INFORMATICA, 2006, Vol. 17, No. 1, 39–54

Improving the Performances of Asynchronous Algorithms by Combining the Nogood Processors with the Nogood Learning Techniques

Ionel MUSCALAGIU
The Faculty of Engineering of Hunedoara, The Politehnica University of Timisoara
Hunedoara, str. Revolutiei, nr. 5, Romania
e-mail: mionel@fih.utt.ro

Vladimir CRETU
Computers Science and Engineering Department Timisoara
The Politehnica University of Timisoara
Timisoara, str. V. Parvan, nr. 2, Romania
e-mail: vcretu@cs.utt.ro

Received: January 2005

Abstract. The asynchronous techniques that exist within the programming with distributed constraints are characterized by the occurrence of the nogood values during the search for the solution. The nogood type messages are sent among the agents with the purpose of realizing an intelligent backtrack and of ensuring the algorithm's completion.

In this article we analyzed the way in which a technique of obtaining efficient nogood values could combine with a technique of storing these values. In other words we try combining the resolvent-based learning technique introduced by Yokoo with the nogood processor technique in the case of asynchrounous weak-commitment search algorithm (AWCS). These techniques refer to the possibility of obtaining efficient nogoods, respectively to the way the nogood values are stored and the later use of information given by the nogoods in the process of selecting a new value for the variables associated to agents. Starting from this analysis we proposed certain modifications for the two known techniques.

We analyzed the situations in which the nogoods are distributed to more nogood processors handed by certain agents. We proposed a solution of distributing the nogood processors to the agents regarding the agents' order, with the purpose of reducing the storing and searching costs. We also analyzed the benefits the combining of nogood processor technique with the resolved-based learning technique could bring to the enhancement of the performance of AWCS technique. Finally, we analyzed the behavior of the techniques obtained in the case of messages filtering.

Key words: artificial intelligence, distributed programming, constraints, agents, nogood messages.

1. Introduction and Problem Statement

The constraint programming is a model of the software technologies, used to describe and solve large classes of problems as, for instance, searching problems, combinatorial

problems, planning problems, etc. A large variety of problems in the Artificial Intelligence (AI) field and other domains specific to computer sciences could be regarded as a special case of constraint programming. Lately, the AI community has shown great interest towards 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 that collaborate among themselves using messages, in the prospective of gaining the solution, proved itself useful, as it conducted to obtaining a new modeling type called Distributed Constraint Satisfaction Problem (DCSP) (Yokoo, 1998; Yokoo, 2001).

There are more complete or incomplete asynchronous searching techniques available, for the DCSP modeling (Armstrong, 1997; Bessière, 2000; Silaghi, 2000; Yokoo, 1998) which allow the solving of a problem in this constraint network. These techniques are remarkable because of the diversity of applied ideas concerning the agents' work in an asynchronous and competitive way, but they also ensure the completeness or a better efficiency, too.

In the distributed constraint satisfaction area, the asynchrounous weak-commitment search algorithm (AWCS) plays a fundamental and pioneer role among algorithms for solving the distributed CSPs. The algorithm is remarked for the suffering of an explosion of the nogood values, but, by dynamically changing the agents' order, it is an efficient algorithm because of its number of cycles. There are several studies having the purpose of enhancing the efficiency of this technique, studies that focus on enhancing the efficiency of this technique, and on minimizing and selecting the nogood values.

The occurrence of nogood values has the effect of inducing new constraints. Nogood values show the cause of failure and their incorporation as a new constraint will teach the agents not to repeat the same mistake. As it is known, the constraints restrict internally or externally the behavior of an agent, therefore the study of the constraints could offer ways of growing the efficiency of the existent algorithms within the DCSP. As the nogoods are, as a matter of fact, a type of dynamically generated constraints during the searching process, which in certain situations grow explosively, it remains to be studied the way of constructing and selecting the nogood values.

The non-restriction for recording the nogood values could become, in certain cases, impracticable. The main reason is that the storing of nogood values excessively consumes memory and could lead to lowering the memory that has been left. Typically, the number of nogood values grows along with the number of conflicts – in the worst case this growth could be exponential in the number of variables. Another unpleasant effect of storing a large number of nogood values is related to the fact that the verification of the current associations in the list of nogood values that are stored becomes very expensive, the searching effort removing the benefits brought by storing the nogood values. These elements are analyzed as targeting to see if this nogood processor technique brings benefits in terms of efficiency.

In (Armstrong, 1997), Armstrong builds a new asynchronous technique in which the problem's solution is divided into epochs. There is a central agent responsible for the start of the searching process and a nogood processor that keeps informations on the

nogoods occurred.When a nogood is discovered (Armstrong, 1997), the variable assignments causing it and the IDs of the agents involved are sent to the no-good processor, which saves the nogood. Later, when an agent has found a tentative assignment for its variables, it consults the nogood processor to make sure that its assignment along with any known assignments of higher priority agents do not constitute a nogood.

In (Hirayama, 2000), Hirayama and Yokoo introduce the term of nogood learning or leaning that refers to obtaining and registering nogood values. In (Hirayama, 2000) it is presented and analyzed a schematic called "resolvent-based learning", that applies to the AWCS algorithm and brings good results regarding the number of necessary cycles for problem solving, even better that the DB (Distributed Breakout) algorithm which is considered the most efficient in the position of cycles number. The technique is based on the constraint of a new effective nogood just by verifying a few nogood values.

In this paper it is presented a way of distributing the nogood processors and more solution of increasing the efficiency of the AWCS technique are identified. We try to adapt the nogood processor technique for the AWCS technique. This technique consists in storing the nogood values and further use the information given by nogoods in the process of selecting a new value for the variables associated to agents. In this paper it is analyzed the distribution of nogood values to agents and the way to use the information stored in the nogood, what we will call the nogood processor technique. We proposed a solution of distributing nogood processors by the agents in accordance with the order of agents, with the purpose of reducing the searching and storing costs.

This study tries to combine the versions of nogood processor with the learning techniques with the purpose of finding a solution of increasing the performances of the AWCS technique. Starting with this analysis we is proposed certain modifications for the two known techniques. We analyzed the situations where the nogoods are distributed to more nogood processors handed by certain agents. We analyzed the benefits that the combination of the nogood processor technique with the "resolvent-based learning" technique could bring to enhancing the performances of the AWCS technique through a set of experiments. More, we analyzed the behavior of the obtained techniques in the case of message filtering.

2. The Framework

In order to do this analysis to the way the two storing and building nogood values techniques could be combined, in this paragraph we will present some notions known from the IT literature related to the DCSP modeling and mostly related to the asynchronous technique, AWCS (Yokoo, 1998; Yokoo, 2001).

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 $D_1, D_2, \ldots, D_n$, 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.

DEFINITION 3 (the assignment). A pair (X_i, v) is called assignment for the variable X_i, where v is a value from the X_i domain.

DEFINITION 4 (the list agent-view). The list agent-view of an agent A_i is a set with the newest assignments received by the A_i agent for distinct variables.

DEFINITION 5 (the nogood list). The Nogood list is a set of assignments for distinctive variables for which looseness was found.

Asynchronous algorithms are characterized by the agents using the messages during the process of solution searching. Typically, we meet more types of messages, used primarily to announce and change the local values attributed to the caretaking of variables. In this category we meet ok? messages, respectively nogood messages or back messages, used to announce the appearance of an inconsistence.

The AWCS algorithm (Yokoo, 1998) is a hybrid algorithm obtained by the combination of the asynchronous backtracking algorithm (ABT) with the weak-commitment search algorithm (WCS), 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. It deliberately follows to record all the nogood values (which are fewer) to ensure the completeness of the algorithm, but also the avoidance of some unstable situations.

The authors show in (Yokoo, 1998) that this new algorithm can be built by a dynamical change of the priority order. The AWCS algorithm uses, like ABT, the two types of ok and nogood messages, with the same significance. There is a major difference in the way you treat the ok message. In case of receiving the ok message, if the agent can't find a value to its variable that should be consistent with the values of the variables that have a greater priority, the agent not only creates and sends the nogood message, but also increases the priority in order to be maximum among the neighbors.

We further present in Fig. 1 the treatment procedures for the messages existent in (Yokoo, 1998).

For a better understanding of the way the nogood processor technique applies, we will present more information related to the behavior of an agent A_i, for the AWCS algorithm (Yokoo, 1998). When an agent A_i receives an ok? message, it updates its agent view list and tests if a few nogood values are violated (Fig. 1, procedure check-agent-view). A very important thing is connected to agent testing only the nogood values that have a greater

```
when received (ok?, (x_j,d_j, priority)) do
    add (x_j,d_j, priority) to agent-view;
    check-agent-view;
end do

when received (nogood, (x_j, nogood)) do
    add nogood to nogood-list;
    when (x_k,d_k, priority), where x_k is not in neighbors is contained in nogood do
        add x_k to neighbors
        add (x_k,d_k, priority) to agent-view;
    end do
    check-agent-view;
end do

procedure check-agent-view
    when agent-view and current-value are not consistent do
        if no value in D_i is consistent with agent-view then
            backtrack
        else
            select d ∈ D_i where agent-view and d minimizes the number of constraint
            violations with lower priority agents;                    **
            current-value ⟵ d
            send (ok?, (x_i,d, priority))to neighbors
        end if
    end do
end procedure

procedure Backtrack
    nogoods ⟵V/ V= inconsistent subset of agent-view
    when an empty set is an element of nogoods do
        broadcast to other agents that there is no solution,terminate this algorithm;
    end do
    when no element of nogoods is included noogod-sent do
        for each V ∈ nogoods do
            add V to nogood-sent
            for each (x_j,d_j,p_j) in V
                send (nogood, x_i, V ) to x_j
            end do
        end do
        P_max ⟵ max_{(x_j,d_j,p_j)∈agent−view}(p_j)
        current-priority ⟵ 1 + P_max
        select d ∈ D_i where agent-view and d minimizes the number of constraint
        violations with lower priority agents;                    **
        current-value ⟵ d
        send (ok?, (x_i,d, priority))to neighbors
    end do
end procedure
```

Fig. 1. The asynchronous algorithm weak-commitment search (procedures for receiving messages).

priority than x_i. As a matter of fact the priority of a nogood value is defined as the lowest priority of the nogood variables, excepting x_i. As a conclusion, a generic agent A_i can have the following behavior:

- 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 (Fig. 1, procedure check-agent-view). If it has to choose between more values, it will select that value that minimizes the inconsistencies in the inferior priority nogood values (a nogood value of inferior priority is the one in which its priority is inferior to the x_i priority).
- If a few higher priority values are inconsistent and this inconsistence can not be eliminated, the agent creates a new nogood messages outside the agent view list and sends a nogood message to each agent that has variables in nogood (Fig. 1, procedure Backtrack). 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. If the new nogood is identical to the previous nogood value, than the agent will do nothing. This step is necessary for assuring the completeness of the algorithm (Yokoo, 1998).

We must underline a certain behavior of A_i agent, specific to the AWCS algorithm: when a nogood message is received, the agent adds the nogood value to its nogood group and executes a verification of the inconsistencies for nogood. If the new nogood has also an unknown variable, the agent needs to receive from the corresponding agent the value of the variable that's been cared for. Unfortunately, the information from nogood is not completely used. It is about the case of attributing a new value for the variable associated to the agent. It is possible that the nogood values contain a reference to this value, that implying the attribution to have appeared more as inconsistent. The use of this information will be the basis of the nogood processor technique construction.

3. The Technique "Resolvent-Based Learning"

The CSP modeling offers more solving techniques based on searching. These can be enhanced by applying the loop-back techniques. They rely on using the information on the realized search.

The nogood learning technique induced in (Hirayama, 2000) is a new method of learning the nogood values applicable to DCSP, based on adapting the loop-back techniques to the DCSP frame. The idea is that for each possible value for the failure variable, we select a nogood that forbids that value and than a new good is built outside the one obtained by unifying the selected nogoods. The authors of that method compare this nogood as resembling to the notion of resolvent from the propositional logic, hence the denomination of this method: resolvent based on learning.

In order to better explain the way of functioning of this method (Yokoo, 1998) we will consider an agent A_i, with the D_i domain and each value from that domain enters a

conflict with certain nogood values of high priority for the current agent view list. This method brings the following behavior for the A_i agent – this agent selects a nogood value for each d $\in D_i$ value, so:

- The agent identifies those nogoods of high priority inconsistent under the current agent view list, and we have $x_i =$ d.
- Then, we selected the smallest of these nogoods.

Finally, the agent A_i builds a new nogood by erasing all the elements including x_i from the multitude of selected nogoods. The agent will select the lowest nogood, regarded from the priorities' prospective because it is desired that a new as small as possible nogood be obtained.

4. Adapting the "Nogood Processor" Technique in the Case of AWCS Technique

In this paragraph we will present a way of distributing and applying the nogood processor technique for the AWCS technique. We will further present the main ideas that served to adapt the nogood processor technique for AWCS.

The nogood processor technique was introduced by Armstrong, in (Armstrong, 1997), applied to a new technique in which the problem's solution was divided to eras (epochs). When a nogood is discovered (Armstrong, 1997), the variable assignments causing it and the IDs of the agents involved are sent to the nogood processor, which saves the nogood. Later, when an agent has found a tentative assignment for its variables, it consults the nogood processor to make sure that its assignment along with any known assignments of higher priority agents does not constitute a nogood.

In asynchronous weak-commitment search, the agents rediscover and store many copies of the same nogood, one for each time a different participating agent in the nogood had lowest priority. A nogood processor gives us the benefit of reduced storage costs (only one copy) and reduced search time (only discovered once). The tradeoff is in additional message passing. It could distribute the nogood processor to reduce the computation and storage load on any particular process. In this case when an agent wants to check a collection of instantiations for nogoods, it sends the set of agents involved (and the variable assignments) to the nogood processors. The nogood processors are each responsible for mutually exclusive partitions of the power set of the agents. Each processor checks all subsets of the current list of agents corresponding to subsets in its piece of the partition.

The adaptation and the application of the nogood processor technique for the AWCS technique lead to the identification of some answers to the problems occurred: how do we store the nogood values and where, how is the storing and evaluation of nogood values distributed? Another very important problem was to see if all agents (or a part of them) appeal the nogood processor, so as there will be no chance that the costs necessary for evaluating the stored nogoods would outrun the benefits brought by the nogood processor technique.

In this paper it is considered that each agent has access to the results of its own nogood processor or of a centralized nogood processor (typically using messages). More, each

agent sends (stores) nogood values that it has received to the associated nogood processor. Therefore, when using a single processor, the treating procedure for nogood messages will have to store in a shared memory zone or to send the nogood values to the nogood processor (these nogood values are stored in the nogoods-store list).

The information stored by each nogood processor will be used in searching a new value for each variable cared for by the agent. For this, each nogood processor will verify (asked by an agent) through its subroutine check-inconsistent-value-nogood-processor, if the value selected by the agent had no previous existence associated with the higher priority agents values. In Fig. 2 we have the checking routine for the inconsistence of a new value. The appeal for the routine (check-inconsistent-value-nogood-processor) is marked with ** in the AWCS algorithm presented in Fig. 1.

Another question needing an answer was identifying the distribution way of nogood processors. In this paper we presented more ways of distribution for the nogood values processors. First solution was adapting the AWCS technique so as to store in a centralized way the nogood values (marked with AWCSnp_2, basic AWCS technique will be called AWCS$_1$). Practically, each agent, when receiving a nogood, sends this to a centralized nogood processor that stores it into a nogood-store list (nogood processor only saves those new nogood values, eliminating the copies). The information will be used later when searching for a new value. Therefore the check-agent-view procedure will select a new value consistent with the agent view list and with the nogood list stored by the nogood processor (the value will be selected if, complementarily, the check-inconsistent-value-nogood-processor subroutine will return the consistent value).

The second AWCS version (called AWCSnp_4), was obtained by modifying the check-inconsistent-value-nogood-processor routine so as to verify just for the higher priority agents, priority it had in the moment of nogood value storing. To put it differently, the identification of the agents with higher priority towards agent A_i, is not made by using their actual priority (in current-view), but relative to the priority stored by the nogood

```
function check-inconsistent-value-nogood-processor [A_i]
    foreach Nogood ∈ nogoods-store do
        foreach x ∈ Nogood with the current priority from current-view
                * bigger than the agent's A_i
          pos ⟵ position x in Nogood
          if x != item pos current-value
              return consistent
           endif
        end do
        if current-value != item A_i in nogood
            return consistent
        endif
        end do
       return inconsistent
end procedure
```

Fig. 2. The Procedure check-inconsistent-value-nogood-processor.

processor. The correspondent modification was noted with * in the verification routine. We took into consideration the case of a sole centralized nogood processor.

The following step was distributing the nogood values to more nogood processors, one for each agent. Practically, when receiving a nogood value, it is stored only by the associated nogood processor. The verification was made equally, obtaining other 2 versions (noted with AWCSnp_3 and AWCSnp_5), the last one uses to identify higher priority agents by using the old priorities).

The following versions calculated were based on identifying the agents that will appeal the associated nogood processors. A first version was obtained by adapting the AWCSnp_2 version so as its verification to be made by all agents, excepting the higher priority one among the neighbors at that moment (that version it noted with AWCSnp_6).

The last versions (noted with AWCSnp_7 and AWCSnp_8), were obtained starting from the AWCSnp_2 and AWCSnp_4 versions, but the verification having been made only by the agent with the higher priority among the neighbors (current priority in current-view, respectively the old one stored). In other words, the 2 other versions of derivate techniques were obtained by deriving the centralized nogood processor versions, in which the evaluation is made only by an agent of higher priority (maximal priority) among the neighbors.

5. Combining the Nogood Processor and Nogood Learning Techniques

The combination of the 2 techniques was not made directly. It was necessary to adapt them for obtaining a derivate technique complete and efficient.

As previously presented, the nogood learning technique intervenes when an agent A_i (with the D_i domain) for each value of its domain it conflicts with a few nogood values of higher priority for the current agent view list. As a matter of fact this is the backtrack situation, in which we apply the Backtracking routine. Inducing the nogood processor technique has the effect of using the information stored by each nogood processor in the process of searching a new value for each variable taken care by the agent. For this each nogood processor will verify when asked by an agent, by the subroutine check-inconsistent-value-nogood-processor, if the value selected by the agent hasn't previously existed combined with the values of the agents of higher priority. Therefore it is possible that when asked by the construction routine for a new nogood, for a certain value there will not be any high priority nogood that would be inconsistent under the current agent view list and so we will have $x_i = \mathrm{d}$. Therefore we will restart the process of building the nogood values, by selecting the current agent view list for the nogood.

From the nogood processor versions proposed in previous paragraph, we chose the distributed versions in which each agent has its own nogood processor called when selecting a new value(noted with AWCSnp_3 and AWCSnp_5 in previous paragraph). We proposed 2 versions, AWCS$_3$ and AWCS$_4$ that would combine the nogood learning technique with the 2 versions of nogood processor. The AWCS$_4$ version is different from AWCS$_3$ because its verification is being made only by the agent with the higher priority among the neighbors. Another observation related to the versions from previous paragraph is that there will be no more supplementary applying to the nogood processors when receiving an ok message.

In the experimenting paragraph, $AWCS_1$ will be considered as being the basic technique, and $AWCS_2$ the version with resolvent-based learning.

6. Experimental Results

6.1. *Introduction*

In this paragraph we will present our experimental results, obtained by implementing and evaluating the asynchronous techniques we introduced. In order to make such estimation, we implemented these techniques in NetLogo 2.0, a distributed environment, using a special language named NetLogo (see (Wilensky, 1999; MAS Netlogo Models)).

The asynchronous techniques were applied to a classical problem: the problem of colouring a graph in the distributed versions. For the problem of graph colouring we took into consideration two types of problems defined as in (Minton, 1992). (We kept in mind the parameters n – number of knots/agents, $k-3$ colours and m – the number of connections between the agents). We evaluated two types of graphs: graphs with few connections (called sparse problems, having $m = n \times 2$ connections) and graphs with a special number of connections, known to be difficult problems (called difficult problems and having $m = n \times 2.7$ connections). 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).

We counted the number of messages (which means the quantity of ok and nogood messages), the number of constraint checks and the number of cycles necessary for obtaining each solution. We also kept in mind the number of stored nogood values number along with the number of comparisons made by the nogood processors. The evaluations have been made for each technique presented.

The evaluations had certain particularities due to the NetLogo medium. The NetLogo medium is a programming medium with agents that allows implementing the asynchronous techniques (Wilensky, 1999; MAS Netlogo Models), but has certain particularities related to asynchronous work with agents. The agents work with the specific command "ask". A command like this will allow launching the work routines with the messages. 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, synchronization that particularizes, in a way, the implementations in use. This type of agents work resembles the one used in (Hirayama, 2000) at the evaluation of AWCS algorithm together with resolvent-based learning.

6.2. *The Analysis of the Experimental Results in the Case of the Obtained Techniques*

In Fig. 3, Fig. 4 and Fig. 5 we presented graphically the experimental results for the 4 versions of AWCS, with regard to the number of cycles, constraints and messages stream.

The first measuring unit for the analyzed asynchronous techniques performances was the cycle. This unit allows evaluating the calculation of the global effort evaluation for

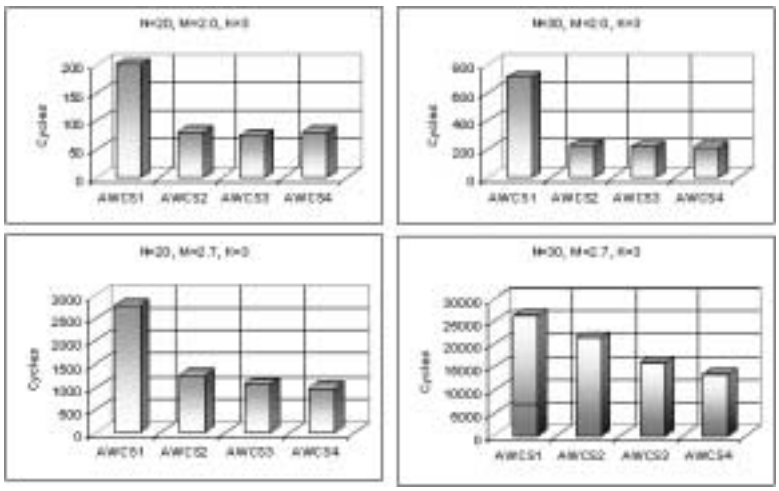

Fig. 3. Comparative study for AWCS versions (Distributed n-Graph-Coloring Problem) – cycles

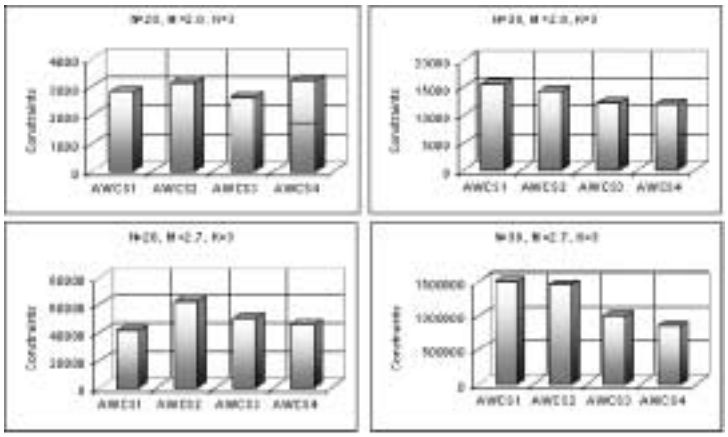

Fig. 4. Comparative study for AWCS versions (Distributed n-Graph-Coloring Problem) – constraints checks.

a certain technique. By analyzing the graphics from the Fig. 3, where we represented graphically the number of cycles necessary for obtaining the solution, one can notice good performances for the versions obtained, AWCS_3 and AWCS_4, comparatively to the basic version cu AWCS and the nogood learning version. The two versions had a smaller number of cycles, compared to the other versions, for both problem classes: sparse and difficult problems. Still it has to be remarked the most difficult problem ($n = 30$, $m = 2.7$), the best behavior the AWCS_4 version had, when the manipulation of the nogood processor was made only by the agent with the highest priority among the neighbors.

As known, the verified constraints quantity evaluates the local effort given by each agent. In Fig. 4 we presented the comparative results for the 4 versions, results obtained by comparing the number of verified constraints. Regarding the calculating effort, we

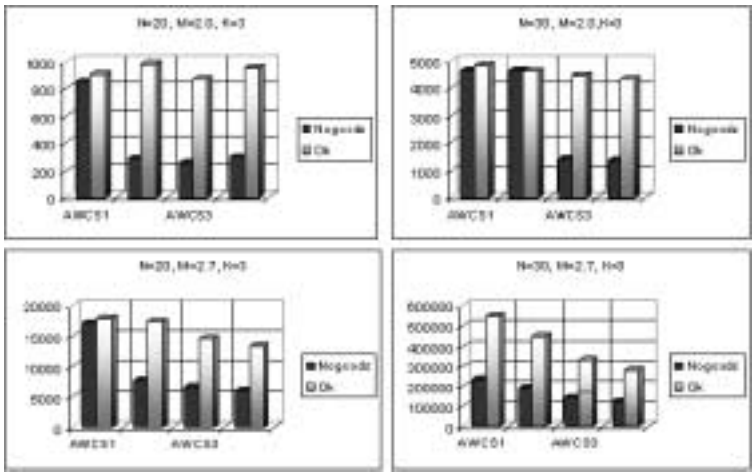

Fig. 5. Comparative study for AWCS versions (Distributed n-Graph-Coloring Problem) – nogood and ok messages.

observe that the proposed versions, including the nogood learning version needed a larger number of constraints. As a matter of fact the evaluated techniques assumed a far larger local calculating effort but the number of cycles necessary for obtaining the solution was reduced.

In the last graph (Fig. 5) we presented the experimental results and the comparative study for the 4 versions, but with respect to the message stream. We counted the number of messages (which means the quantity of ok and nogood messages). It must be stressed that we counted only the nogood messages changed by agents, and we didn't calculate the messages necessary to send the nogood values to the nogood processors.

6.3. *Analysis of the Result of the Experiments in the Case of Message Filtering*

By analyzing the message queues for the asynchronous technique, we noticed a very large quantity of redundant messages (ok messages). We have to point out to the fact that each agent works concurrently and asynchronously, therefore in each message queue there are more ok or nogood type messages gathered.

Starting from these observations, we have defined a simple filtering technique that applies on the message queues (that are FIFO communication channels). When we extract from the message queue a ok type message, this is not immediately dealt with by the nogood routine, but verified not to be redundant or old. If one of these situations should occur, the message is ignored by eliminating it form the message queue, otherwise the normal routine of dealing with ok messages is used.

We further present the filtering algorithm applied for each message queue (Fig. 6).

This filtering technique, applied to certain asynchronous techniques, allows an enhancement of the performances of asynchronous techniques. We have also applied this ok messages filtering method to the versions obtained here by applying the two techniques. The results are presented in Fig. 7, Fig. 8 and Fig. 9.

```
Procedure OK-messages-filtering

    Extract Msg(ok, x_i)
    If there is no other ok message in the message-queue received from x_i
        call the ok messages treatment procedure (ok, x_i)
    endif
end procedure
```

Fig. 6. The Procedure OK-messages-filtering.

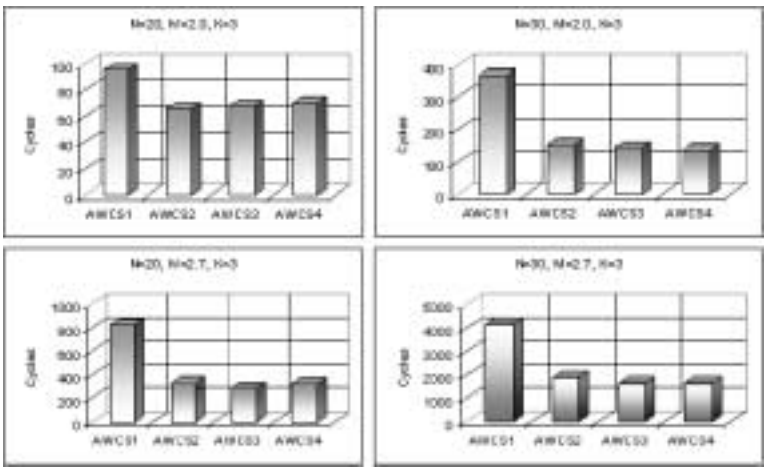

Fig. 7. Comparative study for the AWCS versions with messages filtering (Distributed n-Graph-Coloring Problem) – cycles.

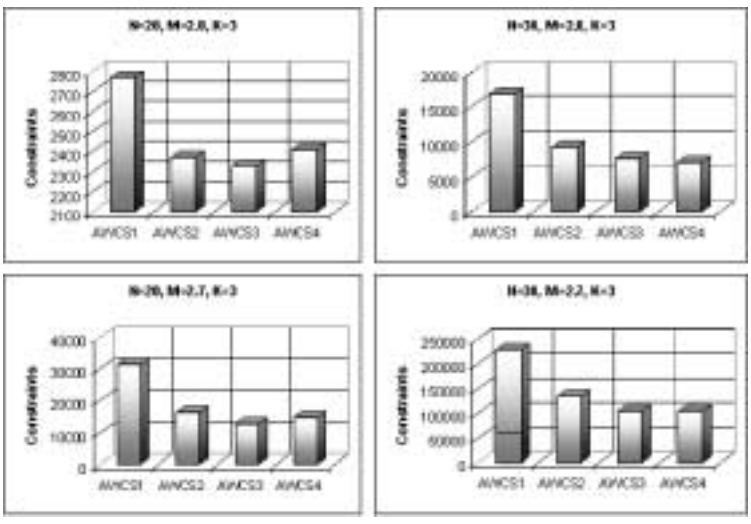

Fig. 8. Comparative study for the AWCS versions with messages filtering (Distributed n-Graph-Coloring Problem) – constraints checks.

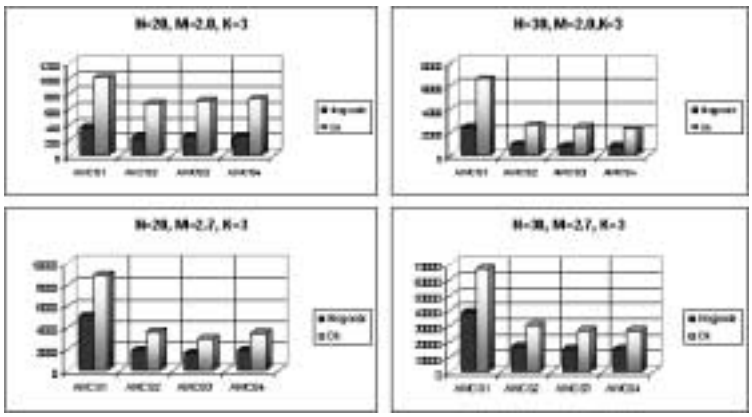

Fig. 9. Comparative study for the AWCS versions with messages filtering (Distributed n-Graph-Coloring Problem) – nogood and ok messages.

We observe the reduction of the costs when applying the filtering technique, regardless of what they were counted down. Messages filtering allowed reducing the number of cycles necessary for obtaining a solution, but also reduced the messages stream, respectively the number of verified constraints.

7. Conclusions

In this article we tried the adaptation of the nogood processor technique (Armstrong, 1997) and resolvent-based learning (Hirayama, 2000) for the AWCS. These techniques refer to the possibility of obtaining effective nogoods (resolvent-based learning), and to the way in which it is made the storing of nogood values, and later use of information provided by the nogoods in the process of selecting a new value. Starting from these techniques we propose certain modifications for the 2 known techniques, obtaining 2 derivate techniques. The purpose of combining the two was the enhancement of the AWCS technique's efficiency, because the techniques concentrate on minimizing and selecting the nogood values.

The adaptation needed certain modifications to the technique of building a new nogood value, a sort of restart of the nogood building process. We proposed a solution of distributing the nogood processors to agents, in regard to the agents' order, with the purpose of reducing the searching and storing techniques. We experimentally analyzed the benefits of the two techniques on the AWCS technique, by identifying few distribution means that bring enhancements to the efficiency. We also analyzed the performances in the case of application of the messages filtering on the analyzed versions.

We analyzed more versions obtained by distributing the nogood values to more nogood processors for each agent. For identifying the higher priority agents we took into account the current priority of the agent (being in current-view).

The most performing version was obtained by distributing the nogood values only to the agent that has the highest priority among the neighbors. The evaluations have shown

a reduction of the messages stream, of the number of verified constraints, but also of the number of necessary cycles for obtaining the solution. By distributing the nogood values and the nogood processors there, along with the resolvent-based learning technique, have been obtained enhancements of performances.

The messages filtering, applied for the proposed versions, led to reducing the costs in finding a solution. We believe that the filtering technique must be applied in all situations, to reduce the costs (the ones relative to the messages stream, along with the ones relative to the number of cycles or the quantity of verified constraints).

We believe that the combination of the two techniques under the form of the two proposed versions could bring important benefits to the performances of the asynchronous techniques, leading to the reduction of the effort in finding the solution.

References

Armstrong, A., and E. Durfee (1997). Dynamic prioritization of complex agents in distributed constraint satisfaction problems. In *Proceedings of the 15th IJCAI*, Nagoya, Japan. pp. 620–625.

Bessière, C., A. Maestre, P. Meseguer (2000). Distributed dynamic backtracking. In *Proceedings of the CP'00 Workshop on Distributed Constraint Satisfaction Problems*, Singapore, Thailand.

Hirayama, K., and M. Yokoo (2000). The effect of nogood learning in distributed constraint satisfaction. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems* (*ICDCS-2000*). pp. 169–177.

Minton, S., M.D. Johnston, A.B. Philips and P. Laird (1992). Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, **58**(1–3), 161–205.

Silaghi, M.C., D. Sam-Haroud and B. Faltings (2000). Asynchronous search with aggregations. In *Proceedings AAAI'00*, Austin, Texas. pp. 917–922.

Yokoo, M., E.H. Durfee, T. Ishida and K. Kuwabara (1998). The distributed constraint satisfaction problem: formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, **10**(5).

Yokoo, M. (2001). *Distributed Constraint Satisfaction-Foundation of Cooperation in Multi-agent Systems*. Springer.

Wilensky, U. (1999). *NetLogo*.
`http://ccl.northwestern.edu/netlogo.` Center for Connected Learning and Computer-Based Modeling. Northwestern University, Evanston, IL.

MAS Netlogo Models.
`http://ccl.northwestern.edu/netlogo/models/community/`,
`http://jmvidal.cse.sc.edu/netlogomas/`

I. Muscalagiu is a university lecturer, Ph.D students in Computer Science at the "Politehnica" University of Timisoara. His research interests include constraint programming and multi-agent system.

V. Cretu is a professor of science computer, a head of the Department of Computer Science and Engineering, The Politehnica University of Timisoara, Romania. His research interests include real-time and distributed systems, design and analyze of algorithms.

Asinchroninių algoritmų vykdymo tobulinimas derinant neadekvačius procesorius su neadekvačiais mokymo būdais

Ionel MUSCALAGIU, Vladimir CRETU

Analizuojamas būdas, kuriuo efektyviai tebeegzistuojančių neadekvačių reikšmių technika derinama su šių reikšmių saugojimo technika. Kitaip sakant, bandoma suderinti sprendimais grįstą mokymo būdą, pristatytą Yokoo, su neadekvačių procesorių technika asinchroninių silpnai suderinamų paieškos algoritmų atveju. Siūlomas neadekvačių procesorių paskirstymo tam tikra tvarka tam tikriems agentams sprendimas su tikslu sumažinti saugojimo ir paieškos sąnaudas.