Scalable Computing: Practice and Experience
Volume 13, Number 1, pp. 59–71. http://www.scpe.org

ISSN 1895-1767
© 2012 SCPE

THE EFFECT OF TEMPORARY LINKS IN RANDOMLY GENERATED NETWORKS OF CONSTRAINTS

IONEL MUSCALAGIU[*], HORIA EMIL POPA [†] AND NEGRU VIOREL[‡]

Abstract. Additional communication links between unconnected agents are used in asynchronous searching, in order to detect obsolete information. A first way to remove obsolete information is to add new communication links, which allow a nogood owner to determine whether this nogood is obsolete or not. The second solution consists in temporarily keeping the links. A new link is maintained until a fixed number of messages have been exchanged through it. This article investigates different values for the number of messages, values that are either statically or dynamically, during the run time, determined. In the case of processing all the messages, we adapt a dynamical solution for determining the number of necessary messages for maintaining a connection. The experiments show a better efficiency in comparison with the standard Asynchronous Backtracking. In this paper we examine the effect of temporary links for the random binary constraints problem. Experiments with asynchronous search techniques are conducted on randomly generated networks of constraints. Experimental results show that the dynamical solution for the temporary links allows obtaining better results for the majority of classes of problems investigated.

Key words: Distributed constraint programming, asynchronous searching techniques, multiagent systems, messages.

AMS subject classifications. 68T20, 68T42, 68W15

1. Introduction.

Constraint programming is a programming approach used to describe and solve large classes of problems such as searching, combinatorial and planning problems. Lately, the AI community has shown increasing interest in the distributed problems, which are solvable through modeling, done by constraints and agents. The idea of sharing various parts of the problem among agents that act independently and collaborate in order to find a solution by using messages has proved itself useful. It has also led to the formal problem known as the Distributed Constraint Satisfaction Problem (DisCSP) [12], [13], [6]. DisCSPs are composed of agents, each owning its local constraint network. Variables in different agents are connected by constraints. Agents must assign values to their variables so that all constraints between agents are satisfied.

There are complete asynchronous searching techniques for solving the DisCSP, such as the ABT (Asynchronous Backtracking), AWCS (Asynchronous Weak Commitment), ABTDO (Dynamic Ordering for Asynchronous Backtracking) and DisDB (Distributed Dynamic Backtracking) [2, 5, 12, 13, 15, 6]. Starting from the algorithm of Asynchronous Backtracking (ABT), there has recently been suggested, in [2] 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.

Several solutions for the elimination of the old information among agents were suggested in [2], such as 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 suggested in the original ABT algorithm.

A second solution (called by its authors ABTtemp, in [2]) consists in temporarily keeping those links between the agents that cannot determine if an information is outdated or not. This algorithm adds new links between the agents during the search, same as ABT. The difference is that new links are temporary. A new link is maintained until a fixed number of messages have been exchanged through it. After that, it is removed.

Different values for the number of messages are investigated in [7]. These values are either statically determined (before the run) or dynamically determined during runtime. A dynamical solution for determining the number of necessary messages for maintaining a connection is suggested in [7]. The first experiments show a better efficiency in comparison with the standard Yokoo version. The dynamic solution is based on determining the outdated nogood message flow and using that information for determining the number of messages.

Starting from the dynamical solution for determining the necessary number of messages needed for keeping a temporary link, in [8] is suggested a new hybrid method for eliminating the outdated information between

*Faculty of Engineering of Hunedoara, "Politehnica" University of Timisoara, Romania (ionel.muscalagiu@fih.upt.ro).
†Faculty of Mathematics and Informatics, West University of Timisoara, Romania (hpopa@info.uvt.ro)
‡Faculty of Mathematics and Informatics, West University of Timisoara, Romania (vnegru@info.uvt.ro).

the agents. This solution consists in transforming some of the temporary links into permanent links, based on the information about the outdated message flow. Applying this method to the ABT kernel, we can obtain a new hybrid technique, that takes what's best from the two derived techniques: ABT and ABT temporary link. A new dynamical solution for determining the number of necessary messages for maintaining a connection is suggested in this paper in the context of processing all the messages.

In a previous research presented in [8], the evaluation of the effect of temporary links is done using a particular problem: the problem of coloring a graph in the distributed versions.

The evaluation of the asynchronous search techniques depends on at least two factors: the types of problems used at the evaluation and the units of measurement used. There are a few types of problems about the evaluation in the DisCSP literature: the distributed problem of the m-coloring of a randomly generated graph and the randomly generated (binary) CSP. These problems are characterized by the 4-tuple (n,m,p1,p2), where: n is the number of variables; m is the uniform domain size; p1 is the portion of the n * (n - 1) /2 possible constraints in the constraint graph; p2 is the portion of the m*m value pairs in each constraint that are rejected by the constraint [11].

It must be mentioned that the randomly generated binary CSP are the most suitable for the evaluation, because they allow different densities for the constraints graph and they have many direct applications in real practice. Therefore, a complete evaluation supposes the selection of a varied class of problems - the more randomly chosen sets of data or the choice of sets of data which allow varied densities for the constraints graph. In this paper, extensive evaluation of the asynchronous search techniques with temporary links is conducted on randomly generated networks of constraints.

In a previous research [8], the evaluation of the effect of temporary links is done using NetLogo environment. NetLogo is a programmable modeling environment, which can be used for simulating certain natural and social phenomena [14]. Also, the NetLogo is a programming environment with agents that allows the implementation of the asynchronous techniques ([14], [16], [17]).

The evaluations from [8] were implemented using certain particularities, supplied by NetLogo, related to the asynchronous run of the agents. The agents work with the specific command "ask-concurrent". A command like this will allow launching the message treating routine, which is specific to each agent. 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. The evaluations performed in [8] are realized in particular conditions, which don't affect the generality of the results.

In order to make such estimation, in this paper these techniques are implemented in NetLogo. The implementation and evaluation is done using the extended model suggested in [9], model that is called DisCSP-NetLogo. Implementation examples for the ABT family can be found on the website [17]. In [9] a general implementation and evaluation model with synchronization and support for message management in Netlogo, for the asynchronous techniques is proposed. This model will allow the use of the NetLogo environment as a basic simulator for the study of asynchronous search techniques. This model can be used in the study of the agents' behavior in several situations, like the priority order of the agents, the behavior in the synchronous and asynchronous case.

2. The Framework.

This paragraph presents some notions related to the DisCSP modeling, ABT algorithm [12], [13], [6] and ABT family, [2].

2.1. The Distributed Constraint Satisfaction Problem.

The Distributed Constraint Satisfaction Problem (DisCSP) has been formalized in [12], [13].

Definition 2.1. *The model based on constraints CSP - Constraint Satisfaction Problem, existing for centralized architectures, is defined by a triple (*X, D, C*), where:* X=$\{x_1,...,x_n\}$ *is a set of n variables; whose values are taken from finite domains* D= $\{D_1, D_2,...,D_n\}$*;* C *is a set of constraints declaring those combinations of values which are acceptable for variables.*

The solution of a CSP implies to find an association of values for all the variables that satisfy all the constraints.

Definition 2.2. *A problem of satisfying the distributed constraints (DisCSP) is a CSP, in which the variables and constraints are distributed among autonomous agents that communicate by exchanging messages. Formally, DisCSP is defined by a 5-tuple (*X, D, C, A, ϕ*), where* X, D *and* C *are as before,* A $= \{A_1,...,A_p\}$ *is a set of* p *agents, and* ϕ : X $\longrightarrow$ A *is a function that maps each variable to its agent.*

In this article we will consider that each agent A_i has allocated a single variable x_i, thus $p = n$. Also, we assume the following communication model [12], [13]:

- agents communicate by sending messages. An agent can send messages to other agents iff the agent knows the addresses of the agents.
- the delay in delivering a message is finite, although random. For transmission between any pair of agents, messages are received in the order in which they were sent.

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 right.
- the *nogood* message, which contains a list (called nogood) with the assignments wherefore a looseness was found, is 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.

Definition 2.3. *Two agents are connected if there is a constraint among the variables associated to them. Agent A_i has a higher priority than agent A_j if A_i appears before A_j in the total ordering. Agent A_i is the value-sending agent and agent A_j the constraint-evaluating agent.*

Definition 2.4. *The $agent - view$ list belonging to an agent A_i is the set of the newest associations received by the agent for the variables of the agents to whom it's connected.*

Definition 2.5. *The nogood list is a set of associations for distinct variables for which an inconsistency was found (an unsatisfied constraint).*

The $agent - view$ list together with the stored *nogood* values constitutes the working context of each agent, depending on them the agent makes decisions.

Definition 2.6. *A nogood list received by agent A_i is consistent for that agent, if it contains the same associations as $agent - view$ for all the variables of the parent agents A_k connected with A_i.*

Definition 2.7. *A nogood message is outdated if it contains a nogood list that isn't consistent with the receiver's agent context.*

ABT requires links to be directed. A constraint causes a directed link between the two constrained agents: the value-sending agent, whence the link departs, and the constraint-evaluating agent, to which the link arrives. When the value-sending agent makes an assignment, it informs the constraint-evaluating agent, which tries to find a consistent value. If it cannot, it sends back a message to the value-sending agent to cause backtracking. To make the network cycle free there is a total order among agents, which is followed by the directed links. In this article the lexicographical order is used.

Each agent keeps its own agent view and nogood store. Considering a generic agent, its own agent view is the set of values that are assigned to agents connected to it by incoming links. A nogood is a subset of agent view. If a nogood exists, it means the agent cannot find a value from the domain consistent with the nogood. When agent A_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 a random agent makes an assignment, it informs those agents connected to it by outgoing links. The agent always accepts new assignments, updating its agent-view accordingly. When it receives a nogood, it accepts it if the nogood is consistent with the agent's own agent view, otherwise it is discarded as obsolete (outdated nogood messages). An accepted nogood is added to the agent's nogood store to justify the deletion of the value it targets. When the agent 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 [2], 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 [2] 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) [2], [5], [12].

The ABT kernel 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, also called the set of the parents of $self$. Conversely, $\Gamma^{+}(self)$ is the set of agents constrained with self appearing below it in the ordering, also called the set of the childrens of $self$.

The ABT kernel algorithm is a new ABT-based algorithm that does not require to add communication links between initially independent agents. The ABT kernel algorithm is sound but may not terminate (the ABT kernel may store obsolete information). In [2] were suggested several solutions for the elimination of the old information among agents, solutions that are summarized hereinafter.

A first approach 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 suggested 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 at preprocessing: ABT_{all}. 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: ABT_{temp}. This algorithm adds new links between agents during search, as ABT. The difference is that new links are temporary. A new link is maintained 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, it forgets all nogoods that hypothetically could become obsolete.

In [8], we proposed a new solution for combining the two methods for eliminating the outdated information, solution that will lead to the fifth hybrid algorithm:

5. Adding temporary (dynamic) links: ABT with permanent and temporary links (ABT_{TPL}). 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 runtime). In exchange, some temporary links are transformed in permanent links, based on some information regarding the maximal flow of outdated nogood values.

3. Asynchronous Backtracking with temporary and permanent links. ABT with permanent and temporary links requests links dynamically, exactly like ABT. When a new link is set from agent i to j, it is maintained for a fixed number k of Info messages going from A_i to A_j. After this number of messages has been sent, the link is removed and agents A_i and A_j become disconnected. The number k of messages for a link is known a priori by both agents.

Some solutions for determining the number k of messages exchanged by the agents with temporary links are suggested and analyzed in [7]. The solutions presented are of two types:

1. statical solutions - for which the number of messages is fixed and doesn't change during the run time;
2. dynamical solutions - for which the number of messages varies during the run time.

The suggested statical solutions are based on determining a value for k common for all the agents, which is determined statically, at the beginning. The statical version supposes the construction of the induced graph associated to the problem (in a preprocessing phase). To each DisCSP 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 (the nodes from $\Gamma^{-}(self)$), if those links don't already exist. That graph is built as in [4]: agents (graph nodes) are processed from last to first, when an agent (graph node) is processed, all its parents (related agents before it in the ordering) are connected by new links if they were not connected before.

Based on this graph, we can determine a fixed number of messages k for all agents, as follows: the number of messages will be equal to the greatest value of the numbers of neighbors of each agent in the induced graph.

The dynamic versions proposed in [7] and [8] are based on using the information regarding the outdated nogood message flow. That information changes during the run time. As we know, when the agent receives a nogood, it is accepted if it is consistent with its own agent view, otherwise it is discarded as obsolete (outdated nogood messages). The outdated message flow also increases because the agents are not informed (because of the nonexistence 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.

Definition 3.1. *For each agent we have a local list of counter variables for counting the number of outdated messages received (named $COldNogood$). Let $MaxNrOldNogood$ be the maximum value from the $COldNogood$ list.*

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 updated during the run time, using the largest value of the number of outdated messages, from all the agents.

The experiments presented in [7] and [8] show that the dynamical solution for determining the number of messages is the most efficient.

The solution suggested in [8] consists in transforming some temporary links in permanent links. In fact, the temporary links with those agents with which A_i has exchanged a maximal number of outdated nogood messages are transformed in permanent links. For each agent is determined the agent A_j with which A_i had exchanged the maximal number of outdated messages (item A_j *COldNogood* = *MaxNrOldNogood*), among those with which it had temporary links. The temporary link that exists with that agent will be transformed into a permanent link.

The agents exchange among themselves the values of $MaxNrOldNogood$ in order to determine and use the maximum one. That solution supposes that each agent knows the maximum number of outdated messages received by each agent ($MaxNrOldNogood$). A solution is based on the transmission of $MaxNrOldNogood$ of each agent to the ones it is connected with, in the moment of the transmission of an info or nogood message. The idea is presented in [8]. Each agent, in the moment of transmitting a message, attaches the value for the maximum flux of outdated messages, value stored in $MaxNrOldNogood$. In exchange, at the receiving of a message from an agent A_k that contains the maximum value of it, *SenderMaxOldNogood* will update the value of the $MaxNrOldNogood$.

In figure 3.1 we show those changes required in the ABT technique (version derived from the core ABT kernel), based on the method of determining temporary and permanent links. We obtain a new hybrid technique, technique that uses whats best from both of the derived techniques: ABT and ABT temporary link. The lines from the figure 3.1 marked with two digits are additional to the algorithm from [2] and the ones marked with *** contain modifications to those from the cited algorithm.

The obtaining of the version with temporary and permanent links supposed many changes in the basic ABT kernel algorithm.

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

The new algorithm needed the introduction of a fourth message *RemoveL*, which notified a child agent about 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 changes appear in line 9.2. from procedure $ABTkernel()$ and procedure *RemoveLink*(), (a newly added routine).

Third after selecting a new value and announcing the child agents about the new selection, it is necessary that the verification of temporary links determines how many of them remain actual. This thing is done in procedure $CheckAgentView(msg)$ line 3.1, by calling a new procedure named $CheckRemoveLink()$. That routine verifies, for child agents from $\Gamma_e^+(self)$, if the maximum number of messages that are transmitted for that link has been reached.

```
procedure ABTkernel()
1    myValue ←empty; end ← false;
1.1  Set $\Gamma_e^+(self) \leftarrow \emptyset$ ***
1.2  Set $\Gamma_e^-(self) \leftarrow \emptyset$ ***
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∈ $\Gamma^+(self)$ do
                 sendMsg:Info(child;myValue);
3.1          CheckRemoveLink() ***
4        else Backtrack();
end

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

procedure ResolveConflict(msg)
1    if Coherent(msg.Nogood;$\Gamma^-(self) \cup \{self\}$) 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;$\Gamma^+(self)$);
2    add(msg.sender;$\Gamma_e^+(self)$); ***
3    sendMsg:Info(msg.sender; myValue);
end

procedure CheckAddLink(msg)
1    for each (var ∈ lhs(msg.Nogood))
2        if not (var ∈ $\Gamma^-(self)$) then
3            sendMsg:AddL(var,self);
4            add(var;$\Gamma^-(self)$); add(var;$\Gamma_e^-(self)$); ***
6            Update(myAgentView; var ← varValue);
end
```

```
procedure RemoveLink(msg) ***
1    remove(msg.sender;$\Gamma^-(self)$);
2    remove(msg.sender;$\Gamma_e^-(self)$);
end

procedure CheckRemoveLink() ***
1    for each child ∈ $\Gamma_e^+(self)$
2        if (item child COldNogood = MaxNrOldNogood ) then
             replace item Child FlagList with 1;
3        if (item child CMessTemporaryLink ≥ MaxNrOldNogood
                 and item Child FlagList = 0 ) then
4            remove(child;$\Gamma^+(self)$);
5            remove(child;$\Gamma_e^+(self)$);
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
```

Fig. 3.1: The ABT algorithm with temporary and permanent links.

4. Experimental results. The evaluation of the asynchronous search techniques depends on at least two factors: the types of problems used for the evaluation and the metrics of measurement used. There are a few types of problems used for evaluation in the DisCSP literature:

- the distributed problem of the m-coloring of a randomly generated graph, characterized by the number of nodes/agents, k=3 colors and the m-number of connections between the nodes/agents. Two types of problems are defined: sparse problems (having m=n x 2 connections) and dense problems (m=n x 2.7).
- The randomly generated (binary) CSPs are characterized by the 4-tuple (n,m,p1,p2), where: n is the number of variables; m is the uniform domain size; p1 is the portion of the n * (n - 1) /2 possible constraints in the constraint graph; p2 is the portion of the m*m value pairs in each constraint that are disallowed by the constraint. That is, p1 may be thought of as the density of the constraint graph, and p2 as the tightness of constraints.

4.1. The randomly generated DisCSP. A randomly generated DisCSP is an example of a homogeneous unstructured problem [11]. These problems have a number of variables with a fixed domain. Variables belonging to constraints are chosen randomly. Specifically, we implemented and generated in NetLogo both solvable and unsolvable randomly generated DisCSPs. These problems had one variable per agent so all constraints are between variables belonging to different agents (inter-agent constraints). Specifically, a tuple $< n, d, p1, p2 >$ was generated, where n is the number of variables, d is the domain size of all variables, p1 is the constraint density and p2 is the constraint tightness.

We implement in NetLogo a random instance generator in two steps [17]:
S1: We select with repetition $nr(C) = p_1\frac{n(n-1)}{2}$ random constraints. Each random constraint is formed by selecting without repetition 2 of n variables.
S2: For each constraint we uniformly select without repetition $nr(v) = p_2 \cdot d^2$ incompatible tuples of values, i.e. each constraint relation contains exactly $1 - p_2 \cdot d^2$ compatible tuples of values.

Implementation examples for the random instance generator can be found on the website [17].

We used binary constraints with the constraint density controlling how many constraints were generated and the constraint tightness determining the proportion of value combinations forbidden by each constraint. For example, a constraint density of 0.4 would generate 40% of the possible constraints in the problem (i.e. (n* (n-1)/2) * 0.4 where n is the number of variables) and a constraint tightness of 0.5 would prevent 50% of the possible value combinations of variables involved in a constraint from satisfying the constraint. Such uniform random constraints networks of n variables, k values in each domain, a constraints density of p1 and tightness p2, are commonly used in experimental evaluations of DisCSP algorithms [2], [3], [6].

The experiments were conducted on networks with 15-20 agents (n = 15 or n=20) and 10 values (k = 10). Three density parameters were used, p1 = 0.2, p1=0.4 and p1 = 0.5. In many cases a density of p1 = 0.2 or 0.3 was used to represent sparse constraint networks and a density of p1 = 0.4 or p1=0.5 used for medium networks. The value of p2 was varied between 0.3 to 0.5. This creates problems that cover a wide range of difficulty, from easy problem instances to instances that take several CPU minutes to solve. For every pair (p1,p2) in the experiments we present the average over 100 randomly generated instances (for each version we carried out a number of 100 trials, retaining the average of the measured values). Specifically, we tested the random classes: $< 20; 10; 0.2; 0.3 >$, $< 20; 10; 0.4; 0.7 >$, $< 20; 10; 0.2; 0.3 >$,$< 20; 10; 0.2; 0.5 >$, $< 20; 10; 0.5; 0.3 >$, $< 20; 10; 0.5; 0.5 >$ (100 solvable and unsolvable instances).

Another experiment is done for networks with n=15 agents, p1=0.4 (medium constraint networks) where the tightness value p2 varies between 0.1 and 0.9 to cover all ranges of problem difficulty. This aimed to test all algorithms near the phase transition region where some problem instances are very difficult to solve [6], [11].

4.2. Evaluation of temporary links for the ABT family. In order to make such estimation, the families of ABT techniques are implemented in NetLogo [14], [16], [17]. The implementation and evaluation is done using the two models proposed in [9].

In order to make the evaluation of the asynchronous search techniques, the message flow was counted i.e. the quantity of info (ok) and back (nogood) messages exchanged by the agents, the number of checked constraints i.e. the local effort made by each agent, and the number of nonconcurrent constraints checks (defined in [6], noted with ncccs) necessary for obtaining the solution.

Asynchronous techniques use some message processing routines. Those procedures process sequentially or in packages the messages that are in the message queues. Typically, each agent extracts one or more messages from its communication channel and calls the appropriate message processing routine. In this paper we analyze two classes of implementations:

- A version in which the messages are read and processed sequentially, one by one [2] -noted with ABT_1. In this version, we eliminate the redundant and outdated messages of the info type;
- A version of the ABT family with complete processing of messages: each agent treats entirely the existing messages in its message queue- noted with ABT_2.

We will present in this paragraph a protocol for message management for the ABT technique [3], [10] in the context of temporary links. This protocol establishes the order in which the messages are treated and the moment in which is tried the association of a new value. Also, this protocol allows complete or partial processing of the messages, by means of the use of the *msize* parameter, which stands for the number of messages read at a given time from the message queue. The *msize* parameter can take values between 1 and the length of the message queue. In the case that *msize* is 1, the sequential message processing solution is obtained.

The protocol presented here supposes the following:

P1. It is processed message by message:

- if it is of the info type, the local work context is updated (agent-view).
- the local counter *MaxNrOldNogood* is updated with *SenderMaxOldNogood* (received from another agent).
- if the message is of the back type, it's stored and verified if it is outdated. If it is outdated, an ok message is returned to the sender to inform him of that. A part of the back messages are thus rejected.
- if the message is of the addlink or removelink type then it's treated normally

P2. The current agent value is saved.
P3. The work context is updated, updating the nogood values.
P4. The routine check-agent-view is called.
P5. The neighboring agents are notified if the agent has kept its old value.

Starting from this protocol we propose a message management routine. This version is presented in fig. 4.1. As we can see in fig. 4.1 each agent can process all the messages until the message queue is emptied, or exactly as many messages as there are in the moment of the call, operation accomplished with the lines 1 and 1'.

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.

Seven implementations are done corresponding to the version presented:

- Variants Yokoo based on the asynchronous model from [9]: ABT-Y_1 (one message), ABT-Y_2 (complete processing of messages).
- Versions that determine statically the number of messages (named ABT-S_1 and ABT-S_2, corresponding to the static solutions presented in the previous paragraph).
- Versions that determine dynamically the number of messages: ABT-TPL_1 (one message), ABT-TPL_{21} (complete processing messages - the solution proposed in this article) and ABT-TPL_{22} (complete processing message - solution proposed in [3]) . These versions are corresponding to the ABT algorithm with temporary and permanent links presented in the previous paragraph.

Results appear in table 4.1, where we report the number of checked constraints (Constr.) the number of nonconcurrent constraint checks (Ncccs) and the total number of messages exchanged(Tmess), averaged over 100 executions.

In figures 4.2 and 4.3 are presented the results of other experiments for n=15 agents and p1=0.4 (medium constraint networks) where the tightness value p2 varies between 0.1 and 0.9 to cover all ranges of problem difficulty. This aimed to test all algorithms near the phase transition region where some problem instances are very difficult to solve [6], [11]. Figure 4.2 shows the computational effort, the number of nonconcurrent constrain checks, for all three versions of ABT. Figure 4.3 presents the total number of messages sent by the algorithms in the same run.

As known, the quantity of constraints checked evaluates the local effort done by each agent, but the number of nonconcurrent constraint checks count computational effort of concurrently running agents only once during each concurrent running instances citemeis1. Analyzing the results from table 4.1, one can notice that the

```
to message-manage [msize]
  set nrm 0
1 while [not empty? message-queue and nrm < msize] or
1' while [not empty? message-queue] ***
 [
     set msg retrieve-message
     if (first msg = "stop")
       [ stop ]
     if (first msg = "info")
       [ Update MyContext with msg
       [ Update $MaxNrOldNogood$ with $SenderMaxOldNogood$ ]     // **if** max COldNogood < SenderMaxOldNogood
     if (first msg = "back")
       [ Update $MaxNrOldNogood$ with $SenderMaXOldNogood$ ]     // **if** max COldNogood < SenderMaxOldNogood
       [ ifelse (Not Is-obsolete msgNogood Sender)
         [ Store msg to BackSet] //builds the list containing the received back messages
         [ SendInfo msg]
       // if it is outdated the sender agent is announced according to the Is-Obsolete procedure
       ]
     if (first msg = "addl")
       [ SetLink msg ]
     if (first msg = "removel")
       [ RemoveLink msg ]
     set nrm nrm + 1
  ]
  UpdateContextInfo
  Check-agent-view
  If Not empty(BackSet)
    [ ProcessMessageBackSet]
end
```

Fig. 4.1: The message-manage procedure for the message management in the case of the techniques from the ABT family

Table 4.1: The results for ABT2 versions (n=20)

n = 20 agents		p1= 0.2		p1= 0.5	
		p2=0.3	p2=0.5	p2=0.3	p2=0.5
ABT-Y_2	TMess	28	238100	89288	279378
	Constr.	1353	4434588	2275457	3495841
	Ncccs	499	1324866	357750	405249
ABT-S_2	TMess	78	570448	110475	273908
	Constr.	1380	14813543	3274523	3876091
	Ncccs	504	3883867	421890	438485
ABT-TPL_{21}	TMess	76	229272	79199	253178
	Constr.	1373	4420864	1982961	3270209
	Ncccs	500	1313842	322808	392978
ABT-TPL_{22}	TMess	71	278654	91054	214715
	Constr.	1422	5920563	2582961	4363385
	Ncccs	533	1454423	362718	491903

dynamical solution of $ABTTL$ reduces the local effort made by the agents. In case of problems with low density, the two solutions require approximatively the same costs (messages and global effort). An explanation is given by the fact that no temporary links appear, the only differences are caused by the delays in supplying the messages. The more the difficulty of the problems and the density of the constraint graph grow (p2=0.5 or p1=0.5), the more the costs of the dynamical solutions decrease. But, as the difficulty of the problems increases (n=20 agents, p2=0.5), the static solution $ABTS_2$ required much greater efforts compared to the dynamical variant ABT-TPL_{21}.

In the case of the message flow, the solutions with temporary links require a smaller flow of messages. Unfortunately, with the increase of the number of agents and the difficulty of the problems (p2=0,5) the static solutions for the temporary links require a much greater flow of messages. This thing is caused also because the temporary links aren't kept long enough to detect obsolete information.

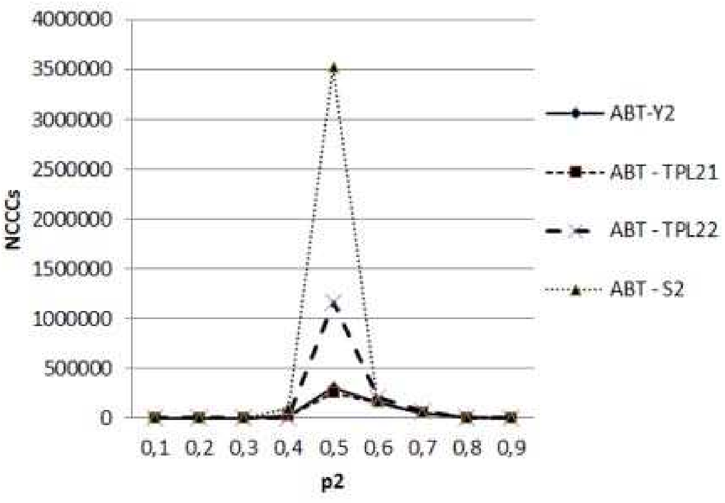

Fig. 4.2: The number of nonconcurrent constraint check for the ABT techniques.

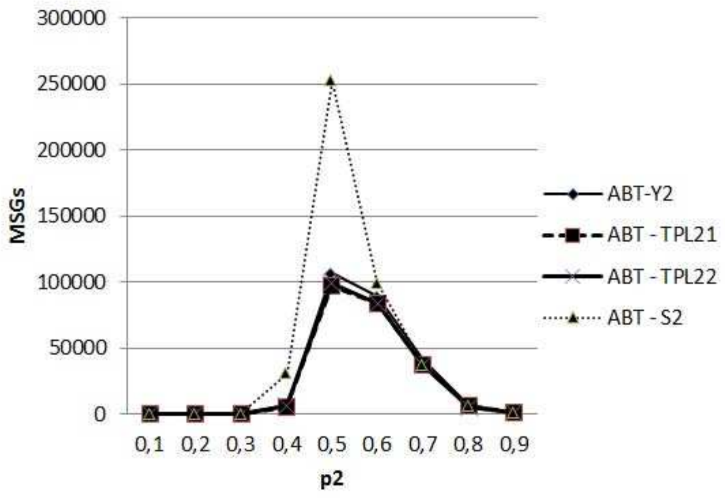

Fig. 4.3: Total number of messages sent for the ABT technique

As regarding the two dynamic solutions that use two different methods of treating the message in packages, the variant proposed here surpasses the one proposed in [3]. There can be observed situations in which the dynamical solution $ABTS_{22}$ is surpassed by the Yokoo solution.

Regarding the effort done by the agents,for the harder problem instances, ABT-TPL$_{21}$ outperforms ABTY by a factor of 1.1. Unfortunately, for the difficult problems we can observe a network load for the all solutions.

A version in which the messages are read and processed sequentially, one by one [2] - noted with ABT$_1$ is evaluated. This solution supposes a message treatment routine, which extracts sequentially each message, identifies its type and calls the appropriate processing routines. In this routine, for message processing, we eliminate the redundant and outdated messages of the info type.

In the case of the versions in which the messages are read and processed sequentially, one by one (noted with ABT$_1$) the results appear in table 4.2. These variants behaved similarly.

The results in the synchronous case where the agents' execution is synchronized appears in table 4.3 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

Table 4.2: The results for ABT1 versions - one message (n=20)

n = 20 agents		p1= 0.2	p1= 0.4
		p2=0.3	p2=0.7
ABT-Y_1	TMess	62	3290
	Constr.	2303	145422
	Ncccs	761	16362
ABT-S_1	TMess	63	3392
	Constr.	2330	153645
	Ncccs	766	18049
ABT-TPL_1	TMess	60	3220
	Constr.	2209	143001
	Ncccs	732	16174

of their messages. For this case we also count the number of cycles necessary obtaining the solution (Ncycles), which is a measure that could approximate the global effort (similar to NCCCs).

Table 4.3: The results for ABT1 versions - one messages (the synchronous case)

n = 20 agents		p1= 0.2	p1= 0.4
		p2=0.4	p2=0.7
ABT-Y_1	TMess	55	2474
	Constr.	2318	94958
	Ncccs	633	24176
	Ncycles	14	390
ABT-S_1	TMess	58	2079
	Constr.	2082	85439
	Ncccs	701	22746
	Nrcycles	14	370
ABT-TPL_1	TMess	55	2258
	Constr.	1949	89542
	Ncccs	622	23312
	Nrcycles	13	380

In this case, also we notice that the dynamical solution requires a lower flow of messages and also a lower global effort.

A general remark is that the static solutions applied to easy problems (low density or p2<0.4) require similar costs or even lower than all the other solutions. This thing is caused by the fact that the management of temporary links determines an extra overhead.

Unfortunately, analyzing the sets of results for certain instances (during runtime) we remarked the existence of problems for which the versions with temporary links (static versions) require very high costs. Although, we should specify that the number of those cases was not very high, not influencing, in the end, the results.

4.3. Discussion. It is interesting to see how many such links can be added by ABT during the search for a solution. The actual number will obviously depend on the instance to be solved, in [2] an estimate of the worst case is made, as follows: When a wipe out occurs on an agent A_i, the agent i builds a nogood by resolution of it's nogood store, and sends the obtained nogood to the agent A_j with the lowest priority in this set. When agent A_j receives the nogood, it checks the compatibility of the nogood with its own agent view. But, since this nogood can contain variables (x_k), unknown for agent A_j, agent A_j will ask the agents A_k to add a link from k to j. In the worst-case, a wipe out occurring at agent A_i will generate a nogood involving the whole set $\Gamma^-(i)$ of the agents linked to i, and preceding i in the agent ordering (the parents of node). More generally, when traveling back to all the ascendent agents, a nogood can lead to the addition of links between each pair of agents in $\Gamma^-(i)$, leading to a total number of links equal to $|\Gamma^-(i)|\ (|\Gamma^-(i)| + 1)\ /2$, see [2] for details.

The estimate presented previously was done in [2] for the worst case. In order to see, though, for the chosen data sets, how many links appear, during the experiments was counted also the number of temporary links. In figure 4.4 is presented the number of links for the chosen types of problems. An average was performed for all the runs and classes of problems. Surprisingly, this average is far from the values of the worst case. For problems in which p2 has small values (the constraint tightness) the number of temporary links is small, but for large values of p2≥4 the number of temporary links is almost the same.

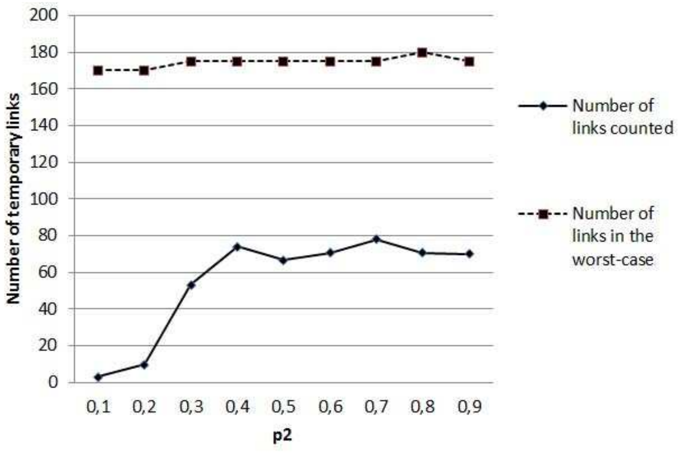

Fig. 4.4: The number of temporary links for ABT2 versions.

5. Conclusions. In this paper we examined the effect of temporary links for the random binary constraints problem. Experiments with asynchronous search techniques are standardly conducted on randomly generated networks of constraints. Experimental results illustrate that the dynamical solution for the temporary links has a better efficiency in comparison with the Asynchronous Backtracking.

A new dynamical solution for determining the number of messages necessary for maintaining a connection is proposed in this paper, the experiments show a better efficiency in comparison with the standard Asynchronous Backtracking.

The new member presumes transforming some of the temporary links in permanent links, based on information relative to the outdated message flux received by each agent.

From the experimental analysis we conclude that statical solutions proposed are not fitted for the case of networks with many links because they require a greater message flux. On the other hand, we remark a smaller general computing effort compared with the classical solution from [2], [12]. In conclusion it is recommended the use of dynamical variants that use message management and the agents work asynchronously.

A last comparison between the cases of processing the messages sequential or in packages, we can notice a neat differentiation between the dynamic solution and the classic or static solutions. The processing of all messages allows the agent to receive much faster the maximums of the other agents, compared to the situation in which it treats one message.

The scale-free graphs in complex networks, recently introduced by Barabasi and Albert [1], have become a very popular interdisciplinary research topic. As a future research, we wish to analyze temporary links in scale-free graphs, since there was little research in network structure for DisCSP.

REFERENCES

[1] A. L. Barabasi and A. L Albert, *Emergence of scaling in random networks*, Science, 286 (1999), pp. 509-512.
[2] C. Bessiere, I. Brito, A. Maestre and P. Meseguer,*Asynchronous Backtracking without Adding Links: A New Member in the ABT Family*, Artificial Intelligence, 161:7-24, 2005.
[3] I. Brito, P. Meseguer, *Synchronous, asynchronous and hybrid algorithm for DisCsp.* In Workshop on Distributed Constraints Reasoning, Toronto, 2004.
[4] R. Dechter and J. Pearl, *Network-based heuristics for constraint-satisfaction problems.* Artificial Intelligence, 34(1998), pp. 1–38.
[5] Y. Hamadi, C. Bessiere and J. Quinqueton, *Backtracking in distributed constraint networks.* In Proceedings ECAI'98, Brighton, UK, 1998, pp. 219–223.
[6] A. Meisels, *Distributed Search by Constrained Agents: algorithms, performance, communication*, Springer Verlag, London, 2008, pp. 105–120.

[7] I. Muscalagiu, H.E. Popa and M. Panoiu, *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.

[8] I. Muscalagiu, H.E. Popa and M. Panoiu, *Asynchronous Backtracking with temporary and fixed links: A New Hybrid Member in the ABT Family.* Journal of Computer Science INFOCOMP, Brazil, Vol. 5, nr. 2 (2006), pp. 29–37.

[9] I. Muscalagiu, H. Jiang, H.E. Popa, *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, Timisoara, IEEE Computer Society Press, 2006, pp. 209–216.

[10] H.E. Popa, I. Muscalagiu, D.M. Muscalagiu and V. Negru, *Experimental analysis of the impact of the message management in the case of the ABT family.* Proceedings of the 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania. IEEE Computer Society Press, 2007.

[11] B. Smith *Phase transition and the mushy region in constraint satisfaction problems.* In Proceedings ECAI'94, Amsterdam, The Netherlands, 1994, pp. 100–104.

[12] 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), 1998, pp. 673–685.

[13] Yokoo, M., Hirayama, K.*Algorithms for Distributed Constraint Satisfaction: A Review.* Autonomous Agents and Multi-Agent System, 3(2), 2000, pp. 198–212.

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

[15] R. Zivan and A. Meisels, *Dynamic ordering for asynchronous backtracking on Discsps*, Constraints, 11(2-3), 2006, pp. 179–197.

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

[17] *MAS NetLogo Models-c.* Available: http://discsp-netlogo.fih.upt.ro/.

Edited by: Dana Petcu and Daniela Zaharie
Received: March 1, 2012
Accepted: April 15, 2012