Asynchronous Backtracking with temporary and fixed links: A New Hybrid Member in the ABT Family

IONEL MUSCALAGIU[1]
POPA HORIA-EMIL[2]
MANUELA PANOIU[1]

The "Politehnica" University of Timisoara,
The Engineering Faculty of Hunedoara
Revolutiei, nr. 5, Hunedoara, Romania

The University of the West,
The Faculty of Mathematics and Informatics
V. Parvan, nr.4, Timisoara, Romania
[1](mionel,m.panoiu)@fih.utt.ro
[2]hpopa@info.uvt.ro

Abstract. Starting from the algorithm of asynchronous backtracking (ABT), a unifying framework for some of the asynchronous techniques has recently been suggested (called ABT kernel). Within this unifying framework, several techniques have been derived, known as the ABT family. They differ in the way they store nogoods, but they all use additional communication links between unconnected agents to detect obsolete information. 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remove obsolete information is to add new communication links to allow a nogood owner to determine whether this nogood is obsolete or not.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. The second solution consists in temporary keeping the links. A new link remains until a fixed number of messages have been exchanged through it. After that, it is removed. In this article is proposed a solution for the elimination of outdated informations between agents, by adding new links for the purpose of informing the agents, some links becoming permanent, others temporary. It consists in combining the permanent links with the temporary ones. The solution is based on determining the number of messages necessary for keeping the temporary links, number determined dynamically during the runtime. Based on these informations some links become permanent, others are kept temporary, until that number of messages is exchanged between the agents connected by temporary links. A new hybrid technique can be obtained from ABT kernel by applying this method, the experiments show a better efficiency in comparison with the asynchronous backtracking.

Keywords: Artificial intelligence, distributed programming, constraints, agents

(Received November 18, 2005 / Accepted February 07, 2006)

1 Introduction

The constraint programming is a model of software technology used to describe and solve large classes of problems as, for instance, combinatorial problems, planning problems, etc. 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 useful, as it lead to obtaining a new modelling type called Distributed Constraint Satisfaction Problem (DCSP) ([7]).

According to the IT literature the backtracking algorithm distributed in an asynchronous way (Asynchronous

Backtracking -ABT), existing for the DCSP model, is considered the first complete algorithm for the asynchronous search. It is the first complete algorithm that is asynchronous, distributed and competitor, in which the agents can roll up in a competitive and asynchronous way, published in [7]. This algorithm is based on sending nogood messages among agents for doing an intelligent backtracking and for ensuring the completeness of the algorithm. The nogood messages are lists of joined values at distinct variables in which there are inconsistent among some variables.

Starting from the algorithm of asynchronous backtracking (ABT), it has recently been suggested in [1] a unifying framework, a starting kernel for some of the asynchronous techniques. From this kernel, several techniques have been derived, known as the ABT family. They differ in the way they store nogoods, but they all use additional communication links between unconnected agents to detect obsolete information. These techniques start from a common core (called the ABT kernel) which can lead to some of the known techniques, including the algorithm of asynchronous backtracking, by means of eliminating the obsolete information among agents. These techniques, too, within the ABT family are based on sending nogood messages among agents for performing an intelligent backtracking and assuring the completeness of the algorithm. Starting from the ABT kernel, one can obtain some of the known techniques, such as asynchronous backtracking (ABT), Distributed Backtracking (DIBT), Distributed Dynamic Backtracking (DisDB), [1].

In [1] were suggested several solutions for the elimination of the old information among agents: adding temporary links.

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 solution (called by its authors ABTemp, in [1]) consists in the temporary keeping of those links between the agents that cannot determine if an information is outdated or not. This algorithm adds new links between agents during search, as ABT. The difference is that new links are temporary. A new link remains until a fixed number of messages have been exchanged through it. After that, it is removed.

In [6] are investigated different values for the number of messages, values that are either statically determined (before the run) or dynamically during the runtime. In [6] a dynamical solution for determining the number of messages necessary for maintaining a connection is proposed, the experiments show a better efficiency (reported to the computational effort made) in comparison with the standard Yokoo variant. The dynamic solution is based on determining the maximum message flux of nogood messages and using those informations for determining the number of messages.

Starting from the dynamic solution proposed in [6] for determining the necessary number of messages needed for keeping a temporary link, in this article is proposed a new hybrid method for eliminating the outdated informations between agents. This solution consists in transforming some of the temporary links into permanent links, based on the information about the outdated message flux. Applying this method to the ABTkernel, we can obtain a new hybrid technique, technique that uses what's best in the two derived techniques: ABT and ABT temporary link. The experiments show a better efficiency (reported to the computational effort made) in comparison with the asynchronous backtracking.

2 The Framework

This paragraph presents some notions known from the IT literature related to the DCSP modeling, ABT algorithm [7] and ABT family, [1].

2.1 The Distribution Constraint Satisfaction Problem

Definition 1 *The model based on constraints CSP - Constraint Satisfaction Problem, existing for centralized architectures, consists in:*
- n variables X_1,...,X_n, whose values are taken from finite domains D_1, D_2,...,D_n.
- a set of constraints on their values.

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

Definition 2 *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.*

In this article we will consider that each agent A_i has allocated a single variable x_i.

The Asynchronous Backtracking algorithm uses 3 types of messages:

- 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 was found is being sent in case the constraint-evaluating-agent found 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. To make the network cycle-free there is a total order among agents, which is followed by the directed links.

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 to justify the deletion of the value it targets. 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 are sent to the closest agent involved, causing backtracking. The process terminates when achieving quiescence, meaning that a solution has been found, or when the empty nogood is generated, meaning that the problem is unsolvable.

2.2 The ABT Family

Starting from the algorithm of asynchronous backtracking (ABT), in [1], several derived techniques were suggested, based on this one and known as the ABT family. They differ in the way that they store 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, including the algorithm of asynchronous backtracking, by eliminating the old information among the agents. In [1] the starting point is a simple procedure that includes the main characteristics of the asynchronous search algorithms. Starting from this procedure, which forms the unifying framework, one can reach the known algorithms or variants that are close to them: asynchronous backtracking (ABT), Distributed Dynamic Backtracking (DisDB), Distributed Backtracking algorithm (DIBT).

The ABTkernel algorithm requires, like ABT, that constraints are directed- from the value-sending agent to the constraint-evaluating agent-forming a directed acyclic graph. Agents are ordered statically in agreement with constraint orientation. Agent i has higher priority than agent j if i appears before j in the total ordering. In this article we will consider the lexicographical order for the agents, order used also in the case of the asynchronous backtracking algorithm. Considering a generic agent self, $\Gamma^{-}(self)$ is the set of agents constrained with self appearing above it in the ordering. Conversely, $\Gamma^{+}(self)$ is the set of agents constrained with self appearing below it in the ordering.

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] were suggested several solutions for the elimination of 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. In that case, the hypothetically obsolete nogood and the values of unrelated agents are forgotten. These two alternative ways lead to the following four algorithms:

1. Adding links as preprocessing: ABTall. This algorithm adds all the potentially useful new links during a preprocessing phase. New links are permanent.

2. Adding links during search:ABT. This algorithm adds new links between agents during search. A link is requested by self when it receives a Back message containing unrelated agents above self in the ordering. New links are permanent.

3. Adding temporary links. This algorithm adds new links between agents during search, as ABT. The diference is that new links are temporary. A new link remains until a fixed number of messages have been exchanged through it.

4. No links: DisDB. 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.

In this article, we will propose a solution for combining the two methods for eliminating the outdated informations, solution that will lead to the fifth hybrid algorithm:

5. Adding temporary links: ABT with fixed and temporary links. This new algorithm adds new links during the search. A part of these links are temporary, they are kept until a certain number of messages is exchanged (number determined dynamically during the runtime). In exchange, some temporary links are transformed in fixed links, based on some information regarding the maximal flux of outdated nogood values.

3 Determining the value for the number of messages

In this paragraph we will present many solutions for determining the number of messages exchanged by the agents with temporary links, messages for which it must be kept a temporary link, solutions proposed in [6].

In [6] are proposed two types of solutions: statical solutions (for which the number of messages is fixed and doesn't change during the runtime) and dynamical solutions (for which the value of the number of messages varies during the runtime).

The static solutions proposed are based on determining for each agent, before the run, the value of the number of messages that have to be transmitted for a link. That number can be determined in many ways, many static variants were proposed. It will be used as the maximum number of messages transmitted.

The static variants suppose the building of the induced graph associated to the problem (in a preprocessing phase). To each DCSP problem we can associate a constraint graph, in which the nodes are agents/variables, and the edges are given by the existence of the constraints between agents/variables. From this constraint graph we can obtain the induced graph, corresponding to the existing order, by adding links between the parents of each node, if those links doesn't exist already. That graph is built as in [2]. Based on this graph, we can determine a number of fixed messages for each agent, as follows:

- The number of messages will be equal to the number of neighbors in the induced graph, for each agent. Each agent will keep a certain link until the number of messages exchanged will be equal to the number of neighbors.

- The second solution is obtained from the previous, but we compute a global value, common for all the agents, which is the greatest value of the numbers of neighbors of each agent.

The experiments in [6] show that for small dimension problems, the solutions are efficient, but along with the increasing in dimension of the problems, the basic variant surpasses clearly the variants with temporary number of links. Thus, the best solution, experimentally observed, is to dynamically determine the maximum number of messages during the runtime, which hasn't a fixed value. The number must be variable, for each agent, a global value wouldn't be the best choice. Practically, during the runtime, the value of the number of messages is dynamically deduced and it is adjusted depending of the evolution of the algorithm.

The dynamic variants proposed in [6] are based on using the informations regarding the outdated nogood message flow. That information changes during the runtime. As we know, when self receives a nogood, it is accepted if it is consistent with self's agent view, otherwise it is discarded as obsolete. The outdated message flow increases also because the agents are not informed (because of the inexistence of the supplementary links). Thus, each agent uses a supplementary data structure, for retaining the number of outdated nogood messages encountered at a given time. Those values are used for the determination of the number of messages exchanged for each temporary link. Practically, that value is the greatest number of nogood messages received at a given time. This is the first dynamical solution proposed. A second dynamical solution is obtained from the first, by also using the information regarding the number of neighbors each agent has. In the case of the first solution, at the beginning of the runtime, the outdated nogood message flow is very small; as a result the number of messages for which a link is kept is small. So, we start with a fixed value for the number of messages, equal to the largest number of neighbors from the induced graph. This initial value is actualized

during the runtime, using the largest value of the number of outdated messages, from all the agents.

4 Asynchronous Backtracking with temporary and fixed links

In [6], the experiments show that the second dynamic solution for determining the number of messages is the most efficient. This solution will be the starting point for building a new derived technique. Thus, we will use the information regarding the flux of outdated nogood messages, for determining the periods of runtime for which some temporay links will be kept. Unlike in that solution, some links will be transformed in fixed links. That will be the basic idea that will stay at the basis of constructing a new derived technique.

In [6], to increase the efficiency of the two dynamical variants proposed, each agent tries to keep as long as possible some temporary links. Starting from those observations, the solution proposed in that article consists in transforming some temporary links in fixed links. In fact, the temporary links with those agents with which it has exchanged a maximal flux of nogood messages, are transformed in fixed links. For each agent is determined the agent with which it had a maximal number of outdated messages (between those with which it had temporary links). The temporary link that exists with that agent will be transformed into a permanent one (the link will remain until the final stage in detecting the solution).

That solution supposes that each agent knows the maximum number of outdated messages received by each agent. A solution is based on the transmission of maximums for each agent to the ones it is connected, in the moment of the transmission of an info or nogood message. The idea is similar to that from the algorithm for determination of the additive cost in [4]. In figure 1 we present the algorithm for determining the maximum value amongst the maximums of the outdated messages flow. Each agent keeps a local list of counter variables (COldNogood) for counting the number of outdated messages received. Also, each agent keeps a counter MaxOldNogood which will retain the maximum of the numbers of messages between neighbors. Each agent, in the moment of transmitting a message, attaches the value for the maximum flux of outdated messages, value stored in MaxOldNogood. In exchange, at the receiving of a message from an agent A_k that contains the maximum value of it, *SenderMx* will update the value of the corresponding counter existing in the agent A_k in the list COldNogood.

This algorithm is applied to the ABT Yokoo technique, allowing each agent to retain the maximum value

Algorithm 1. Determining the maximum number of outdated messages received by the agents.

1: Each agent initializes the counter variable COldNogood with 0. Also, MaxOldNogood is initialized with 0.
2: When an agent sends a message it includes in the message the value of it's MaxOldNogood counter.
3: When an agent receives a outdated nogood value from a Sender agent, the corresponding counter from the COldNogood list is updated.
Replace-item Sender COldNogood **with** item Sender COldNogood + 1
4: **if** an agent receives a message with a counter SenderMx from Sender agent **then**
if item Sender COldNogood $<$ SenderMx **then**
Replace-item Sender COldNogood **with** SenderMx
end if
Set MaxOldNogood = max { COldNogood }
end if

Figure 1: Determining the maximum number of outdated messages received by the agents

for the number of outdated nogood messages exchanged between the agents.

For keeping the evidence of each agent's temporary links that have become fixed, we will use a list of flags that store the value 1 for a temporary link that has becomed permanent (in the algorithm in figure 2 is called FlagList). Practically, the existence of the value 1 in the list of flags of an agent, on the position k, will allow the keeping of the link to the agent A_k.

In figure 2 we show those modifications required in the ABT Yokoo technique (variant derived from the core ABTkernel), based on the method of determining of temporary links and of those temporary links transformed in permanent links, in order to to obtain a new hybrid technique, technique that uses what's best from both of the derived techniques: ABT and ABT temporary link. The modifications are marked with ***.

Obtaining this variant derived from ABTkernel, supposed many changes in the basic ABTkernel algorithm, derived in asynchronous backtracking. Some remarks should be made, for a better understanding of the modifications necessary in the ABT Yokoo code, modifications necessary for obtaining the ABT with temporary links variant.

First of all, each agent will use two extra sets $\Gamma_e^+(self)$ and $\Gamma_e^-(self)$, for the identification of the child and parent agents that appear because of the temporary links. In procedure ABTkernel(), in lines 1.1. and 1.2. they are determined. Also, it is necessary to introduce two

```
procedure ABTkernel()
1   myValue ←empty; end ← false;
1.1 Set Γ_e^+(self) ← ∅ ***
1.2 Set Γ_e^−(self) ← ∅ ***
2   CheckAgentView();
3   while (not end) do
4       msg←getMsg();
5       switch(msg.type)
6           Info : ProcessInfo(msg);
7           Back : ResolveConflict(msg);
8           Stop : end ← true;
9.1         AddL : SetLink(msg);
9.2         RemoveL: RemoveLink(msg); ***
end

procedure CheckAgentView(msg)
1   if not consistent(myValue;myAgentView) then
2       myValue← ChooseValue();
3       if (myValue) then
          for each child∈ Γ^+(self) do
            sendMsg:Info(child;myValue);
3.1       CheckRemoveLink(Self, Child) ***
4       else Backtrack();
end

procedure ProcessInfo(msg)
1   Update(myAgentView; msg.Assig);
2   CheckAgentView();
end

procedure ResolveConflict(msg)
1   if Coherent(msg.Nogood;Γ^−(self) ∪ {self}) then
2.1     CheckAddLink(msg)
3       add(msg:Nogood;myNogoodStore);
4       myValue ← empty; CheckAgentView();
5.1 else
        if Coherent(msg.Nogood; self) then
          SendMsg:Info(msg.sender; myValue);
5.2     Replace item Sender COldNogood with
          item Sender COldNogood + 1 ***
end

procedure SetLink(msg)
1   add(msg.sender;Γ^+(self));
2   add(msg.sender;Γ_e^+(self)); ***
3   sendMsg:Info(msg.sender; myValue);
end
procedure CheckAddLink(msg)
1   for each (var ∈ lhs(msg.Nogood))
2       if not (var ∈ Γ^−(self)) then
3           sendMsg:AddL(var,self);
4           add(var;Γ^−(self)); add(var;Γ_e^−(self)); ***
6           Update(myAgentView; var ← varValue);
end

procedure RemoveLink(msg) ***
1   remove(msg.sender;Γ^−(self));
2   remove(msg.sender;Γ_e^−(self));
end

procedure CheckRemoveLink(msg) ***
1   for each child ∈ Γ_e^+(self)
2       if (item child COldNogood = MaxOldNogood ) then
          replace item Child FlagList with 1;
3       if (item child CMessTemporaryLink ≥ MaxOldNogood
              and item Child FlagList = 0 ) then
4           remove(child;Γ^+(self));
5           remove(child;Γ_e^+(self));
6           sendMsg:RemoveL(child,self);
7           Update(myAgentView; var child ← unknown);
end

procedure Backtrack()
1   newNogood←solve(myNogoodStore)
2   if (newNogood = empty) then
3       end ← true; sendMsg:Stop(system);
4   else
5       sendMsg:Back(newNogood, x_j);
        /*where x_j has the lowest priority in V */
6       Update(myAgentView;rhs(newNogood)←unknown);
7       CheckAgentView();
end

function ChooseValue()
1   for each v∈D(self)not eliminated by myNogoodStore do
2       if consistent(v; myAgentView) then return (v);
3       else add(x_j = val_j) self ≠ v;myNogoodStore);
        /*v is inconsistent with xj 's value */
4   return (empty);
end

procedure Update(myAgentView; newAssig)
1   add(newAssig;myAgentView);
2   for each ng ∈ myNogoodStore do
3       if not Coherent(lhs(ng);myAgentView) then
          remove(ng;myNogoodStore);
end

function Coherent(nogood; agents)
1   for each var ∈nogood ∪ agents do
2       if nogood[var] ≠ myAgentView[var] then
          return false;
3   return true;
end
```

Figure 2: The ABT algorithm with temporary and fixed links.

data structures CMessageTempLink and COldNogood. The first structure will be used by an agent Ai to retain the number of info messages transmitted for each temporary link. The second structure is used for counting the number of outdated messages received.

The new algorithm needed the introduction of a forth message RemoveL, which announces a child agent of the canceling of a temporary link between two agents. Practically, the child agent will cancel the Sender agent from the list of its parents. The required modifications appear in line 9.2. from procedure ABTkernel() and procedure RemoveLink(), (a new added routine).

Third after selecting a new value and announcing the childrens about the new selection, is necessary the verification of temporary links to determine how many of them remain actual. This thing is done in procedure CheckAgentView(msg) line 3.1, by calling a new procedure named CheckRemoveLink(Self, Child). That routine verifies, for child agents from $\Gamma_e^+(self)$, if the maxim number of messages transmitted for that link is reached. That maximum number can be determined with the method proposed in this article. In case that the maximum value MaxOldNogood was attained, that link is canceled and the working framework (MyAgentView) is updated. Also, if the child agent is that that has attained the maximum number of nogood messages received, it is labeled with 1, such as it's link with the current agent becomes permanent and will not be canceled.

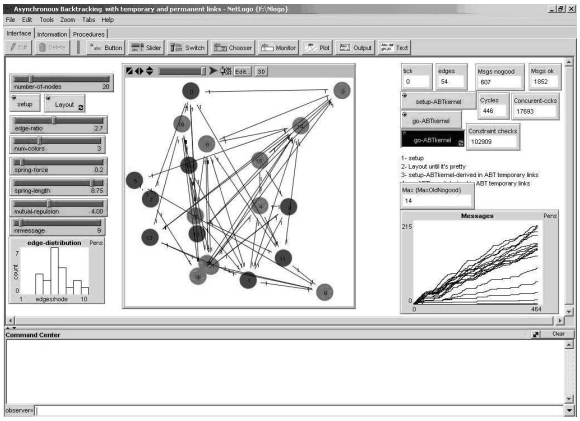

Figure 3: NetLogo implementation for Asynchronous Backtracking with temporary and fixed links

5 Experimental Results

This paragraph will present our experimental results, obtained by implementing and evaluating the asynchronous techniques that were introduced. In order to make such estimation, these techniques have been implemented in NetLogo 2.0.2, a distributed environment, using a special language named NetLogo, [8], [9], [10]. A screenshot is shown in Figure 3.

The asynchronous techniques were applied for a classical problem: the problem of coloring a graph in the distributed versions. This problem can represent various application problems such as channel allocation problems in mobile communication systems, in which adjoining cells (regions) cannot use the same channels to avoid interference. For the problem of graph coloring we took into consideration two types of problems defined as in [5]- (we kept in mind the parameters n - number of knots/agents, k-3 colors 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 x 2 connections) and graphs with a special number of connections, known to be difficult problems (called difficult problems and having m=n x 2.7 connections). For each version a number of 100 trials were carried out, 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 runs).

We counted the number of messages (which means the quantity of ok and nogood messages), the number of constraint checks (the local effort), the number of concurrent constraint checks (as defined in [4]). 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 ([8], [9], [10]), 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 being used. This type of agents work resembles the one used in [3] at the evaluation of AWCS algorithm together with resolvent-based learning.

By analyzing the message queues for the asynchronous technique (after the technique has been implemented in NetLogo) and applied for the problem of graph coloring, it observed a very large quantity of redundant messages. Certain agents have in the messages queue (that they manage) info type messages coming from the same agent, by which they are informed about the changing of that agent changing its value. These info type messages are redundant. Starting from these observations,

Procedure INFO-messages-filtering
Extract Msg(info, x_i)
If there is no other info message in the
message-queue received from x_i
[
call the info messages treatment procedure (info, x_i)
/* ProcessInfo(msg)
]
end

Figure 4: Filtering algorithm

we have defined a simple filtering technique that applies on the message queues. When we extract from the message tail an info-type message, this is not immediately dealt with by the info routine, but verified not to be redundant. Should this situation 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 (figure 4).

Therefore, for each version a number of 100 trials were carried out, but in the conditions of applying the filtering algorithm in the message queues. The two evaluated versions were called ABT (Yokoo's ABT), ABTTFL(ABT temporary and fixed links). The values obtained for the three graph classes, big size graphs (with 30, 40 respectively 50 knots) are stored in the 1st table. For the case of the version with temporary and permanent links two variants of algorithms were evaluated, corresponding to two ways of determining the maximal number of outdated messages received by each agent:

- a version in which the maximum number of outdated messages is determined with the method present in figure 1, version labeled $ABTTFL\#1$. Practically speakings, we obtain the maximum of the flux of outdated messages for all the agents.

- a version in which the maximum number of outdated messages is determined only between the neighbours of each agents, without the need of applying the method in figure 1. Practically, each agent determines MaxOldNogood from it's list COldNogood (that is a local maximum). That version we will label by $ABTTFL\#2$.

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 notice small computational efforts, for small dimensions, compared to the basic Yokoo variant. Besides, the second version of $ABTTFL\#2$, required a smaller effort for obtaining the solution than the first variant. Instead, once the problems increase in dimension (40 and 50 nodes), the $ABTTFL\#1$ variant is more efficient. Regarding the number of concurrent constraints verified, unfortunately the variants proposed require values closer to the basic ABT variant.

Table 1: The results for ABT versions (Graph-Coloring Problem)-constraint checks

n=30	Constraints		C-ccks	
	m=nx2	m=nx2.7	m=nx2	m=nx2.7
ABT	443073.78	484444.93	77567.37	76658
ABTTFL#1	264004.69	466694.05	62819.55	87131.03
ABTTFL#2	223118.56	434123.82	58791.95	81332.32

n=40	Constraints		C-ccks	
	m=nx2	m=nx2.7	m=nx2	m=nx2.7
ABT	1276261.88	3496629.71	187313.21	407892.71
ABTTFL#1	671172.36	2332383.68	127092.26	415689.09
ABTTFL#2	731456.78	2715132.67	138292.29	479856.25

n=50	Constraints		C-ccks	
	m=nx2	m=nx2.7	m=nx2	m=nx2.7
ABT	8003935.93	11706566.68	839876.36	1562254.00
ABTTFL#1	3899996.22	9641356.58	777720.36	1735377.40
ABTTFL#2	7328765.76	12674531.88	866721.34	2100675.20

Table 2: The results for ABT versions (Graph-Coloring Problem-nogood and ok messages

n=30	Nogood		Ok	
	m=nx2	m=nx2.7	m=nx2	m=nx2.7
ABT	1427.20	1691.20	5407.34	6142.25
ABTTFL#1	940.22	1741.61	3053.32	5669.61
ABTTFL#2	981.12	1614.23	3338.19	5551.78

n=40	Nogood		Ok	
	m=nx2	m=nx2.7	m=nx2	m=nx2.7
ABT	2920.43	8820.10	11323.40	32946.23
ABTTFLL#1	1873.33	6541.21	5922.09	20512.21
ABTTFL#2	2163.13	6623.51	6819.81	21231.85

n=50	Nogood		Ok	
	m=nx2	m=nx2.7	m=nx2	m=nx2.7
ABT	11008.18	21130.77	44184.56	71094.45
ABTTFL#1	8287.04	19858.26	26908.60	69642.81
ABTTFL#2	11745.71	23185.81	46174.70	83198.28

In the case of the message stream, the observed behavior of the computing effort remained aproximatively the same, the hybrid variants proposed needing a message flow much less than the Yokoo basic variant with fixed number of links. But in the case of problems with great dimension and density, the $ABTTFL\#1$ variant needed a smaller message stream than the $ABTTFL\#2$ variant.

The experimental values analysis shows that from

the two variants proposed, the first variant is the most efficient, compared to the second variant which for large dimension problems require much higher costs. Thus, the determination of the MaxOldNogood value is recommended, value used as maximum value for the number of messages transmitted for the temporary links, by applying the algorithm presented in figure 1. Another observation is relative to the two classes of problems analyzed: problems with a high density and those with a low density. The values obtained are closer to those of the basic variant, in the case of difficult problems (those with a density greater that 2.7). But, for problems with rare density the two hybrid variants proposed are more efficient than the basic Yokoo variant.

6 Conclusion

In this article is proposed a new member of the ABT family, member derived from the ABT kernel by eliminating the outdated information between agents, combining two older methods for eliminating outdated information between agents: adding permanent links between agents, and adding temporary links. The new member presume transforming some of the temporary links in permanent links, based on information relative to the outdated message flux received by each agent.

To obtain this hybrid technique a method for determining the number of messages that is transmitted for temporary links and a solution for determining the links that become permanent.

The evaluations realized in the conditions of various problems, with initial random values, for the problems with various densities, show that the proposed variants is more efficient than the basic technique, for both types of problems (with low density and difficult), for small and for large dimensions of the problems.

References

[1] Bessiere, C., Brito, I., Maestre, A., Meseguer, P. *Asynchronous Backtracking without Adding Links: A New Member in the ABT Family*. Artificial Intelligence, 161:7-24, 2005.

[2] Dechter, R., Pearl, J. *Network-based heuristics for constraint-satisfaction problems*. Artificial Intelligence, 34:1-38, 1988.

[3] 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, 169-177, 2000.

[4] Meisels, A., Kaplansky, E.,Razgon, I., Zivan, R. *Comparing performance of distributed constraints processing algorithms*. Notes of the AAMAS'02 workshop on Distributed Constraint Reasoning, pages 86-93, Bologna, Italy, 2002.

[5] Minton, S., Johnston, M.D., Philips, A. B., Laird, P. *Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems*. Journal of Artificial Intelligence, 58(1-3): pag. 161-205, 1982.

[6] Muscalagiu, I., Popa, H. E., Panoiu, M. *Determining the number of messages transmitted for the temporary links in the case of ABT Family Techniques*. Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania. IEEE Computer Society Press, 2005.

[7] 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), pag. 673-685, 1998.

[8] Wilensky, U. *NetLogo itself: NetLogo*. Available: http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, 1999.

[9] *MAS Netlogo Models-a*. Available: http://jmvidal.cse.sc.edu/netlogomas/.

[10] *MAS Netlogo Models-b*. Available: http://ccl.northwestern.edu/netlogo/models/community.