Backtracking in Distributed Constraint Networks

Youssef Hamadi[1] and **Christian Bessière**[1] and **Joël Quinqueton**[1,2]

Abstract. The adaptation of software technology to distributed environments will be an important challenge in the next few years. In the scope of constraint reasoning, few works have been published on the adaptation of algorithms searching for a solution in a constraint network to distributed constraint networks. This paper presents a new search procedure for finding a solution in a distributed constraint network. Although based on the principle of backtracking to ensure the completeness of search, this procedure allows a high level of asynchronism, i.e., simultaneous search on independent parts of the network. Furthermore, it fits its behavior to the structure of the constraint graph in order to minimize message passing and to avoid useless restorations when a dead-end is reached. We also present a generic distributed method for computing any variable ordering heuristic.

1 INTRODUCTION

The constraint satisfaction problem (CSP) is a powerful framework for general problem solving. It involves finding a solution to a constraint network, i.e., finding values for problem variables subject to constraints that are restrictions on which combinations of values are acceptable. It is widely used in artificial intelligence, its applications ranging from machine vision to crew scheduling and many other fields (see [8] for a survey). The basic method to search for solution in a constraint network is backtrack search, which performs a systematic exploration of the search tree until it finds an instantiation of values to variables that satisfies all the constraints. In a distributed framework, this process needs the definition of interaction protocols between the different processing units. If we except some theoretical works which use a shared memory communication model, these protocols usually interact by the way of message passing operations.

At that point, we have to distinguish between parallel based work, and distributed work. In parallel backtracking algorithms [7], p processors concurrently perform a local backtracking algorithm in disjoint parts of the search tree. Distributed implementations of backtracking are different. A processor (agent) does not implement a local backtracking. Agent interactions participate to the search. The search algorithm is obtained as the result of an emergence in the whole interaction of agents in the agents society. A classification of different types of multi-processor search in CSPs can be found in [5].

In this paper, we deal only with the search for solutions in constraint networks distributed among several agents, with no restriction on the location of each agent. This means that our goal is not to distribute search algorithms among several agents to speed up the running time of central algorithms. We distribute search algorithms among several agents to solve problems that are distributed among these agents, and for which a loading of the whole problem on a single site is impossible. (It can be impossible because of the time/cost of the loading, because the whole data does not fit to the size of one machine, or because of security protections on some data that cannot be brought from one site to another.) We think that this approach is an important challenge for the next few years because of the increasing number of environments distributed on several sites linked by internet facilities.

The more related work on search for solution in a distributed constraint network was presented in [9]. The authors present a distributed realization of the backtracking algorithm: *asynchronous backtracking*. Their method performs a distributed search with a learning technique, nogood recording. This non-bounded recording of previous local failures ensures the completeness of their algorithm.

The basis of our *distributed backtracking* (*DIBT*) is the backtracking algorithm (BT) [2]. We ensure the completeness of the algorithm by an exhaustive domain exploration. We avoid learning schemes, such as nogood recording, and we present severals ways of improvement of BT search process according to the distributed context. For example, since agents are connected only according to the constraint network structure, we obtain for free a graph-based backjumping (GBJ [1]) behavior during failure phases. Instantiation information is also transmitted between connected agents, so it also benefits from the constraint network topology. DIBT uses a conservative strategy for saving benefits of previous search in independent parts of the network. Furthermore, it is known from studies on central CSPs that the size of the search space greatly depends on variable ordering heuristics. We give a generic distributed method for computing any static variable ordering.

In the following, we first give a basic definition of the CSP/DCSP paradigm, completed by a brief description of our communication model. Then, we present our distributed variable ordering method, and we describe and analyze DIBT. Afterwards, we give an experimentation with random DCSPs, followed by a general conclusion.

2 DEFINITIONS

2.1 Constraint satisfaction problems

A *binary constraint network* involves a set of n *variables* $\mathcal{X} = \{X_1, \ldots, X_n\}$, a set of *domains* $\mathcal{D} = \{D_1, \ldots, D_n\}$ where D_i is the finite set of possible *values* for variable X_i, a set $\mathcal{C} = \{C_{ij}, C_{kl}, \ldots\}$ of the couples of variables (arcs) that are linked by a *constraint*, and a set $\mathcal{R} = \{R_{ij}, R_{kl}, \ldots\}$ of the constraints (or *relations*) between the pairs of variables specified in $\mathcal{C}$. A relation R_{ij} on the couple of variables $C_{ij} = (X_i, X_j)$ is a subset of the Cartesian product $D_i \times D_j$ that specifies the *allowed* combinations of values for the variables X_i and X_j. R_{ij} can be any Boolean function defined on $D_i \times D_j$, such that $R_{ij}(a, b)$ returns true if and only if the

[1] LIRMM (UMR 5506 CNRS) - UMII, 161 rue Ada, 34392 Montpellier Cedex 5 France, {hamadi, bessiere, jq}@lirmm.fr
[2] INRIA, BP105, 78153 Le Chesnay France

ECAI 98. *13th European Conference on Artificial Intelligence*
Edited by Henri Prade
Published in 1998 by John Wiley & Sons, Ltd.

pair (a, b) is allowed for (X_i, X_j). Asking for the value of $R_{ij}(a, b)$ is called a *constraint check*.

$G = (\mathcal{X}, \mathcal{C})$ is called the *constraint graph* associated to the network $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R})$.

A *solution* to a constraint network is an instantiation of the variables such that all the constraints are satisfied.

The *constraint satisfaction problem* (*CSP*) involves finding a solution in a constraint network.

2.2 Distributed constraint satisfaction problems

A *distributed constraint network* $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R}, \mathcal{A})$ is a constraint network (binary in our case), in which variables and constraints are distributed among a set $\{Agent_1, \ldots, Agent_m\}$ of m autonomous processes called *agents*. Each agent $Agent_k$ "owns" a subset A_k of the variables in $\mathcal{X}$ in such a way that $\mathcal{A} = \{A_1, \ldots, A_m\}$ is a partition of $\mathcal{X}$. The domain D_i (resp. D_j), the arc C_{ij} (resp. C_{ji}) and its associated relation R_{ij} (resp. R_{ji}) belong to the agent owning X_i (resp. X_j)[3]. In the present work, we limit our attention to the extreme case, where there are n agents, each only owning one variable, so that $\mathcal{A} = \mathcal{X}$. Thus, in the following, $Agent_i$ will refer to the agent owning variable X_i. The graph of *acquaintances* in the multi-agents society matches the constraint graph. So, for an agent $Agent_i$, Γ is the set of its acquaintances, namely the set of all the agents $Agent_j$ such that X_j shares a constraint with X_i.

The *distributed* CSP (*DCSP*) involves finding a solution in a distributed constraint network.

For a DCSP, we assume the following communication model [10]. Agents communicate by sending messages. An agent can send messages to other agents if and only if it knows their address in the network (i.e., they belong to its acquaintances). . The delay in delivering messages is finite. For the transmission between any pair of agents, messages are received in the order in which they are sent. Agents use the following primitives to achieve *message passing* operations:

- $sendMsg(dest,$"m"$)$ sends message m to the agents in $dest$.
- $getMsg()$ returns the first unread message available.

3 DISTRIBUTED BACKTRACKING

BT is a general complete resolution technique. Given a variable and value ordering, it generates successive instantiations of the problem variables. It tries to extend a partial instantiation by taking the next variable in the ordering and by assigning it a value consistent with previously assigned variables. If no value can be found for the considered variable, the algorithm backtracks. In the basic BT scheme, it goes back to the previous variable in the ordering and changes its value. In some refined backtracking schemes, the algorithm jumps back to the origin of the failure.

Our framework is totally asynchronous but we need an ordering between the agents to apply the backtracking scheme which ensures completeness. Hence, we must find a partial ordering between agents, which will be followed by variable instantiations, and we must complete this partial ordering to a total one for guiding the backtrack steps. Since we have no restriction on the ordering which can be used, it will be worthwhile to use an ordering that fits the constraint graph topology. As in central constraint satisfaction it can reduce the search space, and more, it can significantly reduce message passing. *Asynchronous backtracking* [9] uses the lexicographic ordering of agent tags to avoid infinite processing loops. This ordering does not take advantage of the initial features of the problem.

[3] We suppose that the constraint network is such that $(\mathcal{X}, \mathcal{C})$ is a symmetric graph.

3.1 A generic method for distributed variable ordering

The size of the search space is highly dependent on user's specific heuristic choices such as variable ordering. Usually these heuristics take advantage of knowledge about the variable (domain size) and/or about the variable neighborhood (degree). In this section, we present a generic method for a distributed computation of any static variable ordering in our DCSP framework.

In our system, each agent locally computes its position in the ordering according to the chosen heuristic. Concretely, each agent determines the sets Γ^+ and Γ^-, respectively $children$ and $parent$ acquaintances, w.r.t. an evaluation function f and a comparison operator *op* which totally define the heuristic chosen. This is done in the lines 1 to 2 of Algorithm 1. Notice that the evaluation function f can involve some communication between the agents. To avoid a complex communication behavior, it is better to use heuristics for which the associated function f involves only local communications between neighbor agents. This is the case for the *max-degree* heuristic.

Algorithm 1: Distributed variable ordering

```
   % executed by each agent (self);
   in: f defined on Γ, op a comparison operator;
   out: split of Γ into Γ+ and Γ−, ordering of Γ− ;
   begin
       % Γ split;
1      Γ+ ← ∅; Γ− ← ∅;
       for each Agent_j ∈ Γ do
           if (f(Agent_j) op f(self)) then Γ+ ← Γ+ ∪ {Agent_j};
2          else Γ− ← Γ− ∪ {Agent_j};
       % Γ− ordering;
3      max ← 0;
       for (i = 0; i < |Γ+|; i++) do
           m ← getMsg();
           if (m = value:v; from:j) then
               if (max < v) then max ← v;
       max++;
       sendMsg(Γ−, "value:max; from:self");
       sendMsg(Γ+, "position:max; from:self");
       for (i = 0; i < |Γ−|; i++) do
           m ← getMsg();
           if (m = position:p; from:j) then Level[j] ← p;
4      Rearrange Γ− according to Level[] ;
   end
```

After performing this phase, agents know their *children* (Γ^-) and *parents* (Γ^+) acquaintances. During the search, they will send instantiation value to children, and in case of dead-end, they will backtrack to the *first* agent in Γ^-. So, we need an ordering on Γ^-. This is the second part of Algorithm 1 (lines 3 to 4). Agents without children state that they are at level one, and they communicate this information to their acquaintances. Other agents take the maximum level value received from children, add one to this value, and send this information to their acquaintances. Now, with this new environmental information, each agent rearranges the agents in its local Γ^- set by increasing level. Ties are broken with agent tags.

Figure 1 gives an illustration of this distributed processing for the *max-degree* variable ordering heuristic. On the left side of the figure a constraint graph is represented. For achieving the max-degree heuristic, Algorithm 1 must be called by each agent with the function $f(Agent_i) = |\Gamma_i|$ (where Γ_i is the set of acquaintances of

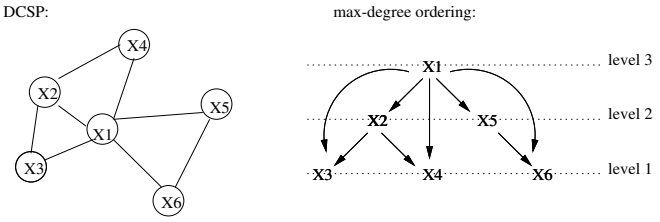

Figure 1. Distributed variable ordering

$Agent_i$) and the comparison operator $op = '<'$. In case of ties, this operator can break them with agent tags. (For a *dom/deg* ordering, $f(Agent_i) = |D_i|/|\Gamma_i|$, $op = '>'$.) Once Algorithm 1 has been applied, the static variable ordering obtained is the one presented on the right side of Fig. 1. Arrows follow the ordering relation, which represents the instantiation transmission order of the search procedure. The ordering of the sets Γ^-, which will be used after each dead-end, is as follows:

- $Agent_1,\ \Gamma^- = \emptyset$
- $Agent_2,\ Agent_5,\ \Gamma^- = \{Agent_1\}$
- $Agent_3,\ Agent_4,\ \Gamma^- = \{Agent_2,\ Agent_1\}$
- $Agent_6,\ \Gamma^- = \{Agent_5,\ Agent_1\}$

Several observations can be made on this new ordering technique. If the evaluation function is well chosen, each agent can perform the ordering algorithm locally, with a constant number of messages with each of its acquaintances. In the resulting ordering, agents at the same level are independent. They do not share any constraint, so they can perform parallel computations at the same time. This point is very important since it leads to a good parallelization feature. Agents in the highest level (top of the hierarchy) are local optima in the constraint graph according to the heuristic.

3.2 Algorithm

The global scheme of the search process is the following (see algorithm 2). Each agent instantiates its variable with respect to its parent constraints. After instantiation, the agent informs its children of its chosen value (message content starting by "**infoVal**"). If no value satisfies the constraints with the agents in Γ^-, a backtrack message is sent to the nearest parent (message content starting by "**btSet**"). Since backtrack occurs between connected variables, it is a graph based backtrack. This message (line 4) includes the local ordered set Γ^- of parent acquaintances, their level positions and agent beliefs about their values. The receiver of the backtrack message, first checks the validity of the message by comparing its current value with the one reported for it in the message (line 6). In case of different values, this means that the sender has not yet received information about the receiver last value, so backtrack decision could be obsolete. If the comparison matches, and if the agent cannot take a different value, it backtracks. It sends a backtrack message to the nearest agent in the ordered set union of Γ^- and the sender set (line 9). This new set is attached to the message with related values and level positions of agents for ensuring continuity of graph based backjumping.

Termination occurs when agents are instantiated and wait for a message or when a source agent finds an empty backtrack set (line 8). In the last case, the problem has no solution. A message $noSolution$ is sent to a *system agent*. This agent stops the distributed computation by broadcasting a *stop* message in the whole multi-agent system. This system agent executes a global state detection algorithm [4]. Global satisfaction occurs when agents instantiated according to parent constraints are waiting for a message (line 2) and when no message transits in the communication network.

DIBT uses the following structures and methods:

- self is the agent running the algorithm, D_{self} is its domain, and $myValue$ its current value.
- $value[]$ stores parent acquaintances values.
- first(S) returns the first element of an ordered set S. With our application, returns the nearest agent in S.
- $getValue$(type),
  - if $type$='info', returns the first value in D_{self} compatible with agents in Γ^-, starting at $myValue$[4].
  - if $type$='bt', returns the first value after $myValue$ in D_{self} compatible with agents in Γ^-.
- $merge$(s1,s2) takes two ordered sets and returns their ordered union.

Algorithm 2: Distributed backtracking

```
   begin
     compute Γ+, Γ−, and rearrange Γ− with Alg. 1;
     nearest ← first(Γ−);
1    myValue ← getValue(info);
     sendMsg(Γ+, "infoVal:myValue; from:self");
     end ← false;
     while (!end) do
2      m ← getMsg();
       if (m = stop) then end ← true;
3      if (m = infoVal:a; from:j) then
         value[j] ← a;
         myValue ← getValue(info);
         if (myValue) then
           sendMsg(Γ+, "infoVal:myValue; from:self");
         else
4          sendMsg(nearest, "btSet:Γ−; Values:value[Γ−]");
5      if (m = btSet:set; Values:values) then
6        if (values[i]=myValue) then
7          myValue ← getValue(bt);
           if (myValue) then
             sendMsg(Γ+, "infoVal:myValue; from:self");
           else
8            if (Γ− = ∅ and set = ∅) then
               sendMsg(system, "noSolution");
               end ← true;
             else
               followSet ← merge(Γ−, set);
               follow ← first(followSet);
9              sendMsg(follow, "btSet:followSet; Values:value[Γ−] ∪ values")
10             if (follow ∉ Γ−) then
                 myValue ← getValue(info);
   end
```

[4] The search for a new compatible value starts from the current value for keeping the maximum of previous work. For ensuring completeness, the values that are before $myValue$ in D_{self} are put at the end of D_{self}.

3.3 Analysis

We present here the important features of DIBT. They improve graph based backjumping in several ways.

- We use an 'intelligent' instantiation scheme:
  Central GBJ uses the constraint graph for backtracking to the origin of the failure. Our system naturally does the same. But the central method successively instantiates variables even when they are independent (i.e., not linked by a constraint). With our method, two variables have a direct instantiation precedence order (from parent to child) only if they are connected. Therefore, successive instantiations are always constraint dependent.
- Our system preserves previous work:
  When an **infoVal** message occurs at $Agent_i$, $getValue$ tests if the current value $myValue$ satisfies the constraint with the sender before trying another value in D_i. If it does, no change occurs, and more importantly, no changes are reported to children, so the maximum of previous work is kept.
- Inherent dynamic feature:
  At the beginning, agents do not know their parent values. Nevertheless, they are instantiated in line 1. When an agent does not know the instantiation of one parent, it does not consider the related constraint (function $getValue()$). During the search, acquaintances values are stored in $value[]$ and instantiations of agent variables are changed according to acquaintances known values. This point is important in our asynchronous system. In fact, constraints 'appear' with related values.
- Repair-like technique:
  Initially, the whole system gets initial parallel instantiation of variables (line 1). During concurrent resolution, agents revise their selected value according to their environment. So, the system starts with a global instantiation of the problem variables and performs local repairs on different parts of the instantiation. Its looks like classical repair methods [6], but here, it is parallelized and complete.

Correctness of DIBT For a complete proof of the algorithm, the following points must be considered. Whatever is the variable ordering heuristic, the directed graph induced by the Γ sets given by Algorithm 1 has no circuit. Thus, a backtrack (resp. instantiation) step from an agent cannot lead to an incoming **bt** (resp. **infoVal**) message from one of its children (resp. parents). During a backtrack step, only the remaining values are tested for satisfaction, and during incoming **infoVal** messages, the agent reconsiders its whole initial domain. The previous observations associated with finite time message transmission and the correctness of termination detection [4] are the keys of the algorithm correctness.

4 EXPERIMENTATIONS

Asynchronous backtracking [9] is the more related work on search for solution in a distributed constraint network. Nevertheless, we do not present a comparison of DIBT with that algorithm. It has indeed several drawbacks that prevent its use on real-world problems. In the worst case, agents record (with duplication) the entire search space. Furthermore, this recorded information must be checked for satisfaction at each new instantiation of a value to a variable. Hence, the cost of an assignment grows according to nogood recording. Finally, even if the original problem is a *binary* one (i.e., involving only binary constraints), nogood storage can add higher arity constraints, which are source of expensive communication, expensive checking, and memory explosion [3, 1].

Our goal in this section is the evaluation of our method in a really distributed environment, when it is implemented with and without the distributed variable ordering algorithm of Section 3.1. We gave a physical processor to each agent in the multi-agents system and a variable to each agent. In real applications, an agent can store several variables, and so, a subnetwork instead of a variable. In this case, the agent satisfaction corresponds to the involved subnetwork consistency. Since our local network has few machines, we solved uniform random DCSPs with only 15 variables and domains of size 5. We fixed the connectivity at 30%. We varied constraint tightness from 10% to 90%. For each tightness, we generated 10 problems. We compared the execution of our distributed method with the *max-degree* variable ordering heuristic to its execution without any heuristic, i.e., *lexicographic* ordering.

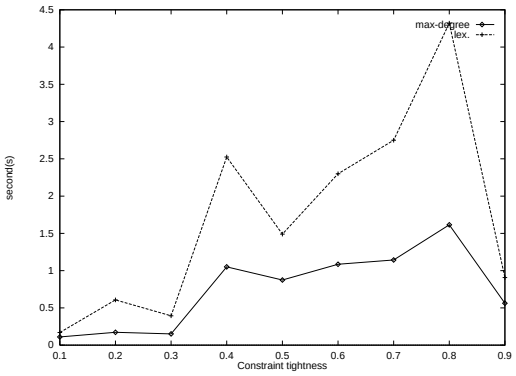

Figure 2. DCSPs with 15 variables, 5 values per domain, and 30% of connectivity (cpu time)

Figure 2 presents mean time results. The time considered for one instance is the one of the last terminated agent. We see that, as in a central framework, *max-degree* ordering leads to a faster resolution. *max-degree* needs also less constraints checks (Figure 3), and less communication (Figure 4).

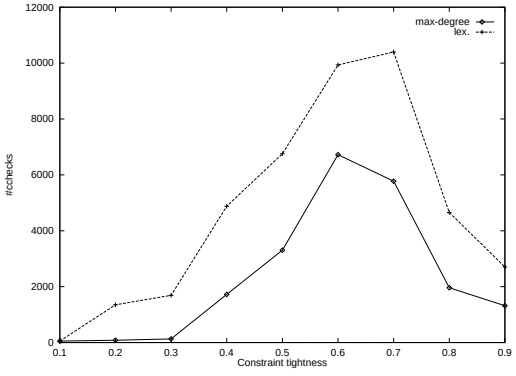

Figure 3. DCSPs with 15 variables, 5 values per domain, and 30% of connectivity (constraint checks)

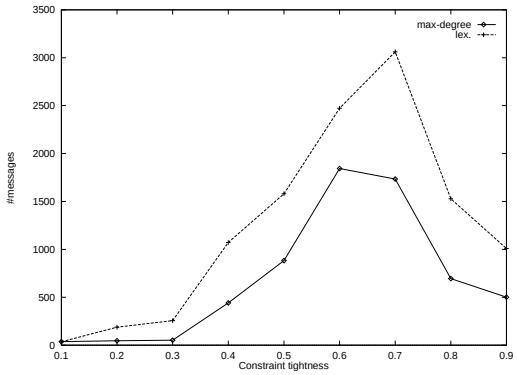

Figure 4. DCSPs with 15 variables, 5 values per domain, and 30% of connectivity (message passing)

5 CONCLUSION AND FUTURE WORK

We have presented DIBT, a fully distributed asynchronous system for solving distributed CSPs. Our method performs a distributed backtrack search. This allows us to prove both its completeness and termination. The whole system fits the constraint network. Hence, communications always occur between connected variables. We use an 'intelligent' backtrack technique as in central backjumping schemes since agents always backtrack to the nearest connected. Furthermore, we also use the constraint graph topology for instantiations. So, DIBT is clever during backtrack steps, but also during instantiations. Constraint checks are parallelized and the whole system operates in a conservative way by keeping the maximum of previous search effort when a change occurs. More, asynchronism features of DIBT are in some ways close to repair methods. At the beginning, all the agents take an instantiation. They revise their instantiations according to incoming information. The system can be assimilated to a whole initial assignment with local parallel perturbations. From the previous remarks we conclude that our distributed backtracking algorithm combines various ideas of central methods. We also presented a fully distributed method to compute variable ordering heuristics. We showed that even in distributed frameworks variable ordering affects the efficiency of search. In the near future, we will test the effect of adding dynamic variable ordering to DIBT. But, many other extensions should be done. Particularly, including some look ahead during search is probably one of the most important improvements to incorporate in our system.

ACKNOWLEDGEMENTS

We would like to thank the referees for their helpful comments, and C. Fiorio for his Algorithm LaTeX style.

REFERENCES

[1] R. Dechter, 'Enhancements schemes for constraint processing: backjumping, learning and cutset decomposition', *Artificial Intelligence*, **41**(3), 273–312, (1990).

[2] S.W. Golomb and L.D. Baumert, 'Backtrack programming', *Journal of the ACM*, **12**(4), 516–524, (1965).

[3] R.J. Bayardo Jr. and D.P. Miranker, 'A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem', in *Proceedings AAAI*, pp. 298–304, (1996).

[4] L. Lamport K. M. Chandy, 'Distributed snapshots: Determining global states of distributed systems', *TOCS*, **3**(1), 63–75, (1985).

[5] Q. Y. Luo, P. G. Hendry, and J. T. Buchanan, 'Strategies for distributed constraint satisfaction problems', in *Proceedings 13th Int. Workshop on DAI*, pp. 207–221, (1994).

[6] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird, 'Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems.', *Artificial Intelligence*, **58**, 161–205, (1992).

[7] V.N. Rao and V. Kumar, 'On the efficiency of parallel backtracking', *IEEE Transactions on Parallel and Distributed Systems*, **4**(4), 427–437, (1993).

[8] H. Simonis, 'A problem classification scheme for finite domain constraint solving', in *Proceedings of the CP'96 workshop on Constraint Programming Applications: An Inventory and Taxonomy*, pp. 1–26, Cambridge MA, (1996).

[9] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara, 'Distributed constraint satisfaction for formalizing distributed problem solving', in *Proceedings 12th Int. Conf. on Distributed Computing Systems*, pp. 614–624, (1992).

[10] M. Yokoo and K. Hirayama, 'Distributed breakout algorithm for solving distributed constraint satisfaction problems', in *Proceedings ICMAS*, pp. 401–408, (1996).