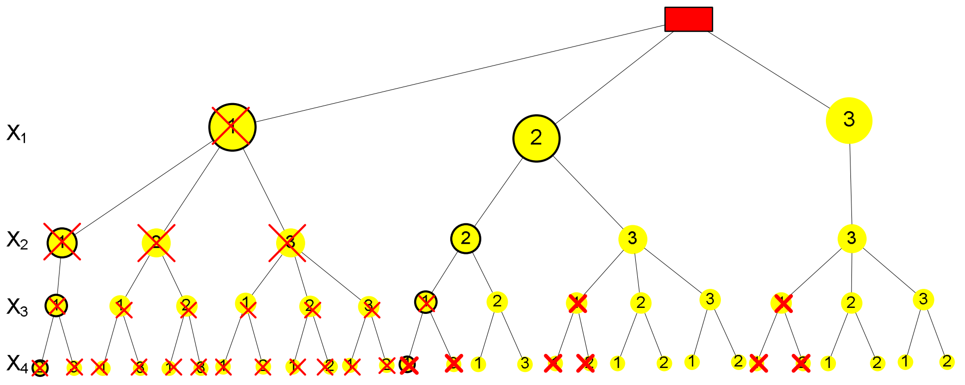

Fig. 7. *Nogood-FC* pruning with *Nogood*-message (X_3=1).

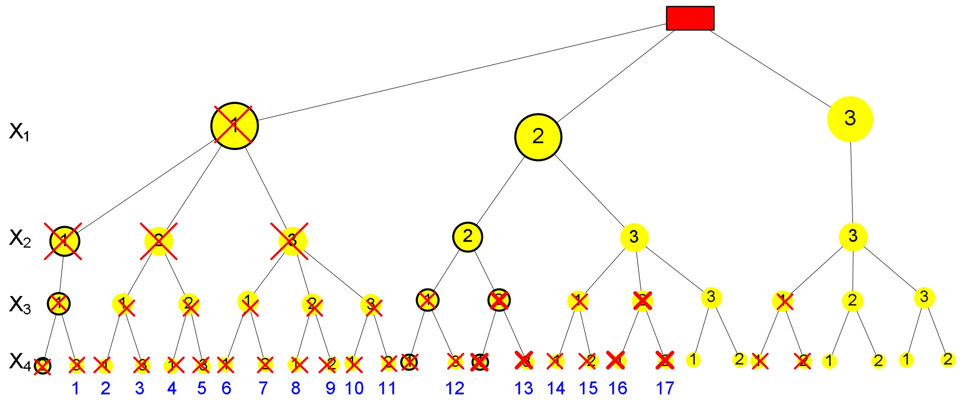

Fig. 8. *Nogood-FC* pruning with *Nogood*-message (X_1=2,X_3=2).

6 Evaluation

In this section, we carry out an evaluation between the DTS algorithm versus a complete CSP solver. To this end, we have used a well-known centralized CSP solver called Forward Checking (FC)[2].

Experiments were conducted on random distributed networks of binary constraints defined by the 8-tuple $< a, n, k, c, p1, p2, q1, q2 >$, where a was the number of sub-CSPs, n was the number of variables on each sub-CSP, k was the values in each domain, c was the probability of connection between two sub-CSPs, $p1$ was the *inter-constraint* density for each connection between two sub-CSPs (probability of a non-trivial edge between sub-CSPs), $p2$ was the *intra-constraint* density on each sub-CSP (probability of a non-trivial edge on each sub-CSP), $q1$ was the tightness of *inter-constraints* (probability of a forbidden value pair

[2] FC was obtained from: http://ai.uwaterloo.ca/ vanbeek/software/software.html

in an *inter-constraint*) and $q2$ was the tightness of *intra-constraints* (probability of a forbidden value pair in an *intra-constraint*). These parameters are currently used in experimental evaluations of binary Distributed CSP algorithms [4]. The problems were randomly generated by using the generator library in [4] and by modifying these parameters. Each problem instance execution had a limited CPU-time (TIME_OUT) of 3600 seconds.

In this evaluation, we compare the running times and *concurrent constraint checks* (CCC) ([8]) of the DTS and FC algorithms. The *intra-agent* search in each sub-CSP of the distributed model is carried out with the Nogood-FC algorithm. After evaluating the parameters of the 8-tuple $< a, n, k, c, p1, p2, q1, q2 >$, we detected that the parameter c (probability of connection between two sub-CSPs) implies the major behavior differences between DTS and FC. Obviously, random problems with low values of c represent constraint networks with clear clusters, and random problems with high values of c represent dense networks.

The following graphs show the influence of parameter c on the DTS and FC behaviors. For each 8-tuple $< a, n, k, c, p1, p2, q1, q2 >$, we tested these algorithms with 4 problem instances of random CSPs according to the following values: the number of variables on each sub-CSP (n) was increased from 5 to 30; the number of sub-CSPs (a) and the domain size (k) were fixed to 5; $c = \{10, 50, 90\}$; $p1 = \{1, 5, 9\}$; $p2 = \{10, 50, 90\}$; $q1 = \{10, 50, 90\}$; $q2 = \{10, 50, 90\}$. Each result shown in the graphs is the average of $4^4 = 256$ instances of random CSPs with different combinations of $p1$, $p2$, $q1$ and $q2$.

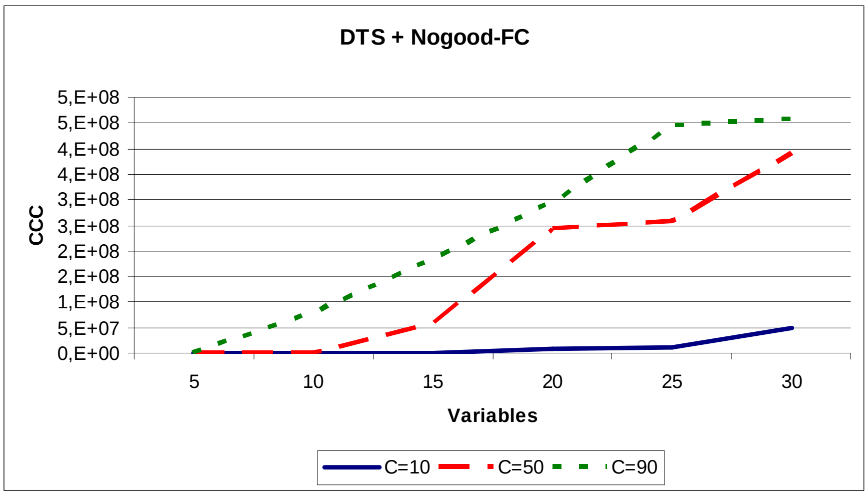

Fig. 9. The influence of parameter C on Concurrent Constraint Checks executing the DTS algorithm and the Nogood-FC *intra-agent* search with a different number of variables.

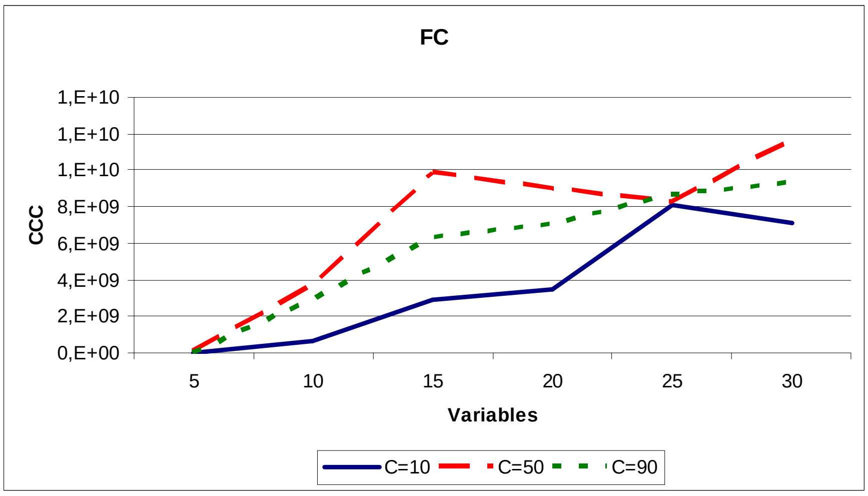

Fig. 10. The influence of parameter C on Concurrent Constraint Checks executing the FC algorithm with a different number of variables.

In Figures 9 and 10, we show the number of *concurrent constraint checks* used for DTS and FC to solve problems with different values of parameter c: 10%, 50% and 90%, and different numbers of variables. It can be observed that DTS *concurrent constraint checks* are always lower than FC *concurrent constraint checks*. However, DTS running times are not always better than FC running times. This is due to the cost of message exchange carried out by the DTS algorithm. Figures 11 and 12 show the running times used for DTS and FC to solve the same problems. In constraint networks with clusters ($c = 10$), DTS running times are lower than FC running times, but FC has lower running times than DTS when problems are very connected ($c = 90$). The running times of both algorithms are very similar when $c = 50$.

In Figures 13 and 14, we show TIME_OUT executions of the DTS and FC algorithms, respectively. As the number of variables increases, the number of problem instance executions that achieve TIME_OUT increases. Furthermore, as in the running time case, DTS has fewer TIME_OUT executions than FC when $c = 10$, and FC has fewer TIME_OUT executions than DTS when $c = 90$.

7 Conclusions

We have proposed a distributed algorithm (DTS) for solving *partitionable* CSPs that can manage sub-CSPs with several variables. First, we translate the original CSP into a *DFS-tree CSP structure*, where each node is a sub-CSP. Later, DTS solves the resultant *DFS-tree CSP structure*. Furthermore, we present a new algorithm (Nogood-FC) to carry out *intra-agent* search. Nogood-FC exploits the Nogood *information* for pruning the search space. The evaluation shows a

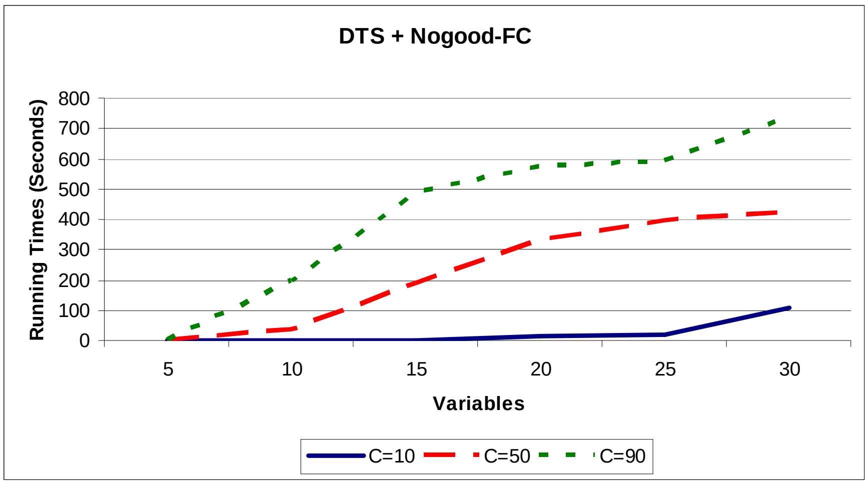

Fig. 11. The influence of parameter C on Running Times executing the DTS algorithm and the Nogood-FC *intra-agent* search with a different number of variables.

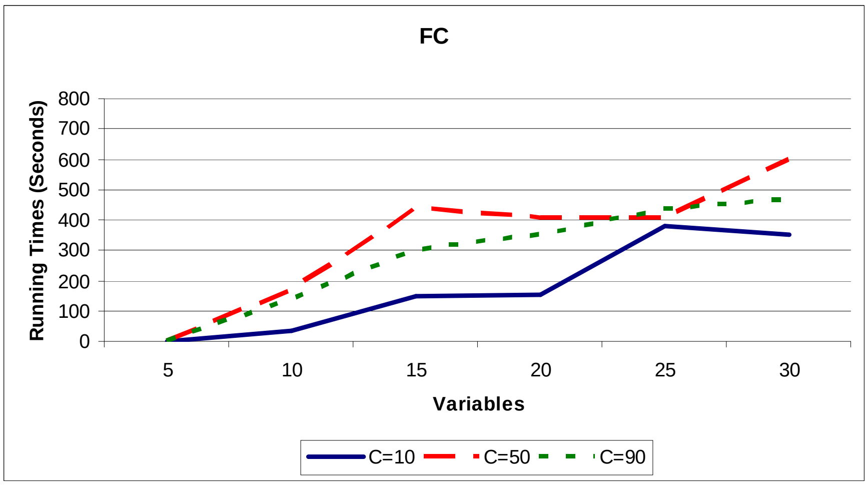

Fig. 12. The influence of parameter C on Running Times executing the FC algorithm with a different number of variables.

good behavior in partitionable CSPs. Thus, this technique is suitable for solving centralized problems that can be divided into smaller subproblems in order to improve the search for solution.

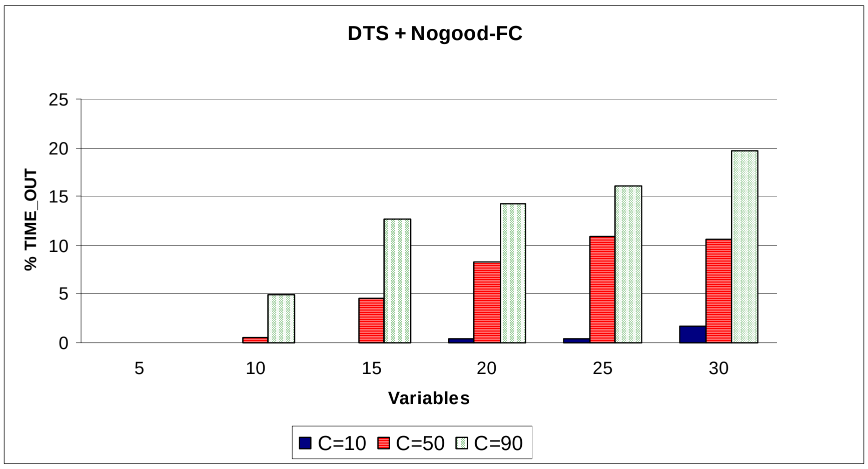

Fig. 13. The influence of parameter C on % of TIME_OUTs executing the DTS algorithm and the Nogood-FC *intra-agent* search with a different number of variables.

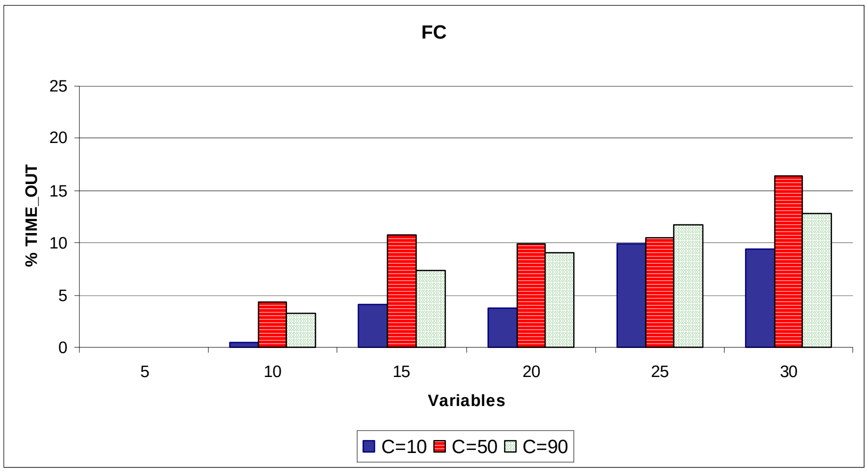

Fig. 14. The influence of parameter C on % of TIME_OUTs executing the FC algorithm with a different number of variables.

References

1. A. Abou-Rjeili and G. Karypis, 'Multilevel algorithms for partitioning power-law graphs', pp. 10 pp.+, (2006).
2. R. Dechter, 'Constraint networks (survey)', *Encyclopedia Artificial Intelligence*, 276–285, (1992).
3. R. Dechter, *Constraint Processing*, Morgan Kaufman, 2003.
4. R. Ezzahir, C. Bessiere, M. Belaissaoui, and El-H. Bouyakhf, 'DisChoco: A platform for distributed constraint programming', *In Proceedings of IJCAI-2007 Eighth*

International Workshop on Distributed Constraint Reasoning (DCR'07), 16–27, (2007).

5. B. Faltings, 'Distributed constraint programming', 699–729, (2006).
6. G. Karypis and V. Kumar, 'Using METIS and parMETIS', (1995).
7. Yokoo M. and Hirayama K., 'Distributed constraint satisfaction algorithm for complex local problems', *Proceedings of the 3rd International Conference on Multi Agent Systems*, (1998).
8. A. Meisels, E. Kaplansky, I. Razgon, and R. Zivan, 'Comparing performance of distributed constraint processing algorithms', *In Proc. 4th Workshop on Distributed Constraint Reasoning*, (2002).
9. F. Pellegrini, 'A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries', *Springer-Verlag, published in the LNCS series*, **4641**, 191–200, (2007).
10. M.A. Salido and F. Barber, 'Distributed csps by graph partitioning', *Applied Mathematics and Computation*, **183**, 491–498, (2006).
11. M. Yokoo and K. Hirayama, 'Algorithms for distributed constraint satisfaction: A review', *Autonomous Agents and Multi-Agent Systems*, **3**, 185–207, (2000).