The Effects of Agent Synchronization in Asynchronous Search Algorithms

Ionel Muscalagiu[1], Jose M. Vidal[2], Vladimir Cretu[3], Popa Horia Emil[4] and Manuela Panoiu[1]

[1] The "Politehnica" University of Timisoara, The Faculty of Engineering of Hunedoara, Revolutie, 5, Romania,
{mionel,m.panoiu}@.fih.utt.ro

[2] University of South Carolina, Computer Science and Engineering, Columbia SC 29208, USA
vidal@sc.edu

[3] The "Politehnica" University of Timisoara, Computers Science and Engineering Department, Timisoara, V. Parvan 2, Romania,
vcretu@cs.utt.ro

[4] The University of the West, The Faculty of Mathematics and Informatics, Timisoara, V. Parvan 4, Romania
hpopa@info.uvt.ro

Abstract. The asynchronous searching techniques are characterized by the fact that each agent instantiates its variables in a concurrent way. Then, it sends the values of its variables to other agents directly connected to it by using messages. These asynchronous techniques have different behaviors in case of delays in sending messages. This article depicts the opportunity for synchronizing agents' execution in case of asynchronous techniques. It investigates and compares the behaviors of several asynchronous techniques in two cases: agents process the received messages asynchronously (the real situation from practice) and the synchronous case, when a synchronization of the agents' execution is done i.e. the agents perform a computing cycle in which they process a message from a message queue. After that, the synchronization is done by waiting for the other agents to finalize the processing of their messages. The experiments show that the synchronization of the agents' execution leads to lower costs in searching for solution. A solution for synchronizing the agents' execution is proposed for the analyzed asynchronous techniques.

1 Introduction

Constraint programming is a programming approach used to describe and solve large classes of problems such as searching, combinatorial, planning problems, etc. Lately, the AI community has shown increasing interest in the distributed problems that are solvable through modeling by constraints and agents. The idea of sharing various parts of the problem among agents that act independently and collaborate among themselves to find a solution by using messages proves itself useful. It was also lead to the formalized problem known as the Distributed Constraint Satisfaction Problem (DCSP) [4].

There exist complete asynchronous searching techniques for solving the DCSP, such as the ABT (Asynchronous Backtracking) and DisDB (Distributed Dynamic Backtracking) [1, 4]. There is also the AWCS (Asynchronous Weak-Commitment Search) [4] algorithm which records all the nogood values. This technique allows the agents to modify a wrong decision by a dynamic change of the agent priority order. The technique proved beneficial for AWCS. The ABT algorithm has also been generalized by presenting a unifying framework, called ABT kernel [1]. From this kernel two major techniques ABT and DisDB can be obtained.

Asynchronous algorithms are characterized by a message passing mechanism among agents when searching for solution. There are several types of messages used to announce and change the local values attributed to the caretaking of variables. Any practical implementation of these techniques need to manipulate the FIFO channels of messages. These asynchronous techniques have different behaviors in case of delays in sending messages. Which thus leads to different behaviors if we synchronize the execution of the agents.

It is intereseting to examine how these algorithms behave under different synchonization assumptions, as this type of analysis has not been done before. Specifically, it is interesting to investigate the opportunity of synchronizing the agents in case of asynchronous techniques. The behaviors of several asynchronous techniques are investigated in two cases: the agents execute asynchronously the processing of received messages (the real situation from practice) and the synchronous case where the agents' execution is synchronized. In other words, the agents perform a computing cycle in which they process a message from a message queue in the synchronous case. After that, a synchronization is done waiting for the other agents to finalize the processing of their messages. The experiments show that the synchronization of the agents' execution reduces the costs in finding the solution for several families of asynchronous techniques. Two solutions of the agents synchronization of execution are proposed. The first one is based on the existence of a central agent or the access possibility to a common memory zone, solution that allows complete synchronization of the agents. The second one, based on the use of a synchronization message between neighboring agents, allows us to obtain a partial synchronization of the agents.

In this article two families of asynchronous techniques are analyzed: the AWCS and the ABT families. The AWCS family [2, 4] uses a dynamical order for agents. Learning techniques can be applied to this family for building efficient nogoods. This article analyzes a variant of this family improved by applying a nogood learning technique. The second family analyzed is the ABT family [1, 4]. It uses a statical order between agents. The ABT techniques and DisDB are considered from this family. The last technique remarks itself by the fact that it does not require additional links during the search process. The DisDB technique is recommended to be used in real situations in which additional links cannot be added for various external reasons.

2 The Framework

This section presents theoretical considerations regarding the DCSP modeling and the asynchronous techniques[1], [2], [4].

2.1 The Distribution Constraint Satisfaction Problem

Definition 1 (CSP model). *The model based on constraints CSP-Constraint Satisfaction Problem, existing for centralized architectures, consists in:*

-n variables $X_1, X_2, ..., X_n$, whose values are taken from finite, discrete domains $D_1, D_2, ..., D_n$, respectively.

-a set of constraints on their values.

The solution of a CSP supposes to find an association of values to all the variables so that all the constraints should be fulfilled.

Definition 2 (The DCSP model). *A problem of satisfying the distributed constraints ($DCSP$) is a CSP, in which the variables and constraints are distributed among autonomous agents that communicate by transmitting, messages.*

This article considers that each agent A_i has allocated a single variable x_i.

The two families of techniques (the ABT, and AWCS families) are characterized by using many types of messages in the process of agents communication for obtaining the solution. These messages are similar for these families. They are based on asynchronous search principles defined by the ABT technique. Thus, this article presents further on these principles used in ABT and the other techniques presented in here.

In this algorithm, each agent instantiates its variables in a concurrent way and sends the value to the agents with which it is directly connected. The Asynchronous Backtracking algorithm uses 3 types of messages [1],[4]:

– the OK message, which contains an assignment variable-value, is sent by an agent to the constraint-evaluating-agent in order to see if the value is good.
– the nogood message which contains a list (called nogood) with the assignments for which the looseness is found is being sent in case the constraint-evaluating-agent finds an unfulfilled constraint.
– the add-link message, sent to announce the necessity to create a new direct link, caused by a nogood appearance.

ABT requires constraints to be directed. A constraint causes a directed link between the two constrained agents: the value-sending agent, from which the link departs, and the constraint-evaluating agent, to which the link arrives.

Each agent keeps its own agent view and nogood store. Considering a generic agent self, the agent view of self is the set of values that it believes to be assigned to agents connected to self by incoming links. A nogood is a subset of agent view. If a nogood exists, it means the agent can not find a value from the domain consistent with the nogood. When agent X_i finds its agent-view including a nogood, the values of the other agents must be changed. The nogood store keeps nogoods as justifications of inconsistent values. Agents exchange assignments and nogoods. When self makes an assignment, it informs those agents connected to it by outgoing links. Self always accepts new assignments, updating its agent-view accordingly. When self receives a nogood, it is accepted if it is consistent with self's agent view, otherwise it is discarded as obsolete. An accepted nogood is added to self's nogood store. When self cannot take any value consistent with its agent-view , because of the original constraints or because of the received nogoods, new nogoods are generated as inconsistent subsets of the agent-view, and sent to the closest agent involved, causing backtracking. The process terminates when achieving quiescence i.e. a solution has been found, or when the empty nogood is generated i.e. the problem is unsolvable.

2.2 The ABT Family

Starting from the algorithm of ABT, in [1], several derived techniques are suggested, based on this one and known as the ABT family. They differ in the storing process of nogoods, but they all use additional communication links between unconnected agents to detect obsolete information. These techniques are based on a common core (called ABT kernel) hence some of the known techniques can be obtained, by eliminating the old information among the agents.

The ABT kernel algorithm, is a new ABT-based algorithm that does not require to add communication links between initially unconnected agents. The ABT kernel algorithm is sound but may not terminate (the ABT kernel may store obsolete information). In [1] several solutions are suggested to eliminate the old information among agents, solutions that are summarized hereinafter.

A first way to remove obsolete information is to add new communication links to allow a nogood owner to determine whether this nogood is obsolete or not. These added links were proposed in the original ABT algorithm. A second way to remove obsolete information is to detect when a nogood could become obsolete. This solution leads to the DisDB algorithm No new links are added among the agents. To achieve completeness, this algorithm has to remove obsolete information in finite time. To do so, when an agent backtracks forgets all nogoods that hypothetically could become obsolete.

2.3 AWCS Technique

The AWCS algorithm [4], is a hybrid algorithm obtained by combining the ABT algorithm with the WCS algorithm, which exists for CSP. It can be considered as being an improved ABT variant, but not necessarily by reducing the nogood values, but by changing the priority order. The AWCS algorithm uses, like ABT, the two types of ok and nogood messages, with the same significance.

When an agent A_i receives an ok? message, it updates its agent view list and tests if a few nogood values are violated. A generic agent A_i can have the following behavior [4] :

- If no higher priority nogood value is violated, it doesn't do anything.
- If there are a few higher priority nogood values that have inconsistent values and these values could be eliminated by changing the x_i value, the agent will change this value and will send the ok? message.

– If a few higher priority values are inconsistent and this inconsistence can not be eliminated, the agent creates a new nogood messages and sends a nogood message to each agent that has variables in nogood. Than the agent increases the priority of x_i, by changing the x_i value with another value that minimizes the inconsistencies number with all the nogood values and sent the ok? message.

The AWCS algorithm can be improved by applying a learning schema. In [2] there is presented and analyze a schema called "resolvent-based learning", that applies to the AWCS algorithm and brings good results regarding the number of necessary cycles for problem solving. The nogood learning technique induced in [2] is a new method of learning the nogood values applicable to DCSP. The idea is that for each possible value for the failure variable, a nogood that forbids that value is selected and than a new good is built outside the one obtained by unifying the selected nogoods.

3 The synchronization of the agents' execution in case of asynchronous techniques

In [3] are presented two models of implementation and evaluation for the asynchronous techniques in NetLogo. The NetLogo is a programming medium with agents that allows implementing the asynchronous techniques These models are based on two different methods for detecting the ending of the execution of the asynchronous algorithms and are based on the use of the NetLogo environment as a basic simulator for the evaluation of asynchronous search techniques. The two methods from [3] allow the evaluation of asynchronous techniques in the asynchronous case and in the case with synchronization. The model with synchronization is based on NetLogo elements, using the *ask* command for executing the procedures for treating the agents messages. This command does a synchronization of the commands attached to the agents such as the synchronization of the agents' execution is made automatically. Of course, each agent works asynchronously with the messages, but at the end of a command's execution there is a synchronization of agents' execution. Examples of implementation can be found on the web sites [6],[7]. In reality, the agents run concurrently and asynchronous, each agent treating its messages from the messages queue in the arrival order, without expecting for the finalization of computations from the other agents.

The analysis of experimental results shows that the AWCS techniques behave better in case of synchronizing the agents' execution. Starting from this remark, in this paragraph are proposed two solutions of synchronization for the agents' execution.

The first solution proposed is based on using a common memory zone to which the agents have access. In that common memory zone the value of a global variable Nragents, accessible to all the agents is stored. Initially, that variable is initialized with the number of agents. Each agent will mark the status of its execution in the variable Nragents. Practically, each agent A_i decrements the Nragents variable with 1 when the message processing routine is executed. Also, when the agent finishes to process the messages from its message queue it increments the value of Nragents with 1. In other words, the variable Nragents allows the identification of the status of the agents' execution in any moment. At a given moment, that variable can be equal to the number of agents i.e. all agents have finished to process the messages from the message queue. That could become a moment for synchronization, which can be retained by means of a variable called Sincronizare. The first solution allows a complete synchronization of all the agents.

The second solution consists in the synchronizing only the neighboring agents. Each agent will wait for it's connected neighbors to finish their computations, which are placed before him in a lexicographical order. That solution allows a partial synchronization of the agents' execution. That second solution of partial synchronization is based on the use of a synchronization message. This message is similar to a token that each agent needs to receive in order to carry on with the execution of it's computing cycle. For that, each agent uses a second channel of communication for receiving the synchronization messages (the first channel is used for receiving the ok or nogood messages).

The working protocol supposes for each agent the completion of tho stages:

- each agent processes all the messages from it's main communications channel performing a computing cycle. In the moment that the main message channel is empty, it sends a message of the "synchronous" type to the neighboring agents, that are before him in a lexicographical order.

- after each cycle, the agents check if they have received the synchronization messages from all of it's neighbors, placed after him in a lexicographical order, and if not it waits until it receives all those messages.

The implementation of any asynchronous technique implies to build some execution routines for the computations by each agent. In the models proposed in [3] this routine is called update. Also, each agent needs another routine to treat its messages (and here, each technique is different by the fact it treats differently those messages) named in the models from [3] handle-message. These notations are also used in the synchronization solution.

The new procedure update which is performed by each agent is presented in figure 1(a). Each agent verifies if the synchronization status has been reached, in the affirmative case it runs its own message manipulation procedure (handle-message). There are some differences between the two solutions. For the first solution, the agents enter the wait state until the variable Nragents becomes equal to the number of agents. But, for the second synchronization solution, the agents wait until they receive the synchronous message from all it's neighbors placed before it in a lexicographical order.

The procedure that each agent calls to process its messages is depicted in figure 1(b). In that procedure one can notice that the agent decreases the global variable Nragents at the entry point of the procedure. In the end, after the agent has processed a message or many (in packets), the incrementation of Nragents variable is done (first solution). One can see that in order to implement each technique, the agents can process message by message, or in packs by processing all or just partially the messages that exists in the message queue. This article

```
to update
 if Sincronizare
 [ handle-message ]
 if Solution
 [
   WriteSolution
   stop]
 While [synchronization condition]
 [
   wait
   set Sincronizare false
 ]
 set Sincronizare true
end
```
(a) The update procedure

```
to handle-message
1' set Nragents Nragents - 1 // only for the first solution
if not empty? message-queue
 [
    set msg retrieve-message
    Process-message msg
 ]
if (empty? message-queue)
 [ stop ]
1' set Nragents Nragents + 1 //only for the first solution
2' set msg list "sincron" who //only for the second solution
2' send msg to Neighbors //is sent only to the neighbors
    that are before him in a lexicographical order
end
```
(b) The handle-message procedure

Fig. 1. The new procedure

analyzes the behavior of the asynchronous techniques in both variants (treating of one message or complete treating of messages from the message queue of each agent). But, for the second solution, in the routine for message processing, the synchronous message is sent to all it's neighbors placed before him in a lexicographical order (at the end of processing the messages)

The two message manipulation procedures can be applied in any language chosen when implementing. In particular, those two routines can be applied for the asynchronous model proposed in [3], obtaining a synchronization of the agents execution. Thus, starting from the elements from [3] one can obtain a third model of implementation and evaluation for the asynchronous techniques.

4 Experimental results

In order to make such estimation, these techniques are implemented in NetLogo 3.0, a distributed environment, using a special language named NetLogo, [5], [6], [7]. The implementation and evaluation is done using the two models proposed in [3] and the model with synchronization is proposed in this article.

The asynchronous techniques are applied to a classical problem: the problem of coloring a graph in the distributed versions. For this problem we take into consideration two types of problems (we keep in mind the parameters n-number of knots/agents, k-3 colours and m - the number of connections between the agents). We evaluate three types of graphs: graphs with few connections (called sparse problems, having m=n x 2 connections) and graphs with a special number of connections (m=n x 2.3 and m=n x 2.7 connections, called difficult problems). For each version we carried out a number of 100 trials, retaining the average of the measured values (for each class 10 graphs are generated randomly, for each graph being generated 10 initial values, a total of 100 runnings).

In order to make the evaluation of the asynchronous search techniques, the message flow was counted i.e. the quantity of messages ok and nogood exchanged by the agents, the number of verified constraints i.e. the local effort made by each agent, and the number of concurrent constraints verified (noted with c-ccks) necessary for obtaining of the solution.

4.1 AWCS family

In the AWCS family there are many variants that are based on building of efficient nogoods (nogood learning) or on storing and using those nogoods in the process of selecting the values (nogood processor). The basic variant proposed in [4] improved with the nogood learning technique from [2] (noted with AWCS-nl) is applied in this article. A version in which each agent treats entirely the existing messages in its message queue (noted AWCS-nl$_k$) is implemented for this variant. Three implementations are done corresponding to the three obtained models:

– variants based on synchronization with the aid of the "ask" command: AWCS-nl$_1$.
– variants based on the asynchronous model from [3] : AWCS-nl$_2$.
– variant based on the first method of synchronization introduced in this article (complete synchronization)-AWCS-nl$_3$.
– variant based on the second solution of synchronization -AWCS-nl$_4$.

Table 1. The results for AWCS versions (Distributed n-Graph-Coloring Problem)

		n=20			n=30			n=40		
		m=nx2	m=nx2.3	m=nx2.7	m=nx2	m=nx2.3	m=nx2.7	m=nx2	m=nx2.3	m=nx2.7
AWCS-nl$_1$	Nogood	126.66	261.58	902.57	575.84	2052.00	3639.52	713.60	4857.77	24652.61
	Ok	470.48	745.27	1918.56	1864.50	5135.07	7624.27	2403.25	12733.40	51132.39
	Constr.	856.43	1474.91	4475.59	2913.94	9252.07	14482.23	3747.34	20371.70	102388.82
	c-ccks	307.12	528.49	1608.16	802.74	2414.97	4351.37	824.10	4422.60	21888.16
AWCS-nl$_2$	Nogood	129.86	217.62	1412.17	502.77	3407.93	12181.62	733.93	6877.65	93259.95
	Ok	456.06	633.52	2881.41	1625.23	8725.69	23964.88	2422.19	17612.68	185845.00
	Constr.	900.47	1334.05	6659.52	2628.08	15309.45	45813.89	4163.19	27261.72	387355.71
	c-ccks	294.81	442.89	2243.28	662.13	3625.55	12512.22	793.45	5409.08	73099.00
AWCS-nl$_3$	Nogood	137.31	244.91	1137.93	534.27	2840.31	3439.96	755.42	5302.25	22394.24
	Ok	493.51	710.35	2374.80	1738.56	6978.90	7129.70	2512.65	13643.28	47181.16
	Constr.	894.40	1413.90	5507.58	2618.45	12559.69	13858.95	3943.26	22631.85	93429.74
	c-ccks	319.11	506.20	1992.89	759.39	3300.41	4109.20	865.93	4749.67	20065.21
AWCS-nl$_4$	Nogood	133.24	227.05	1211.12	567.23	1423.10	3672.32	723.15	2820.10	23371.67
	Ok	471.54	662.94	2417.56	1799.98	3776.31	7889.14	2579.42	7604.10	50083.72
	Constr.	931.18	1354.89	5863.72	2751.92	7031.34	15219.11	4081.18	12692.65	109657.25
	c-ccks	328.38	478.78	2101.82	795.28	1842.69	4310.87	876.12	2837.00	21011.11

As known, the verified constraints quantity evaluates the local effort given by each agent, but the number of concurrent constraint checks allows the evaluation of this effort without considering that the agents work concurrently (informally, the number of concurrent constraint checks approximates the longest sequence of constraint checks not performed concurrently). Analyzing the results from table 1, we can remark that synchronization of the agents execution reduces the local effort made by the agents, regardless of the variant used (that based on the ask command or the general one introduced in this article). But, as the dimension of the problems increases (40 nods), the asynchronous variants AWCS-nl$_2$ required much greater efforts compared to the variants with synchronization. Big differences in the local effort occur especially for the problems of high density (problems known as difficult problems). Still one can remark that for problems of scarce density (m= n x 2.0) the asynchronous variants have almost equal costs to the synchronous variants, even lower efforts.

In the case of the message flow, the behavior remarked regarding the computing effort has maintained almost the same, synchronous variants requiring a message flow lower that the asynchronous variants. The message flow increased for the synchronous variants together with the increase of dimension for the solved problems. Comparing the two synchronous variants, one can remark almost equal efforts for obtaining the solution, such as the practical solution proposed in this article require almost equal efforts to those used in the variant simulated with the ask command.

4.2 The ABT Family

Starting from the ABT kernel, by eliminating the outdated information two important techniques are obtained: the Asynchronous Backtracking and the Distributed Dynamic Backtracking [1]. These two techniques based on a static order are analyzed in order to see the effect of agents synchronization. Unlike the AWCS technique, this article depicts the versions in which each agent treats at each cycle just one message from its message queue (noted with ABT_k and DisDB_k). For each of the ABT and DisDB techniques three implementations are done corresponding to the three obtained models:

- variants based on synchronization by means of the "ask" command: ABT_1 and DisDB_1.
- variants based on the asynchronous model from [3]: ABT_2 and DisDB_2.
- variants based on the method of synchronization introduced in this article : $\mathrm{ABT}_{3,4}$ and $\mathrm{DisDB}_{3,4}$.

(a) ABT

		n=20		n=30	
		m=nx2	m=nx2.7	m=nx2	m=nx2.7
A_1	Nogood	359.70	481.88.74	3883.12	3936.20
	Ok	1214.93	1615.48	8578.23	12343.88
	Constr.	67500.30	80834.37	957934.98	938790.08
	c-ccks	14502.26	16572.45	147128.29	123501.58
A_2	Nogood	305.70	366.94	3181.57	2196.52
	Ok	1124.16	1438.48	6518.74	9943.55
	Constr.	60382.63	69332.91	763992.16	698240.00
	c-ccks	8993.06	7593.00	90601.22	45264.01
A_3	Nogood	360.41	376.99	3713.65	3764.15
	Ok	1237.79	1495.00	8065.23	10209.04
	Constr.	68321.27	76823.12	85234.16	800193.03
	c-ccks	13401.26	12312.34	112675.05	89000.45
A_4	Nogood	371.12	395.18	3945.18	3799.55
	Ok	1323.11	1578.06	8245.32	11311.71
	Constr.	67001.81	75121.12	83555.23	753712.20
	c-ccks	12182.11	11133.56	109129.82	90101.34

(b) DisDB

		n=20		n=30	
		m=nx2	m=nx2.7	m=nx2	m=nx2.7
D_1	Nogood	138.76	316.94	2803.34	3104.44
	Ok	300.04	722.89	6855.19	7235.88
	Constr.	18548.68	40841.66	603288.94	778016.08
	c-ccks	4526.52	8112.34	124728.67	154409.43
D_2	Nogood	83.10	216.41	2785.65	2983.69
	Ok	213.72	556.03	6518.74	6878.05
	Constr.	12117.12	29850.88	585992.16	655243.39
	c-ccks	2468.48	3963.02	70605.97	82785.79
D_3	Nogood	125.38	271.25	2813.03	3016.96
	Ok	277.64	646.82	7005.56	7029.70
	Constr.	16837.14	35589.37	603192.16	712024.78
	c-ccks	3791.24	6289.09	115059.67	90786.78
D_4	Nogood	117.21	273.67	2767.28	2994.39
	Ok	272.40	651.71	6981.90	7002.81
	Constr.	15002.91	36111.52	523023.73	679821.34
	c-ccks	34230.74	6421.90	102011.72	88564.20

Table 2. The results for ABT and DisDB versions

In table 2(a) are presented the values obtained for the ABT versions, and in table 2(b) those for the versions DisDB. The two techniques based on a static order behave different from the AWCS technique. The lower local effort is carried out for the asynchronous variant, the variants with synchronization require the checking of a much greater number of constraints. Also, the lowest message flow is obtained for the synchronous variants. The two techniques behave the same for both type of problems: problems with scarce density or difficult problems.

The synchronization solution proposed in this article is superior to that offered by the ask command from NetLogo. Unfortunately the techniques from din ABT family, regardless of the fact that they require or not adding extra links, behave better in the asynchronous case, the synchronization is not a solution for reducing the costs.

5 Conclusions

This article remarks two types of behaviors and, as a consequence, two classes of asynchronous techniques (among the ones evaluated in here).

The techniques from the AWCS family, based on a dynamic order for the agents, require lower costs for obtaining the solution in case of synchronization of the agents' execution. A synchronization solution is proposed in this article. The experiments show a decrease of local computing effort and of message flow compared to the asynchronous variants.

The second category of techniques, i.e. from the ABT family behave different, requiring lower costs in the asynchronous case that in case of the synchronization of the agents' execution. All these techniques use a static order for the agents.

After the analysis of these empirical studies, we deduce that the synchronization of the agents' execution is recommended for the techniques with dynamical order.

References

1. Bessiere, C., Brito, I., Maestre, A., Meseguer, P. Asynchronous Backtracking without Adding Links: A New Member in the ABT Family. A.I., **161** (2005) 7–24.
2. Hirayama, K., Yokoo, M. The Effect of Nogood Learning in Distributed Constraint Satisfaction. In Proceedings of the 20th IEEE International Conference on Distributed Computing Systems, (2000) 169–177.
3. Muscalagiu, I., Jiang, H., Popa, H. E. Implementation and evaluation model for the asynchronous techniques: from a synchronously distributed system to a asynchronous distributed system. Proceedings of the 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2006), Timisoara, Romania, IEEE Computer Society Press, (2006) 209–216.
4. Yokoo, M., Durfee, E. H., Ishida, T., Kuwabara, K. The distributed constraint satisfaction problem: formalization and algorithms. IEEE Transactions on Knowledge and Data Engineering **10(5)** (1980) 673–685.
5. Wilensky, U.NetLogo itself:NetLogo. Available: http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Evanston, (1999).
6. MAS Netlogo Models-a. Available: http://jmvidal.cse.sc.edu/netlogomas/.
7. MAS Netlogo Models-b. Available: http://ccl.northwestern.edu/netlogo/models/community.