

# FRODO: A FFramework for Open/Distributed constraint Optimization

Adrian Petcu

Ecole Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne (Switzerland)

adrian.petcu@epfl.ch

Technical Report EPFL:2006/001

**Abstract.** We present a framework for distributed combinatorial optimization. The framework is implemented in Java, and simulates a multiagent environment in a single Java virtual machine. Each agent in the environment is executed asynchronously in a separate execution thread, and communicates with its peers through message exchange.

The framework is highly customizable, allowing the user to implement and experiment with any distributed optimization algorithm.

Support for synchronous/asynchronous message passing, monitoring and statistics, as well as problem visualization tools are provided.

A number of distributed algorithms are already implemented in this framework, like the Distributed Breakout Algorithm [17] and the DPOP Algorithm [13].

A number of random evaluation problems are also provided, from two distinct domains: meeting scheduling and resource allocation in a sensor network.

## 1 Introduction

Constraint satisfaction and optimization are powerful paradigms that model a large range of tasks like scheduling, planning, optimal process control, etc. Traditionally, such problems were gathered into a single place, and a centralized algorithm was applied in order to find a solution. However, problems are sometimes naturally distributed, so Distributed Constraint Satisfaction (DisCSP) was formalized by Yokoo in [16]. These problems are divided between a set of agents, which have to communicate among themselves to solve them.

DisCSP has a number of practical advantages over its centralized counterpart. Centralized solving may be inappropriate due to privacy and data integration problems. Dynamic systems are another reason: by the time we manage to centralize the problem, it has already changed.

To address distributed optimization, complete algorithms like OptAPO and ADOPT have been recently introduced. ADOPT [5] is a backtracking based bound propagation mechanism. It operates completely decentralized, and asynchronously. The downside is that it may require a very large number of messages, thus producing big communication overheads. OptAPO [4] centralizes parts of the problem; it is unknown a priori how much needs to be centralized where, and privacy is an issue. On the positive side, its communication requirements may be not that extreme.

Distributed local search methods like DSA ([3]) / DBA([18]) for optimization, and DBA for satisfaction ([17]) start with a random assignment, and then gradually improve it. Sometimes they produce good results with a small effort. However, they offer no guarantees on the quality of the solution, which can be arbitrarily far from the optimum. Termination is only clear for *satisfaction* problems, and only if a solution was found.

*DPOP* (see [13]) is a dynamic programming based algorithm that generates a linear number of messages. However, in case the problems have high induced width, the messages generated in the high-width areas of the problem become too large.

There have been proposed a number of variations of this algorithm that address this problem and other issues, offering various tradeoffs (see [14, 9, 8, 11, 10, 15]). [8] proposes an approximate version of this algorithm, which allows the desired tradeoff between solution quality and computational complexity. An anytime algorithm is also presented, which provides increasingly accurate solutions while the propagation is still in progress. This makes it suitable for very large, distributed problems, where the propagations may take a long time to complete.

This paper presents FRODO: a Framework for Open/Distributed Optimization [6]. The framework is implemented in Java, and simulates a multiagent environment in a single Java virtual machine. Each agent in the environment is executed asynchronously in a separate execution thread, and communicates with its peers through message exchange.

Support for synchronous/asynchronous message passing, monitoring and statistics, as well as problem visualization tools are provided.

A number of distributed algorithms are already implemented in this framework, like the Distributed Breakout Algorithm [17] and the DPOP Algorithm [13].

A number of random evaluation problems are also provided, from two distinct domains: meeting scheduling and resource allocation in a sensor network.

## 2 Definitions & notation

**Definition 1.** A discrete multiagent constraint optimization problem (*MCOP*) is a tuple  $\langle \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$  such that:

- $\mathcal{X} = \{X_1, \dots, X_m\}$  is the set of variables, each controlled by an homonym agent;
- $\mathcal{D} = \{d_1, \dots, d_m\}$  is a set of discrete, finite domains of the variables;
- $\mathcal{R} = \{r_1, \dots, r_p\}$  is a set of relations, where a relation  $r_i$  is a function  $d_{i1} \times \dots \times d_{ik} \rightarrow \mathbb{R}$  which denotes how much utility is assigned to each possible combination of values of the involved variables (negative values can be used to model costs).

In this paper we deal with unary and binary relations; the extension to higher arity relations is shown in [15]. In a MCOP, any value combination is allowed; the goal is to find an assignment  $\mathcal{X}^*$  for the variables  $X_i$  that maximizes the aggregate overall utility. Hard constraints can be simulated by assigning large negative valuations to disallowed tuples, and 0 to allowed ones.

## 3 Framework structure

# Structure of the multiagent optimization framework

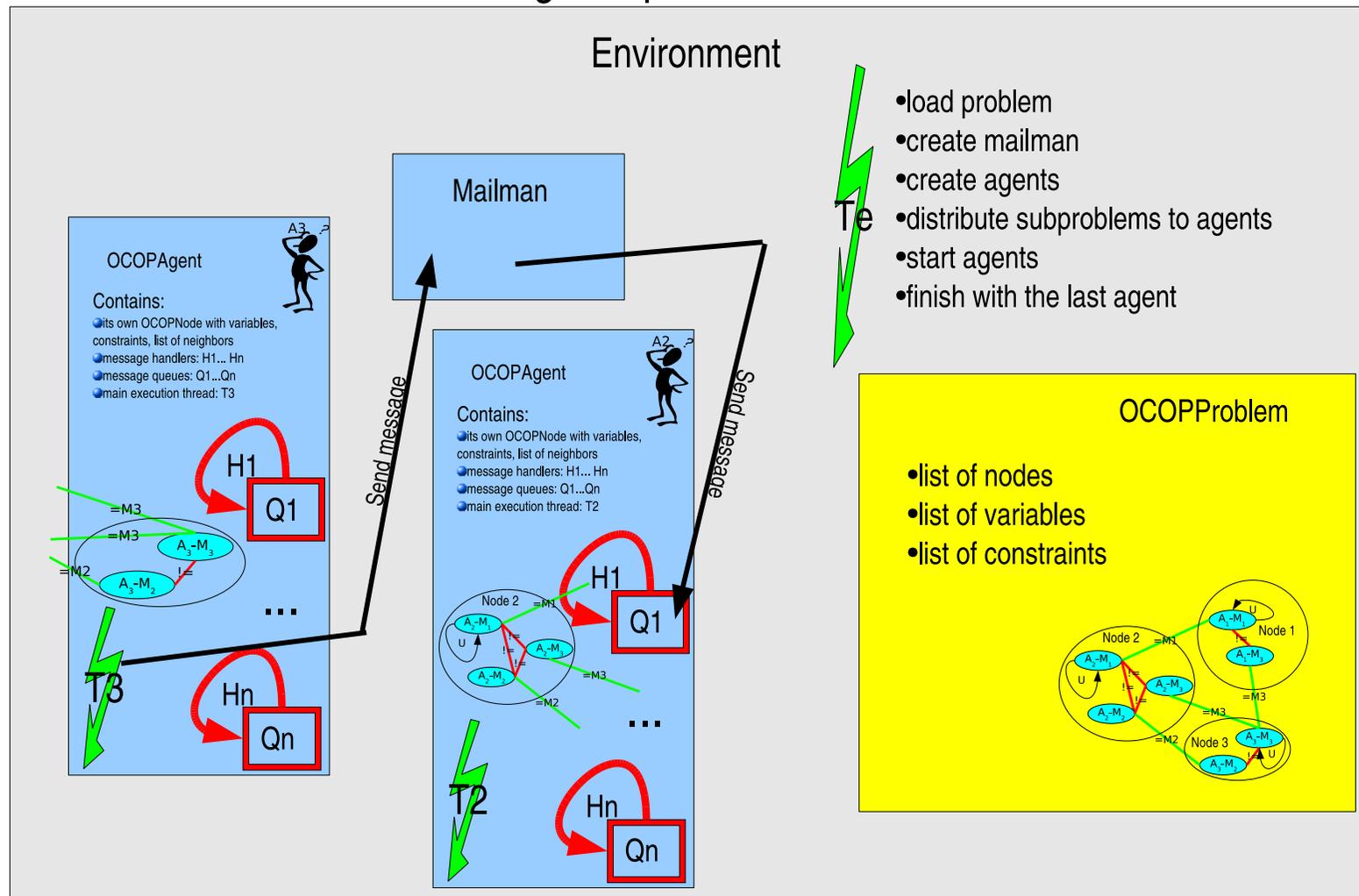


Fig. 1. The overall structure of the multiagent simulation framework.

Figure 1 presents an overview of the structure of the multiagent simulation framework.

### 3.1 Environment

The Environment class provides low level functionality like communication/message passing, loading/distribution of problem/subproblems, etc, much like the equivalent of a multiagent platform like JADE, for instance (however, at a much lower scale).

It is a container for several entities:

- OCOPProblem
- main execution thread  $T_e$
- mailman
- list of agents

The main execution thread  $T_e$  of the environment performs the following tasks:

- loads problem from file
- creates a mailman
- creates agents, one for each node found in the problem
- distributes problem to agents
- starts all agents
- the agents announce the environment when they finish their optimization
- finishes once all agents finished
- announces solution, statistics, etc.

**OCOPProblem - the optimization problem** This object is created at the initialization of the environment. The normal way to create such an object is from a text file that contains the description of the problem, in DPOP format. This is done by calling the static method `OCOPProblem.loadFromFile(fileName)`.

This object contains a complete representation of the optimization problem to be solved: a list of nodes, variables and constraints/relations. It features a number of convenience methods like finding a node/variable by id, finding neighbors of a node, etc.

Please note that while this is a centralized/complete representation of the optimization problem, it is not available to any of the agents. The agents are handed just their own subproblems, without having any global knowledge about the overall problem.

**Mailman - message exchange** This entity serves as a message relay in the system. All agents in the system have unique IDs, known to the mail man.

When an agent  $A_i$  wants to send a message to an agent  $A_j$ , the message exchange between agents is performed as follows:

- $A_i$  is responsible for computing the content of the message (its payload) (presumably taking into account previously received messages)
- $A_i$  builds a *Message* object that contains the serialization of the payload into a string, a sender ID ( $i$ ), a destination ID ( $j$ ) and a message type

- $A_i$  hands this message over to the Mailman
- the mailman inspects the message, finds the ID of the intended recipient, and delivers it into the recipient's message queue that corresponds to the type of the message declared by the sender

### 3.2 OCOPAgent

This entity models an agent from the multiagent system. It has access to the environment and the functionality provided by the environment, as well as to its own optimization subproblem.

An optimization subproblem (OCOPNode) has a localized representation of the neighbors of an agent (local copies of the neighboring variables).

An agent  $OCOPAgent_i$  contains:

- its own  $OCOPNode_i$  with variables, constraints, list of neighbors
- message queues:  $Q_1 \dots Q_n$  (one for each message type)
- message handlers:  $H_1 \dots H_n$  (one for each message queue)
- main execution thread:  $T_i$

The main execution thread  $T_i$  of an agent  $OCOPAgent_i$  performs the following tasks:

- loads its node from the environment
- initializes queues, semaphores, handlers, etc
- sends any initial messages
- waits for messages, and processes them according to algorithm
- builds outgoing messages for (some of) its neighbors and gives them to the mailman

Each agent has a number of message queues and message handlers, one for each type of message that is required by the protocol (the optimization algorithm). The message queues are normal FIFO queues (deliver the messages in order).

Each message handler performs an infinite cycle:

- wait on its respective message queue for incoming messages
- when message received, parse it, construct a corresponding object from the string, and deliver it to the main execution thread of the agent

### 3.3 OCOPNode

This entity models an agent's subproblem - a part from the global optimization problem which belongs to  $Agent_i$ .

An optimization subproblem  $OCOPNode_i$  has a localized representation of the neighbors of an agent (local copies of the neighboring variables).

An agent  $OCOPNode_i$  contains:

- its own variables, constraints, list of neighbors
- convenience methods, like finding/adding/removing variables/constraints, finding neighbors, etc.
- computational logic, algorithm-dependent (e.g. in the case of DPOP, computing the *JOIN* of incoming *UTIL* messages)

## 4 Implementing your own agent system/optimization algorithm

The framework is flexible and powerful enough to allow an experienced user to extend it and implement any distributed optimization algorithm.

We provide an example implementation of DPOP, an optimization algorithm recently introduced in [13]. In the following we will give step-by-step instructions for implementing any optimization algorithm, and analyze the corresponding step from the example implementation of DPOP.

### 4.1 Main class

This is the main class of the program. Its task is to load the environment class, instruct it to load the OCOPProblem from a file, create a distributed representation of the problem, create the agents and populate the environment, and instruct the environment to start the agents.

The DPOP equivalent is `testbed.OCOP.DPOP.TesterDPOP`.

### 4.2 OCOPAgent class

This represents an agent from the system. Normally it should extend `OCOPAbstractAgent`, which provides basic functionality (listing neighbors, sending messages, etc).

The DPOP equivalent is `testbed.OCOP.DPOP.OCOPAgent`.

It performs a initialization sequence, and then the main thread is started.

`initialization()`:

- loads its node from the environment
- initializes queues, semaphores, message handlers, etc

The main execution thread  $T_i$  of an agent  $OCOPAgent_i$  (`run()`):

- sends any initial messages
- waits for notifications of message receipt from the message handlers, and processes the incoming messages according to algorithm
- builds outgoing messages for (some of) its neighbors and gives them to the mailman
- until algorithm is finished;
- then, announce `Environment` that it is finished, and exit

Each agent has a number of message queues and message handlers, one for each type of message that is required by the protocol (the optimization algorithm). For each message type, the agent has a `processMessage` method that is invoked by the corresponding message handler when a new message of that type is received. The logic in these methods, together with the interplay between the various message sending/processing phases are actually the implementation of the optimization algorithm.

In the case of DPOP, there are 3 such methods: `processTOPO()`, `processUTILS()` and `processVALUE()`, corresponding to DPOP's 3 message types (*TOPO*, *UTIL* and *VALUE*). These methods are called by the corresponding message handlers upon receiving/parsing a message and constructing the object corresponding to its payload (`HandleIncomingTOPOMessageBehaviour`, etc).

### 4.3 Message handlers

Examples are `testbed.OCOP.DPOP.behaviours.HandleIncomingTOPOMessageBehaviour`, or `testbed.OCOP.DPOP.behaviours.HandleIncomingUTILMessageBehaviour`, or `testbed.OCOP.DPOP.behaviours.HandleIncomingVALUEMessageBehaviour`.

They extend the `MessageQueueProcessor` class. Their purpose is to be attached to a message queue, and wait until a new message is added to that queue by the `MailMan`. When this happens, they parse the message, building the corresponding object out of the serialization from the message (e.g. a `MessageContext` from a *TOPO* message, or a `HyperCube` from a *UTIL* message). Then, they pass this object to the corresponding method of the agent, which will process it (e.g. `processTOPO`, or `processUTIL`).

## 5 DPOP specific details

The flow of the algorithm is given by the succession of the 3 phases (*TOPO*, *UTIL* and *VALUE*) which is implemented in `OCOPAgent.run()`, and the message passing which is handled by the message handlers (`HandleIncomingTOPOMessageBehaviour`, etc which handle incoming messages), and by the `testbed.communication.MessageSender` object that sends outgoing messages to the `MailMan`.

### 5.1 DFS creation

This is the first phase of the DPOP algorithm.

The node chosen as root initiates this phase by sending a token to one of its neighbors. A token is a *TOPO* message which contains the id's of the nodes that were visited by the token.

This phase is performed by all nodes by calling the `pseudotreeConstruction()` method of `OCOPAgent`. *TOPO* messages are received by `testbed.OCOP.DPOP.behaviours.HandleIncomingTOPOMessageBehaviour` which builds the payload of the message (the `MessageContext` object) out of the serialization, and then calls the `processTOPO` method of the agent.

A node finishes this phase when it has received a token from all its neighbors. Then, it engages in the second phase.

### 5.2 UTIL propagation

This phase is initiated when the previous one finishes. Each node waits for all *UTIL* messages from its children to be received, then computes its outgoing message (see below for details), and then sends it to its parent.

*UTIL* messages are received by `testbed.OCOP.DPOP.behaviours.HandleIncomingUTILMessageBehaviour` which builds the payload of the message (the `HyperCube` object) out of the serialization, and then calls the `processUTIL` method of the agent.

**Hypercubes and generic operations on hypercubes** Modeled by `DPOP.HyperCube`.

We define hypercubes as matrices with 0 or more dimensions. A hypercube without any dimension is just a number, one with 1 dimension is a vector, etc. The mapping between an n-ary constraint and a hypercube with n dimensions is straightforward: the variables involved in the constraint are the dimensions of the hypercube, and the valuation of each tuple of the constraint is recorded in the corresponding cell of the hypercube. We define below a number of operations on hypercubes, which we will use in the algorithms.

**Definition 2.** The  $\oplus$  operator (*join*):  $H = H_1 \oplus H_2$  is the join of two hypercubes. This is also a hypercube, with  $\dim(H_1) \cup \dim(H_2)$  as dimensions. The value of each cell in the join is the sum of the corresponding cells in the two source hypercubes.

**Note:** the  $\oplus$  join operator is associative and commutative. It is implemented in `testbed.OCOP.DPOP.HyperCube.join()`.

**Definition 3.** The (*projection operator*)  $\perp$ : if  $H$  is a hypercube and  $X_k \in \dim(H)$ , then  $H^+ = H \perp_{X_k}$  is the projection through optimization of  $H$  along the  $X_k$  axis: for each tuple of variables in  $\{\dim(H) \setminus X_k\}$ , all the corresponding values from  $H$  (one for each value of  $X_k$ ) are tried, and the best one is chosen. The result is a hypercube with one less dimension ( $X_k$ ).

A projection is essentially a *selection* operation. One eliminates one of the dimensions of a hypercube by selecting from the hypercube a value for each combination of values of the remaining dimensions. This projection has the semantics of a precomputation of the optimal utility that can be achieved with the optimal values of  $X_k$ , for each instantiation of the other variables.

Implemented in `testbed.OCOP.DPOP.HyperCube.project()`.

### 5.3 VALUE propagation

The *VALUE* phase is initiated by the root after receiving all *UTIL* messages. Based on these *UTIL* messages, the root finds its optimal value, and sends a *VALUE* message to its children and pseudochildren.

In addition to its own value, the  $VALUE_j^i$  message a node  $X_j$  sends to its child  $X_i$  also contains the values of all the variables that were present in the context of  $X_i$ 's *UTIL* message for  $X_j$ . E.g.:  $X_0$  sends  $X_2$   $VALUE_0^2(X_0 \leftarrow v_0^*)$ , then  $X_2$  sends  $X_5$   $VALUE_2^5(X_0 \leftarrow v_0^*, X_2 \leftarrow v_2^*)$ , and  $X_5$  sends  $X_{11}$   $VALUE_5^{11}(X_0 \leftarrow v_0^*, X_5 \leftarrow v_5^*)$ . The *VALUE* messages sent to pseudochildren can contain only  $X_j$ 's value.

Upon receipt of the *VALUE* message from its parent, each node is able to pick the optimal value for itself.

After sending its *VALUE* messages,  $X_i$  terminates. This phase is equivalent to the solution reconstruction phase from the bucket elimination scheme (see [1, 2]).

## 6 Experimental evaluation

We experimented with this framework with two problem domains: distributed meeting scheduling problems, and resource allocation in a sensor network.

The sensor allocation experiments were performed with the implementation of the standard Distributed Breakout Algorithm ([17]), and some enhancements based on interchangeability - see [7].

The meeting scheduling experiments were performed with the standard DPOP implementation (see [13]), and with the variants from [14, 8, 11, 12].

## 7 Conclusions and future work

We presented in this paper a framework for distributed optimization, which simulates a multiagent system in a single Java virtual machine. The framework allows all the agents in the system to execute concurrently, and can be extended to implement any distributed optimization algorithm.

Future work should address efficiency issues and possibly a real distribution over the network of the (now centralized) system. Current evaluation metrics (message count, message size) made it uninteresting for the time being to take into consideration issues related to time (execution time, message latency, throughput etc). However, it would be interesting to extend the current framework to deal with more realistic network models that take into account also these issues, and others, like message loss, message corruption, etc.

## References

1. Rina Dechter. Bucket elimination: A unifying framework for processing hard and soft constraints. *Constraints: An International Journal*, 7(2):51–55, 1997.
2. Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
3. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, Number 4598, 13 May 1983, 220, 4598:671–680, 1983.
4. Roger Mailler and Victor Lesser. Solving distributed constraint optimization problems using cooperative mediation. *Proceedings of Third International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2004)*, 2004.
5. P. J. Modi, W. M. Shen, and M. Tambe. An asynchronous complete method for distributed constraint optimization. In *Proc. AAMAS*, 2003.
6. Adrian Petcu. FRODO: a FFramework for Open/Distributed Optimization. <http://liawww.epfl.ch/frodo/>, Jan 2005.
7. Adrian Petcu and Boi Faltings. Applying interchangeability techniques to the distributed breakout algorithm - poster. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence, IJCAI-03*, Acapulco, Mexico, 2003.
8. Adrian Petcu and Boi Faltings. Approximations in distributed optimization. In *CP05 - workshop on Distributed and Speculative Constraint Processing, DSCP*, October 2005.
9. Adrian Petcu and Boi Faltings. An efficient constraint optimization method for large multi-agent systems. In *AAMAS05 - LSMAS workshop*, July 2005.
10. Adrian Petcu and Boi Faltings. Incentive compatible multiagent constraint optimization. In *WINE'05: Workshop on Internet and Network Economics*, Hong Kong, Dec 2005.

11. Adrian Petcu and Boi Faltings. Optimal solution stability in continuous time optimization. In *IJCAI05 - Distributed Constraint Reasoning workshop, DCR05*, August 2005.
12. Adrian Petcu and Boi Faltings. A propagation/local search hybrid for distributed optimization. In *CP 2005- LSCS'05: Second International Workshop on Local Search Techniques in Constraint Satisfaction*, Sitges, Spain, October 2005.
13. Adrian Petcu and Boi Faltings. A scalable method for multiagent constraint optimization. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI-05*, Edinburgh, Scotland, Aug 2005.
14. Adrian Petcu and Boi Faltings. Superstabilizing, fault-containing multiagent combinatorial optimization. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-05*, Pittsburgh, USA, July 2005.
15. Adrian Petcu and Boi Faltings. Distributed generator maintenance scheduling. In *Proceedings of the First International ICSC Symposium on ARTIFICIAL INTELLIGENCE IN ENERGY SYSTEMS AND POWER: AIESP'06*, Madeira, Portugal, Feb 2006.
16. Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *International Conference on Distributed Computing Systems*, pages 614–621, 1992.
17. Makoto Yokoo and Katsutoshi Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In Victor Lesser, editor, *Proceedings of the First International Conference on Multi-Agent Systems*. MIT Press, 1995.
18. Weixiong Zhang and Lars Wittenburg. Distributed breakout algorithm for distributed constraint optimization problems - DBArelax. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS-03)*, Melbourne, Australia, 2003.