DFS-Tree Based Heuristic Search*

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. In constraint satisfaction, local search is an incomplete method for finding a solution to a problem. Solving a general constraint satisfaction problem (CSP) is known to be NP-complete; so that heuristic techniques are usually used. The main contribution of this work is twofold: (i) a technique for de-composing a CSP into a DFS-tree CSP structure; (ii) an heuristic search technique for solving DFS-tree CSP structures. This heuristic search technique has been empirically evaluated with random CSPs. The evaluation results show that the behavior of our heuristic outperforms than the behavior of a centralized algorithm.

keywords: Constraint Satisfaction Problems, CSP Decomposition, heuristic search, DFS-tree.

1 Introduction

One of the research areas in computer science that has gained increasing interest during last years is constraint satisfaction, mainly because many problems like planning, reasoning, diagnosis, decision support, scheduling, etc., can be formulated as constraint satisfaction problems (CSPs) and can be solved by using constraint programming techniques in an efficient way.

Many techniques to solve CSPs have been developed; some originate from solving other types of problems and some are specifically for solving CSPs. Basic CSP solving techniques include: search algorithms, problem reduction, and heuristic strategies.

The more basic sound and complete search technique is Chronological Backtracking. It systematically traverses the entire search space in a depth-first manner. It instantiates one variable at a time until it either finds a solution or proves no solutions exist. However, it can be inefficient because of thrashing. To improve efficiency during search, some basic stochastic algorithms have been developed. Random guessing algorithm is the most naive stochastic search. Like blindly

* 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).

throwing darts, it repeatedly 'guesses' a complete assignment and checks if the assignment satisfies the constraints until it finds a solution or reaches timeout (or some maximum number of iterations). Since the algorithm assigns variables in a non-systematic way, it neither avoids checking for the same assignment repeatedly, nor guarantees to verify all possible assignments. Because it does not guarantee to check all possible assignments, the algorithm is incomplete and so it cannot guarantee a solution or prove no solution exists.

However, searching for solutions can be very time consuming, especially if the search space is large and the solutions are distributed in a haphazard way. To improve the efficiency, one can sometimes trim the size of the search space and simplify the original problem. Problem reduction [11] is such a method that can be used at the beginning of a search or during search. Once a problem becomes smaller and simpler, search algorithms can go through the space faster. In some cases, problem reduction can solve CSPs without searching [11].

Because neither basic search nor consistency checks alone can always solve CSPs in a timely manner, adding heuristics and using hybrid algorithms are often used to improve efficiency. For instance, a common strategy is to vary the order in which variables, domain values and constraint are searched. Some algorithms incorporate features such as ordering heuristics, (*variable ordering* and *value ordering* [7], and constraint ordering [12], [8]). Furthermore, some stochastic local search techniques have also been developed for solving CSPs (tabu search, iterated local search, ant colony, etc. [10],[9]).

Furthermore, many researchers are working on graph partitioning [4], [6]. The main objective of graph partitioning is to divide the graph into a set of regions such that each region has roughly the same number of nodes and the sum of all edges connecting different regions is minimized. Achieving this objective is a hard problem, although many heuristic may solve this problem efficiently. For instance, graphs with over 14000 nodes and 410000 edges can be partitioned in under 2 seconds [5]. Therefore, we can apply graph partitioning techniques to decompose a binary CSP into semi-independent sub-CSPs.

In this paper, we present a method for structuring and solving binary CSPs. To this end, first, the binary CSP is decomposed into a DFS-tree CSP structure, where each node represents a subproblem. Then, we introduce a heuristic search algorithm which carries out the search in each node according to the partial solution of parent node and the pruning information of children nodes.

In the following section, we present some definitions about CSPs and classify them in three categories. A method of decomposition into a DFS-tree CSP structure is presented in section 3. A heuristic search technique for solving the DFS-tree CSP structure is presented in section 4. An evaluation of our heuristic search technique is carried out in section 5. Finally, we summarize the conclusions in section 6.

2 Centralized, Distributed and Decomposed CSPs

In this section, we present some basic definitions related to CSPs.

*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); a finite collection of constraints $C = \{c_1, ..., c_p\}$ restricting the values that the variables can simultaneously take.

A **solution** to a CSP is an assignment of values to all the variables so that all constraints are satisfied; a problem with a solution is termed *satisfiable* or *consistent*.

*A **binary constraint network*** is one in which every constraint subset 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 [1].

*A **DFS-tree CSP structure*** is a tree whose nodes are composed by subproblems, where each subproblem is a CSP (sub-CSPs). 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 [13].

Each agent has a set of variables; it knows the domains of its variables and a set of *intra-constraints*, ant it attempts to determine the values of its variables. However, there are *inter-constraints* and the value assignment must also satisfy these *inter-constraints*.

2.1 Decomposition of CSPs

There exist many ways for solving a CSP. However, we can classified these problems into three categories: Centralized problems, Distributed problems and Decomposable problems.

- A CSP is a *centralized CSP* when there is 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 examples are: sudoku, n-queens, map coloring, etc.
- A CSP is a *distributed CSP* when the variables, domains and constraints of the underlying network are inherently distributed among agents. This distribution is due entities are identified into the problem (which group a set of variables and constraints among them), constraints may be strategic information that should not be revealed to competitors, or even to a central authority; 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 *decomposable CSP* when the problem can be divided into smaller problems (subproblems) and a coordinating (master-) entity. For example, the search space of a CSP can be decomposed into several regions and a solution could be found by using parallel computing.

Note that centralized and distributed problems are inherent features of problems. Therefore, a distributed CSP can not be solved by a centralized technique. However, can be solved an inherently centralized CSP by a distributed technique? The answer is 'yes' if we previously decompose the CSP.

Usually, real problems imply models with a great number of variables and constraints, causing dense network of inter-relations. This kind of problems can be handled as a whole only at overwhelming computational cost. Thus, it could be an advantage to decompose this kind of problems to several simpler interconnected sub-problems which can be more easily solved.

In the following example we show that a centralized CSP could be decomposed into several subproblems in order to obtain simpler sub-CSPs. In this way, we can apply a distributed technique to solve the decomposed 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 we must to color each country of Europe. In 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 (Spain, Figure 1(3); France, Figure 1(4)) of Europe, it is easy to think that the problem can be decomposed into a set of subproblems, 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 solved by first converting the map into a graph where each region is a vertex, and an edge connects two vertices if and only if the corresponding regions share a border. In our problem of coloring the regions of each country of Europe, it can be observed that the corresponding graph maintains clusters (Spain, Figure 1(3); France, Figure 1(4)) representing each country. Thus, the problems can be solved in a distributed way.

Following, we present a technique for i) decomposing a binary CSP into several sub-CSPs and ii) structuring the obtained sub-CSPs into a DFS-tree CSP structure. Then, in section 4, we propose a heuristic search technique for solving DFS-tree CSP structures.

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. 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 single-trees is minimized.

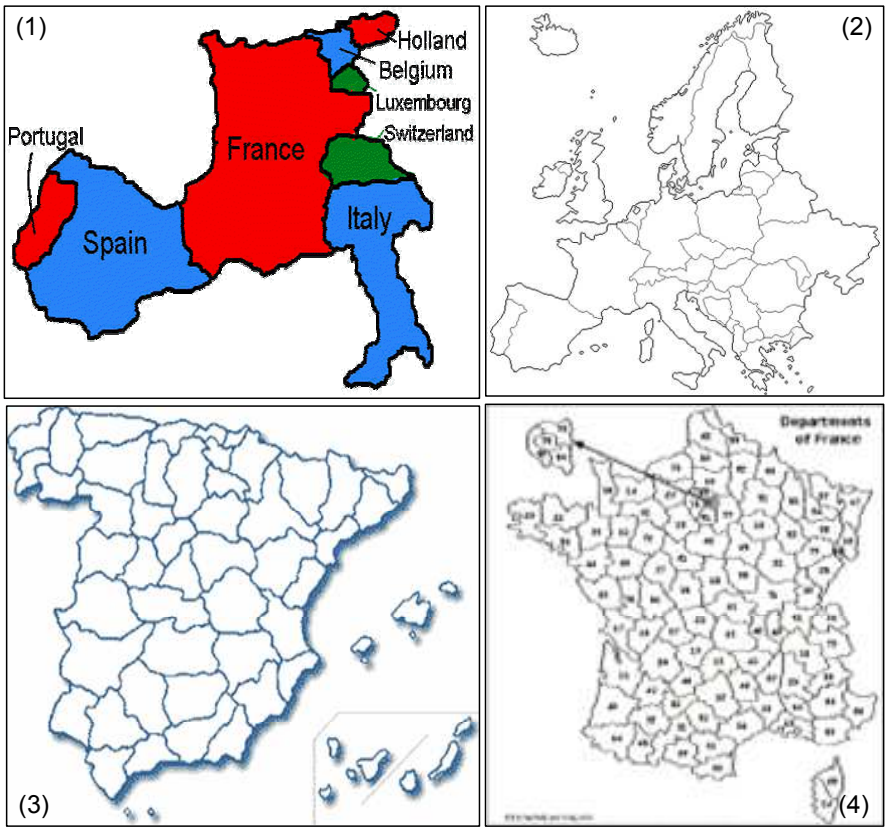

Fig. 1. Map coloring of Europe.

We present a proposal which 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 [5]. 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. 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.

The next step is to build the DFS-tree CSP structure with k *DFS-nodes* in order to be studied by *agents*. This DFS-tree CSP structure is used as a hierarchy to communicate messages between *DFS-nodes*. The DFS-tree CSP structure is built 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 most constrained *DFS-node*. DFSStructure algorithm then simply put *DFS-node* v into 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 calls recursively 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 for whether a *DFS-nodes* is visited until the recursive calls for previous *DFS-nodes* are finished.

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);
mark v as visisted;
forall *DFS-node i adjacent*[1] *to v not visited* **do**
  DFSStructure(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.

3.1 Example of DFS-Tree CSP structure

Figure 2 shows two different representation of 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 $< C, 20, 50, 0.1, 0.1 >$. This problem is very hard to solved with well-known CSP solver methods: Forward-Checking (FC) and Backjumping (BJ). We can see that this problem can be divided into several cluster (see Figure 3) and it can be converted into a DFS-tree CSP structure (see Figure 3). By using our DFS-tree heuristic, it is solved in less than one seconds, and by using FC, this problem has not been solved in several minutes.

In Figure 4 we can see a specific sequence of nodes (a numeric order). Following this sequence, FC algorithm has a great drawback due to the variables without in-links (link with a previous variable): 1, 2, 3, 6, 7, 8, 9, 10, 11 and 16 (see Figure 4). Theses variables have not bounded domains, thus provoking the exploration of all their domains when the algorithm backtracks. This kind of variables is an extreme case of variables with their domain weakly bounded. An example of this situation can be seen in Figure 4, where the variable 12 has its domain bounded by the variable 6. When the bounded domain of the variable 12 has not a valid assignment, FC algorithm backtracks to change the values of the bounded domain, but it will need examine completely the domain of variables 11, 10, 9, 8 and 7 before it changes the assignment of variable 6. In this example, with *domain size* $= 50$, it involves 50^5 assignments in vain.

[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

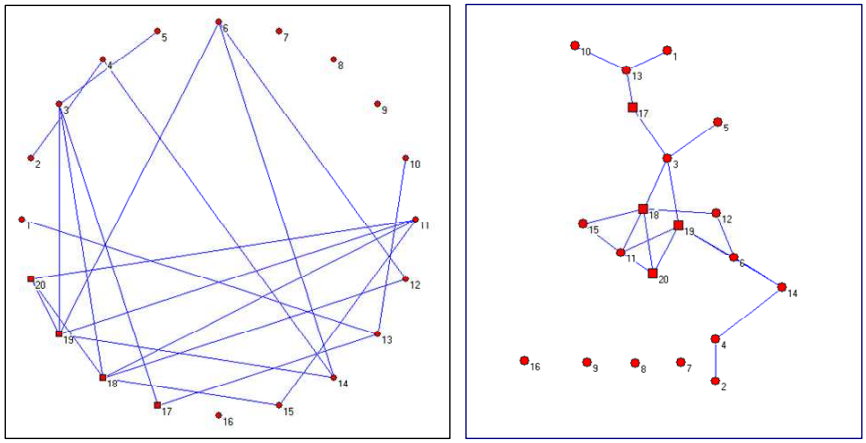

Fig. 2. Random CSP $< n = 20, d = 50, p = 0.1, q = 0.1 >$.

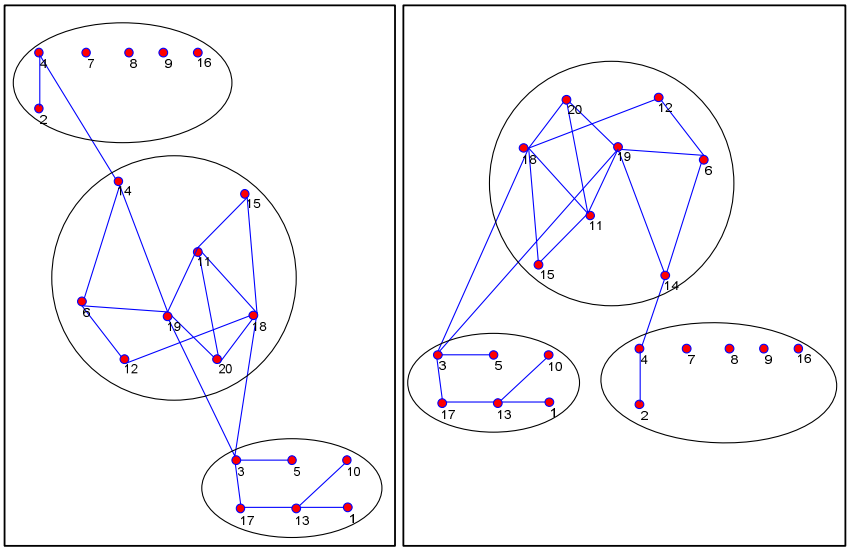

Fig. 3. Left: Decomposed Problem. Right: DFS-Tree CSP structure.

4 DFS-Tree Based Heuristic Search (DTH)

In section 3 we have presented a method for structuring a binary CSP into a DFS-Tree CSP structure. In this section we propose a new heuristic search technique for solving DFS-Tree CSP structures.

Our Heuristic called DFS-Tree Based Heuristic Search (DTH) can be considered as a distributed and asynchronous technique. In the specialized literature, there are many works about distributed CSPs. In [13], Yokoo et al. present a

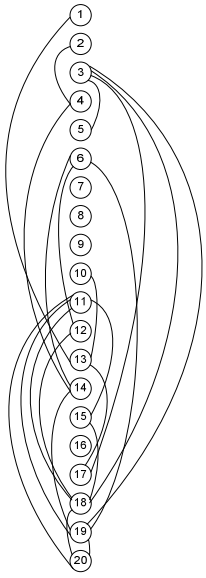

Fig. 4. Sequence of nodes for Forward-Checking Algorithm.

formalization and algorithms for solving distributed CSPs. These algorithms can be classified as either centralized methods, synchronous or asynchronous backtracking [13].

DTH is committed to solve the DFS-tree CSP structure in a *Depth-First Search Tree* (*DFS Tree*) where the root *DFS-node* is composed by the most constrained sub-CSP, in the sense that this sub-CSP maintains a higher number of single-nodes. DFS trees have already been investigated as a means to boost search [2]. 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, 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*.

Thus, the root *agent* works on its subproblem (root meta-node). If the root *agent* finds a solution then it sends the consistent partial state to its children *agents* in the *DFS-tree*, 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 sends again 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

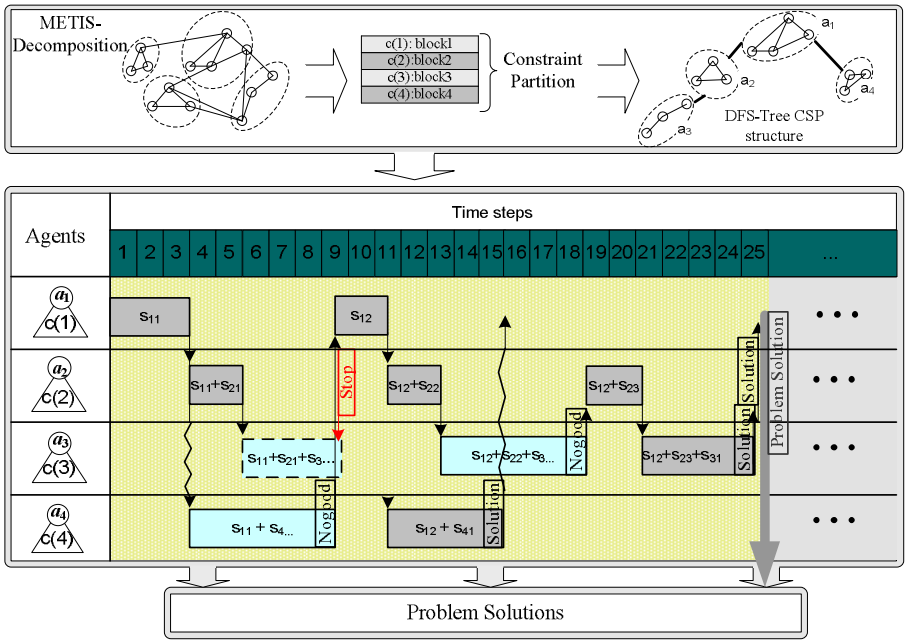

Fig. 5. Decomposing Technique and DFS-Tree Based Heuristic Search.

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 which 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. However, if the root *agent* does not find solution, then DTH returns *no solution found*.

The heuristic technique is based on Nogood *information*, which allows us to prune search space. First, DTH uses Nogood *information* to prune the search space jumping to the variable involved in the Nogood within the lowest level in the search sequence. Furthermore, DTH does not backtracks in the inverse order of the search, but it jumps directly to other variable involved in the Nogood. Therefore, the search space is pruned in this level of the search sequence and solutions can be deleted. This heuristic technique is not complete. Its level of completeness depend on how Nogood *information* is used.

Figure 5 shows our technique for de-composing a CSP into a DFS-tree CSP structure. Then, DTH is carried out. The root *agent* (a_1) starts the search process finding a partial solution. Then, it sends this partial solution to its

children. The *agents*, which are brothers, 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 solution. For example, (state $s_{12}+s_{41}$) + (state $s_{12}+s_{23}+s_{31}$) is a problem solution. The concurrence can be seen in Figure 5 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 of the global problem will be found. It happens in *Time step* 25.

Let's see an example to analyze the behavior of DTH (Figure 6). First the constraint network of Figure 6(1) is partitioned in 3 sub-CSPs and the *DFS tree CSP structure* is built (Figure 6(2)). *Agent a* finds its first partial solution ($X_1 = 1, X_2 = 1$) and sends it to its children: *agent b* and *agent c* (see Figure 6(3)). This is a good partial solution for *agent c* (Figure 6(4)), but this partial solution empties the $X3$ variable domain, thus *agent b* sends a Nogood message to its father (Nogood ($X_1 = 1$)) (Figure 6(5)). Then, *agent a* processes the Nogood message, prunes its search space, finds a new partial solution ($X_1 = 2, X_2 = 2$) and sends it to its children (Figure 6(6)). At this point in the process, *agent c* sends a Nogood message to its father (Nogood ($X_1 = 2$)) because $X5$ variable domain is empty (Figure 6(7)). *Agent a* stops the search of *agent b* (Figure 6(8)) and then it processes the Nogood message, prunes its search space, finds a new partial solution ($X_1 = 3, X_2 = 3$) and sends it to its children (Figure 6(9)). This last partial solution is good for both children, thus they respond with a OK message and the search finishes (Figure 6(10)).

4.1 DTH: soundness

If no solution is found, this algorithm terminates. For instance, in Figure 6(10), if *agent b* and *agent c* send Nogood messages, then the root agent empties its domain and terminates with "no solution found".

For the agents to reach a consistent state, all their assigned variable values must satisfy all constraints (*inter-constraints* and *intra-constraints*). Thus, the soundness of DTH is clear.

What remains is that we need to show that DTH must reach one of these conclusions in finite time. The only way that DTH might not reach a conclusion is at least one *agent* is cycling among its possible values in an infinite processing loop. Given DTH, we can prove by induction that this cannot happen as follows.

In the base case, assume that root *agent* is in an infinite loop. Because it is the root agent, it only receives Nogood messages. When it proposes a possible partial state, it either receives a Nogood message back, or else gets no message back. If it receives Nogood messages for all possible values of its variables, then it will generate an empty domain (any choice leads to a constraint violation) and DTH will terminate. If it does not receive a Nogood message for a proposed

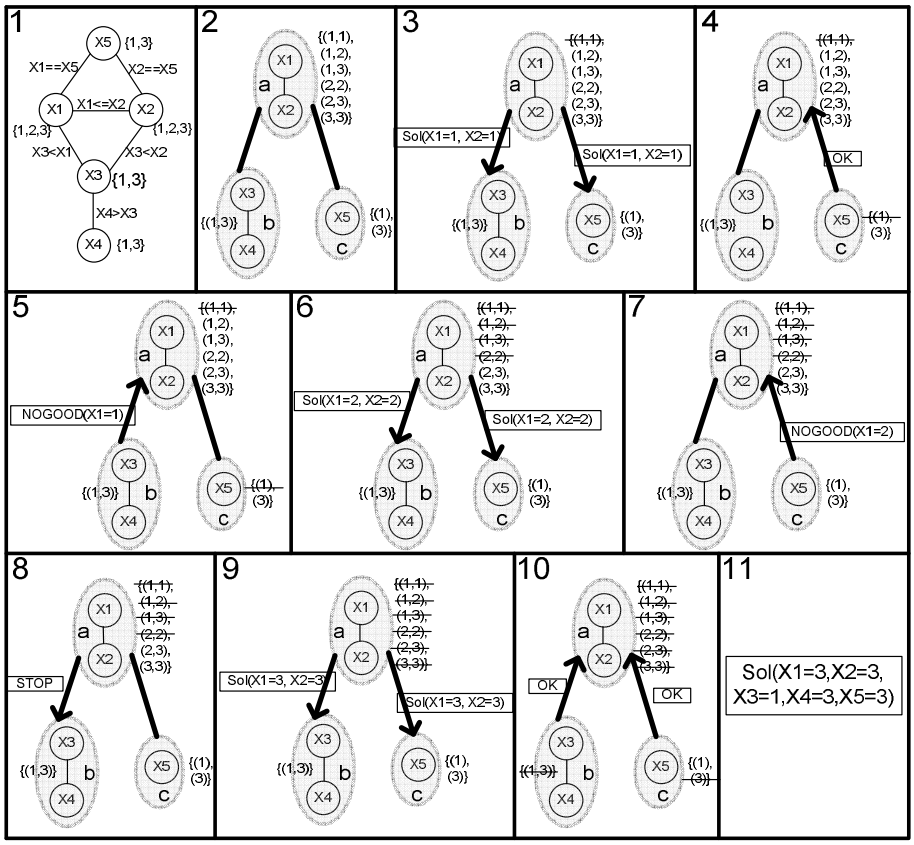

Fig. 6. Example of DTH.

partial state, then it will not change that partial state. Either way, it cannot be in an infinite loop. Now, assume that from root *agent* to *agent* in level $k-1$ in the tree ($k > 2$) maintain a consistent partial state, and *agent* in level k is in an infinite processing loop. In this case, the only messages that *agent* in level k receives are Nogood messages from its children. *Agent* in level k will change instantiation of its variables with different values. Because its variable's domain is finite, this *agent* in level k will exhaust the possible values in a finite number of steps and sends a Nogood message to its parent (which contradicts the assumption that *agent* in level k is in a infinite loop). Thus, by contradiction, *agent* in level k cannot be in an infinite processing loop.

5 Evaluation

In this section, we carry out an evaluation between DTH and 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 $< a, n, k, t, p1, p2, q1, q2 >$, where a was the number of sub-CSPs, n was the number of variables on each sub-CSP, k the values in each domain, t 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 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 [3]. The problems were randomly generated by using the generator library in [3] and by modifying these parameters. For each 8-tuple $< a, n, k, t, p1, p2, q1, q2 >$, we have tested these algorithms with 50 problem instances of random CSPs. Each problem instance execution has a limited CPU-time (TIME_OUT) of 900 seconds.

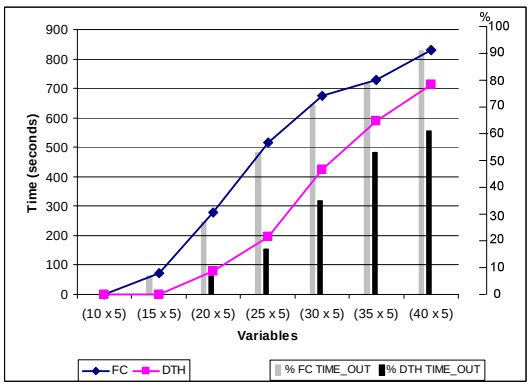

Fig. 7. Running Times and percentage of solution over $TIME\_OUT = 900$ seconds with different variable number.

In Figures 7 and 8 we compare the running time of DTH with a well-known complete algorithm: Forward Checking. In Figure 7, number of variables on each sub-CSP was increased from 10 to 40, the rest of parameters were fixed ($< 5, n, 8, 0.2, 0.01, 0.2, 0.9, 0.3 >$). We can observe that DTH outperformed the FC algorithm in all instances. DTH running times were lower than FC running times. Furthermore, DTH had less TIME_OUT executions than FC. As the number of

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

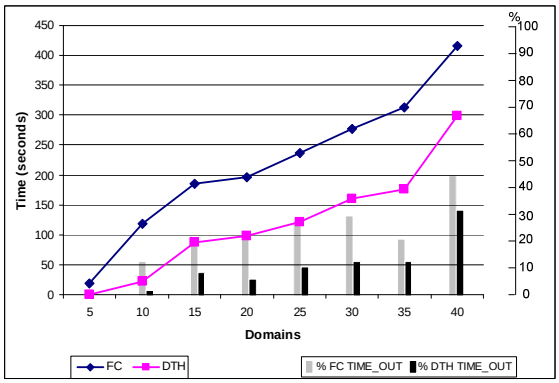

Fig. 8. Running Times and percentage of solution over $TIME\_OUT = 900$ seconds with different domain size.

variables increased, the number of problem instance executions which achieve TIME_OUT was increased, that is precisely why running time increase more slowly when number of variables increases.

In Figure 8, the domain size was increased from 5 to 40, the rest of parameters were fixed ($< 5, 15, d, 0.2, 0.01, 0.2, 0.9, 0.3 >$). It can be observed that DTH outperformed FC in all cases. As the domain remained greater, the computational cost of DTH and FC also increased. As the domain size increased, the number of problem instance execution which achieve TIME_OUT was increased. Again, DTH had less TIME_OUT executions than FC.

Figure 9 shows the behavior of DTH and FC by increasing $q2$ in several instances of $q1$ with 5 sub-CSPs, 15 variables and domain size of 10 ($< 5, 15, 10, 0.2, 0.01, 0.2, q1, q2 >$). It can be observed that:

- independently of $q1$ and $q2$, DTH maintained better behaviors than FC.
- As $q1$ increased, problem complexity increased and running time also increased. However, as problem complexity increased, DTH carried out a better pruning. Thus, DTH running times were rather better than FC running times as $q1$ increased.
- Problem complexity also depend on $q2$. With regard to $q2$, the most complex problems were those problems generated, in general, with half-values of $q2$ ($q2 = 0.3$ and $q2 = 0.5$). Moreover, for high values of $q1$, problems were also complex with low values of $q2$ ($q2 = 0.1$). In all these cases, DTH had the best behaviors.

Figure 10 shows the percentages of unsolved problems by DTH when tightness of q1 increased, that is, problem complexity increased. Problem instances are the same as Figure 9 instances. When problem complexity increased, the number of unsolved solution also increased. However the number of unsolved solution by DTH was always smaller than the number of TIME_OUTs when we

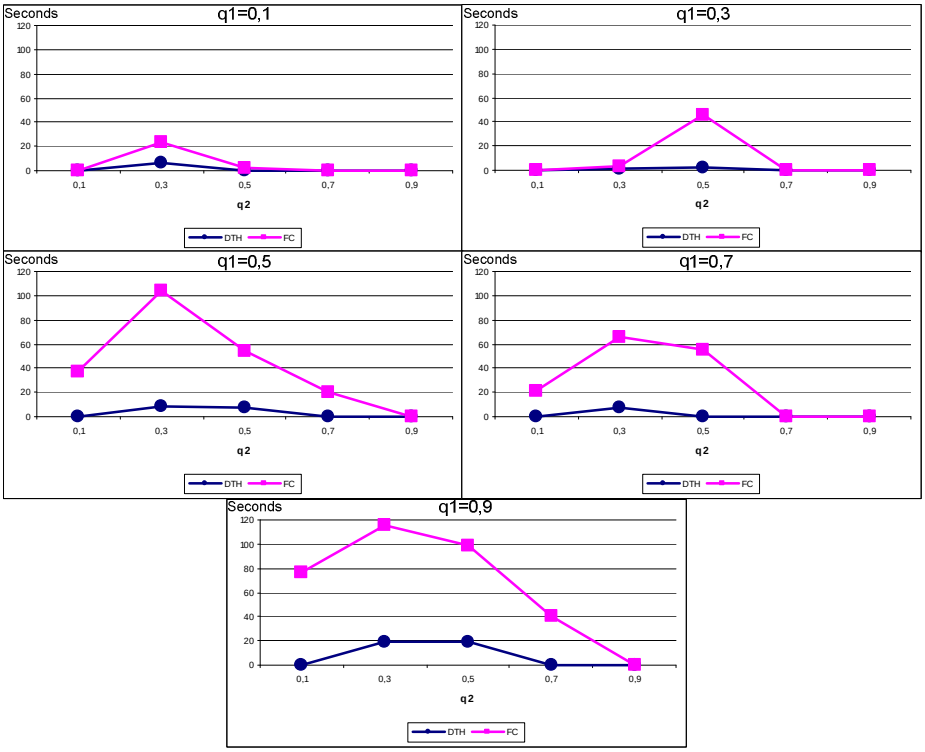

Fig. 9. Running Times with different q1 and q2.

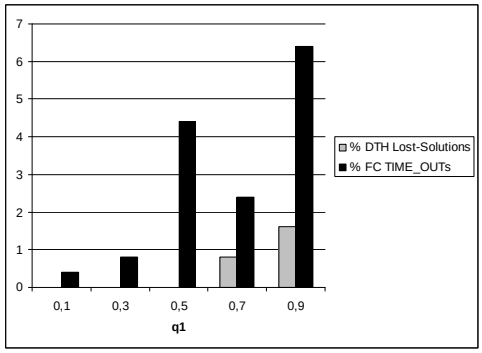

Fig. 10. Percentage of unsolved problems by DTH and percentage of FC TIME_OUTs with different values of $q1$.

run FC algorithm. Therefore, our heuristic search is a good option for solving complex decomposable CSPs when we have a time limit.

6 Conclusions

We have proposed an heuristic approach for solving distributed CSP. First, we translate the original CSP into a DFS-tree CSP structure, where each node in the DFS-tree is a sub-CSP. Our heuristic proposal is a distributed heuristic for solving the resultant DFS-tree CSP structure. This heuristic exploits the Nogood *information* for pruning the search space. DTH is sound but not complete. The evaluation shows a good behavior in decomposable CSPs; particularly, the results of the heuristic search are better as problem complexity increases. Furthermore, completeness degree of the heuristics is appropriate for solving decomposable CSPs. Thus, this technique became suitable for solving centralized problems that can be decomposed in smaller subproblems in order to improve solution search.

References

1. R. Dechter, 'Constraint networks (survey)', *Encyclopedia Artificial Intelligence*, 276–285, (1992).
2. R. Dechter, *Constraint Processing*, Morgan Kaufmann, 2003.
3. 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).
4. B. Hendrickson and R.W. Leland, 'A multi-level algorithm for partitioning graphs', in *Supercomputing*, (1995).
5. G. Karypis and V. Kumar, 'Using METIS and parMETIS', (1995).
6. G. Karypis and V. Kumar, 'A parallel algorithm for multilevel graph partitioning and sparse matrix ordering', *Journal of Parallel and Distributed Computing*, 71–95, (1998).
7. N. Sadeh and M.S. Fox, 'Variable and value ordering heuristics for activity-based jobshop scheduling', *In proc. of Fourth International Conference on Expert Systems in Production and Operations Management*, 134–144, (1990).
8. M.A. Salido and F. Barber, 'A constraint ordering heuristic for scheduling problems', *In Proceeding of the 1st Multidisciplinary International Conference on Scheduling : Theory and Applications*, **2**, 476–490, (2003).
9. C. Solnon, 'Ants can solve constraint satisfaction problems', *IEEE Transactions on Evalutionary Computation*, **6**, 347–357, (2002).
10. T. Stutzle, 'Tabu search and iterated local search for constraint satisfaction problems', *Technischer Bericht AIDA9711, FG Intellektik, TU Darmstadt*, (1997).
11. E. Tsang, *Foundation of Constraint Satisfaction*, Academic Press, London and San Diego, 1993.
12. R. Wallace and E. Freuder, 'Ordering heuristics for arc consistency algorithms', *In Proc. of Ninth Canad. Conf. on A.I.*, 163–169, (1992).
13. M. Yokoo and K. Hirayama, 'Algorithms for distributed constraint satisfaction: A review', *Autonomous Agents and Multi-Agent Systems*, **3**, 185–207, (2000).