LiveABT: A Real-Time Repairing Protocol for Incremental and Dynamic DisCSPs

Amine Benamrane[1] **Yosra Acodad** [1] **El Houssine Bouyakhf** [1]
and Imade Benelallam [2]

[1]LIMIARF Laboratory
Faculty of Sciences Mohammed V University in Rabat, Morocco
benamraneamine@gmail.com
yosra.acodad@gmail.com
bouyakhf@fsr.ac.ma

[2]INSEA, National Institute of Statistics and Applied Economic
Irfane Rabat, Morocco
imade.benelallam@ieee.org

ABSTRACT

Several methods have been developed in Dynamic Distributed Constraint Satisfaction Problem ($DDisCSP$), for solving problems that change continuously over time. These proposed approaches require a batch processing resolution. A group of changes is collected and processed, over a period, after finding a solution or detecting the unsolvability of the initial problem. In contrast, many situations require immediate actions to act within the few seconds or minutes following the changes (e.g., flight delay management in a distributed air traffic control system). This paper proposes a new approach, called $LiveABT$, to solve Dynamic Distributed Constraint Satisfaction Problems. This new approach is based on the Asynchronous Backtracking (ABT) algorithm and real-time processing techniques that run the solver immediately against live perturbations.

We evaluate our approach and compare it to the Dynamic Asynchronous Backtracking ($DynABT$), the latest algorithm proposed in the context of $DDisCSP$, on various problems kinds. The results show that $LiveABT$ outperforms significantly $DynABT$ especially for the problems much more perturbed.

Keywords: DDisCSP, LiveABT, Repairing, RealTime, Perturbation, injection constraint, removing constraint

The ACM Computing Classification System: D13, I12, I28, I211.

1 Introduction

Distributed Artificial Intelligence (DAI) (Ferber, 1999) is concerned with interaction and coordination among artificial automated agents to solve a given problem.
In real life, most of the DAI problems (e.g, scheduling problems (Salido and Giret, 2008), sensor networks (Zhang, Xing, Wang and Wittenburg, 2003), allocation problems (Benamrane,

Acodad, Benelallam, El Houssine and Mohammed, 2014), servosystems (Precup, Preitl and Korondi, 2007), (Türkşen and Tez, 2016)) are dynamic. For example, in distributed meeting scheduling any participant could change his private calendar while the process of searching for a solution is turning, new meetings might need to be scheduled.
These unpredictable and challenging behaviors can be related to the:

- User: the requirements of the user can change in the context of an interactive design;
- Environment: the conditions/characteristics of the environment may evolve in the framework of a dynamic design;
- Agent: the autonomous decisions of the agent can change in the context of a distributed system;

Certainly, it is, possible to solve the new changed problem from scratch. This technique can remember nothing about the previous reasoning process, and has some drawbacks:

- Inefficiency, which is unacceptable in real time applications context (planning, scheduling, etc.), where the time allowed for re-planning is limited;
- Instability of the successive solutions, which may be unpleasant in the framework of an interactive design or a planning activity, if some work has started on the basis of the previous solution, or if it is necessary to keep the resource allocations as stable as possible etc.

Many complete and incomplete approaches such as, (Wallace, Grimes and Freuder, 2009) $DynABT$ (Omomowo, Arana and Ahriz, 2008) DBA (Mailler, 2005), DSA (Zhang, Wang, Xing and Wittenburg, 2005) have been proposed to solve Dynamic $DCSP$.
Generally, the solution found by the above approaches concerning the Dynamic $DCSP$ is based on the previous one, but these approaches can not consider the changes until solving the current problem instance, thus, they can not support real-time perturbations.
We distinguish between two scenarios of perturbation :

- Slow scenario: in which the time between two successive perturbations is enough to solve a problem, in this case, the use of the previous solution seems to be natural and simple.
- Rapid scenario: in which case, no enough time exists to solve the intermediate problems, the problem is continuously disturbed and the intermediate problems become quickly obsolete.

There are many recent applications for which the approaches must include the dynamism treatment in their native process, e.g. the many to many bargaining model especially for the high-frequency trading problem, the simulation of the biological cell division, etc.

This paper proposes a new approach, called $LiveABT$, for repairing solutions in $DDisCSP$. $LiveABT$ has the advantage to deal with changes in real time by including the perturbation management in its native behavior.

The present paper is organized as follows: First, we introduce the formalism of the $DDisCSP$. Then we present the ABT and the $DynABT$ algorithms. Next we outline our proposed algorithm (i.e., $LiveABT$) and the results of various experiments that compare $LiveABT$ with $DynABT$. Finally, we conclude and propose some perspectives.

2 Background

2.1 Distributed and Dynamic CSPs

A Constraint Satisfaction Problem (CSP) can be defined as a tuple (X, D, C), where:

- X is a set containing n variables x_1 , x_2 , ..., x_n ;
- D is a set of domains $D(x_1)$, $D(x_2)$,..., $D(x_n)$ for these variables, with each $D(x_i)$ containing the possible values which x_i may take;
- C is a set of m constraints c_1 , c_2 , ..., c_m between variables in subsets of X. Each $c_i \in C$ expresses a relation defining which variable assignment combinations are allowed/prohibited for the variables in the scope of the constraint.

Dynamic CSPs ($DCSPs$) were introduced (Dechter and Dechter, 1988) to cover problems that change continuously over time and were defined as series of $CSPs$ that one differ from the other in some of their constraints (added and/or removed). In fact, all changes can be represented as a series of constraints suppression and addition.

Formally, if P is considered as a dynamic constraint satisfaction problem, P is a sequence of static $CSPs$ $P_0, ..., P_\alpha, P_{\alpha+1}, ...$, where each P_i differs from the previous one by the addition or removal of some constraints.
A distributed CSP ($DisCSP$) (Yokoo, 2012) is a constraint satisfaction problem in which variables and constraints are distributed among multiple agents. A $DisCSP$ can be described as a four-tuple (X, D, C, A, α), where:

- X, D and C is a CSP.
- A is a set of agents.
- $\alpha : X \longrightarrow A$ is a function that maps each variable $x \in X$ to its agent $\alpha(x)$.

In distributed $CSPs$ ($DisCSPs$), we assume that:
Agents do not have a global view of the problem due to privacy, security and an agent can send messages to other agents if it knows their addresses. Also, the delay in delivering a message is finite, though random. Messages are received in the same order in which they were sent.
Solving a $DisCSP$ consists in finding an assignment of values to all variables, such that all constraints are satisfied
Dynamic and Distributed Constraint Satisfaction Problems ($DDisCSPs$) can be defined as a six-tuple (X, D, C, A, α, δ) where:

- X, D, C, A and α is a $DisCSPs$;
- δ is the change function which introduces changes at different time intervals, i.e., it represents changes in the problem over time.

$DDisCSPs$ can be used to model problems which are distributed in nature and change over time.
To simplify the algorithm's description in the following, we assume that the agents of the $DisCSP$ manage only one variable and all the constraints are binary.
There are more forms of the dynamism in the $DDisCSP$ network, the majority of these can be converted to adding/removing constraints. For example: removing a value from a domain can be translated as adding a unary constraint to the concerned variable, revision an existing constraint can be converted to remove and to add a new constraint with the new parameters. That is why we interested in this work to add and removal a constraint as a form of dynamic event in the $DDisCSP$ network.

2.2 Asynchronous Backtracking

Asynchronous Backtracking (ABT)(Yokoo, Durfee, Ishida and Kuwabara, 1998) (Yokoo, 2012) is the reference algorithm to solve $DisCSP$ problem. It is executed systematically, autonomously and asynchronously by each agent, which makes its decisions, informs other agents about them, and no agent must wait for the others' decisions. The algorithm computes a consistent global solution (or detects that no solution exists) in finite time. Its correctness and completeness have been proven (Yokoo, 2012). ABT requires a determined priority order of agents, and constraints are directed between the involved agents: the $value - sending$ agent, which is the higher priority one, and the $constraint - evaluating$ agent (lower priority). When a $value - sending$ agent assigns its variables, it sends their assignment to lower priority ($constraint - evaluating$) neighbors who try to make the assignment consistent.
If a $constraint - evaluating$ agent is unable to make a consistent assignment, it backtracks by sending a $nogood$ message to a higher priority agent, to change its current value assignment. Agents keep a nogood list of backtrack messages and use it to guide the search. A solution is found if there is quiescence in the network, while unsolvability is determined when an empty nogood is discovered.
ABT sends many obsolete messages and uses a large space to store nogoods, and various improvements (*ABT-family* (Bessière, Maestre, Brito and Meseguer, 2005),*ABT-Dyn* (Zivan and Meisels, 2006), *Agile-ABT*(Wahbi, 2013)) have been proposed to reduce the number of obsolete messages or the space required for storing nogoods or to support the ABT agents order changing.

Because $LiveABT$ is an extension of the ABT approach and use the same pseudo-code of ABT, we choose to present and describe more formally the ABT algorithm in the $LiveABT$ section.

2.3 Dynamic Asynchronous Backtracking

Dynamic Asynchronous Backtracking ($DynABT$) is an asynchronous algorithm for $DDisCSPs$ which is complete and does terminate (Omomowo et al., 2008). It is based on ABT, and the major difference is that it is designed to treat dynamic problems by repairing existing solutions when the problems change. The approach treats episodic constraints changes, i.e., changes (additions and removals) occurring after each problem has been solved. $DynABT$ use the previous solution of the initial problem, and the knowledge that agents had e.g agentView, nogood store, etc, and the constraint stored in the nogoods to detect and remove the obsolete nogood when a constraint is removed.
When detecting inconsistency, the agent composes a nogood of the form $x_i = a \cap x_j = b$ $\{C_1, ..C_n\} \Rightarrow x_k \neq c$. Thus, the justification ($\{C_1, ..C_n\}$) included in the nogoods indicates which nogoods should become obsolete when modifying some constraints.
$DynABT$ agents maintain, like ABT, a list of higher priority agents and their values in their $agentview$ and a list of values which are inconsistent with their $agentview$ in the nogood store. Higher priority agents send, in $info\,messages$, their value assignments to lower priority agents. When an $info\ message$ is received, the agent updates its $agentview$ and checks for consistency.
In $DynABT$, each agent initializes its variables, starts the search and solves the problem, like in ABT. Nevertheless, agents monitor the system to react when detecting changes. Problem changes are handled in two phases: the Propagation phase (lines 12-18) and the Solving phase (line 9). In the propagation phase, agents are informed of constraints addition/ retraction, then, they update their constraint lists, $neighbour\ lists$, $agentView$ and $nogoods$ where necessary. $AddConstraint$, $RemoveConstraint$ and $AdjustNogood$ messages are used during this phase to handle agents behavior. After propagating all changes, the new problem is defined, the $canProceed$ flag is set to true and the agents can move on to the Solving phase, in order to solve the new problem similarly to the ABT algorithm.

When an agent receives an $addConstraint$ message (lines 19-21), it updates its $constraint\ list$ and $neighbour\ list$ where necessary.
When receiving a $removeConstraint$ message (lines 22-26), the agent modifies its $neighbour\ list$ by removing from its $neighbour\ list$ and
its $agentview$ neighbors that only share the excluded constraint, then, the constraint is removed. When a constraint is removed, an $adjustNogood$ message (lines 27-34) is broadcasted to agents that are not directly involved in this constraint. The agents receiving this message update their nogoods store: they remove the nogoods containing the rejected constraint in justification and return the values to their domains since their source of inconsistency is no longer present in the network. Thus, the new problem starts at a consistent point before beginning the Solving phase.

```
Procedure DynABT()
1. $changes \leftarrow 0; changeBox \leftarrow empty; canProceed \leftarrow true$
2. $ABT^{+}$(ABT with nogoods containing justifications)
3. repeat
4.   $changes \leftarrow monitorChanges$
5.   if(changes) then
6.     $canProceed \leftarrow false$
7.     PropagateChanges(changeBox)
8.     current value$\leftarrow$ value from the last solution
9.     $ABT^{+}()$
10.  end if
11.  until termination condition met

Procedure PropagateChanges(changeBox)
12. while $changeBox \neq empty \cap canProceed \leftarrow false$ do
13.   $con \leftarrow getChange; changeBox \leftarrow changeBox - con$
14.   Swich(con.msgType)
15.   $con.removeConstraint : removeConstraint(con);$
16.   $con.addConstraint : includeConstraint(con);$
17.   $con.adjustNogood : incoherentConstraint(con);$
18. end while

Procedure IncludeConstraint(con)
19. $newCons \leftarrow$ con.getConstraint()
20. add new neighbours $\in newCons$ to $neighbourlist$
21. $constraintList \leftarrow constraintList \cup newCons$

Procedure ExcludeConstraint(con)
22. incoherentConstraint(con)
23. $constraint \leftarrow$con.getConstraint()
24. Remove unique neighbours $\in$ constraint
from $neighbour\ list$
25. Delete unique neighbours from $agentView$
26. Remove constraint from $constraintList$

Procedure AdjustNogoods()
27. IncoherentConstraint(con)
28. $constraint \leftarrow$con.getConstraint()
29. for each nogood $\in nogoodStore$ do
30.   if $contains(nogood, constraint)$ then
31.     return eliminated value $\in$ nogood to domain
32.     remove nogood from $nogoodStore$
33.   end if
34. end for
```

Figure 1: Pseudocode of $DynABT$ algorithm (Omomowo et al., 2008)

3 LiveABT

3.1 LiveABT description

$LiveABT$ is a new behavior of ABT, which supports three more messages types, to treat dynamism of $DisCSPs$. The three new messages, $AddConstraint$,

$RemoveConstraint$ and $AdjustNogood$, are the same of those used in $DynABT$. However, they are applied in a different way, with the aim of considering the perturbations in real time.
As known, ABT is a distributed algorithm, i.e., variables are distributed among multiple self-acting agents: each agent runs by himself, while managing only the context where it is involved. Therefore, to make ABT supporting changes, it is not necessary when injecting a perturbation to stop the whole multi-agent system for updating the network, as $DynABT$ does, since the perturbation does not impact all agents directly.
Thus, as all network communications are insured by messages, such as each agent, when receiving a message, reacts appropriately (e.g., receiving an Ok message is followed by an $update\ Agent\ View$ for the receiver agent, and receiving a $nogood$ message is systematically treated by $change\ value$), handling perturbations in ABT can be simply integrated into this communication protocol.
Then, to incorporate the $AddConstraint$ or $RemoveConstraint$ messages, the treatment can be done in the $solving$ phase exactly as for any other message.
Therefore, in $LiveABT$, a $propagation$ phase is not required. A perturbation injection is reflected by the immediate treatment of a message sent from the system, which is generally the master. This treatment consists of two parts: supporting the perturbation locally and propagating it in the network.

3.2 LiveABT vs DynABT

As shown in Fig. 2, $DynABT$ is executed in two major phases, the solving phase to solve the problem and the $propagation$ phase to update it. Although injecting a perturbation at a time t, $DynABT$ continues solving the initial problem as before, but stores the perturbation for a future treatment. After solving the initial problem (or detecting the unsolvability), $DynABT$ propagates the perturbations in the $propagation$ phase. Then, it restarts the $solving$ phase to repair the last found solution (or last assignment).
$LiveABT$, differently to $DynABT$, is executed in one phase: the $solving$ phase. Contrary to $DynABT$, whose $solving$ phase treats a problem which is entirely initially defined, $LiveABT$ has the advantage of supporting perturbations in its native behavior. Thus, when detecting a perturbation, this one is considered and propagated immediately, and the resolution of the whole problem (the problem after perturbation) is kept on. The propagation is done in the $solving$ phase which is continuously running.

3.3 Simulation example

Let us consider the example of a binary $DDisCSP$ containing 4 agents: $A1$, $A2$, $A3$ and $A4$. Each agent has only one variable, respectively $X1$, $X2$, $X3$ and $X4$. The problem constraints are as shown in Fig. 3 :

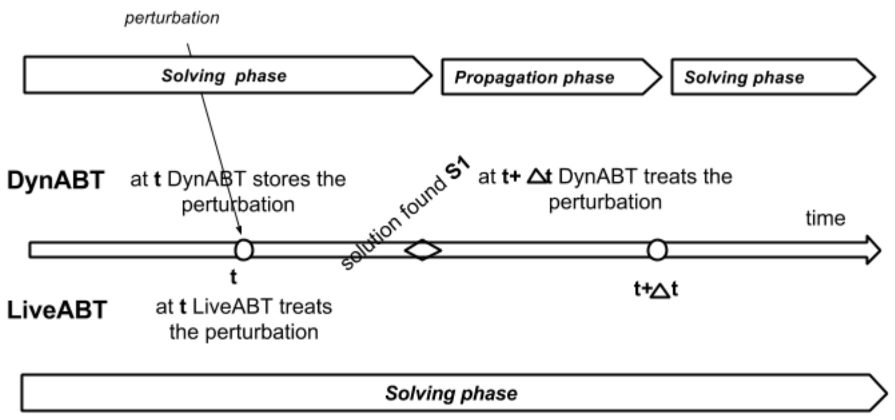
perturbation
Solving phase
Propagation phase
Solving phase
DynABT
at t DynABT stores the perturbation
solution found S1
at t+ Δt DynABT treats the perturbation
time
t
t+Δt
LiveABT
at t LiveABT treats the perturbation
Solving phase

Figure 2: behavior of $LiveABT$ vs $DynABT$

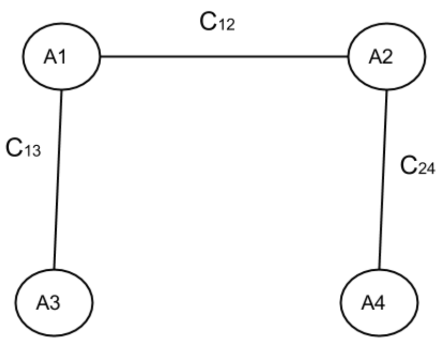

Figure 3: Example of a DisCSP

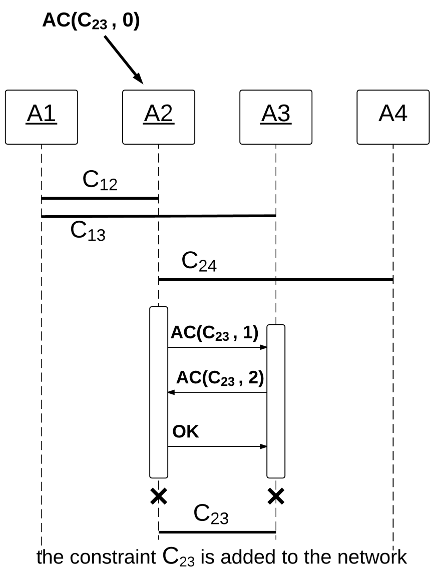

Figure 4: diagram of a constraint addition

3.3.1 Perturbation 1 : A constraint addition

When a perturbation affects the problem, such as a new constraint must be added into the network (C_{23} between the agents $A2$ and $A3$ as shown in Fig. 4).

The perturbation system manager sends an add constraint message ($AC(A2, C_{23}, stage)$ at the initial stage ($stage$=0) to the higher priority agent concerned by the constraint (the agent $A2$). This message contains the address of the second part of the constraint (the agent $A3$) since $A2$ ignores $A3$, which is not yet its neighbour, and the status of injecting the constraint which is incremented in each step. When $A2$ receives the $AC(A2, C_{23}, stage)$, it adds $A3$ in its neighbours list, puts C_{23} in its constraints list and sends the $AC(A3, C_{23}, stage++)$ message to $A3$. When $A3$ receives the $add\ constraint$ message, it adds $A2$ to its neighbours, updates its $agentView$, puts the new constraint C_{23} in its constraints list and informs $A2$ that the constraint addition is finished by sending to it the $AC(A2, C_{23}, stage++)$ message. $A2$ reacts by sending an OK message to $A3$, to evaluate the new constraint and change its value if necessary. At the end of the $add\ constraint$ process, the network becomes like Fig.5.
Thus, to add one constraint into the network, 4 messages are needed.

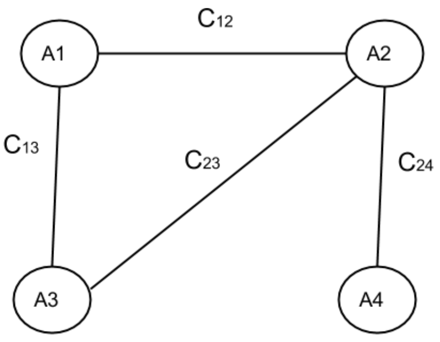

Figure 5: the problem after the constraint addition

3.3.2 Perturbation 2 : A constraint suppression

a- Removing a constraint:

If the coming perturbation is a constraint suppression, $LiveABT$ removes the constraint. Then, differently, to the previous case, it broadcasts the $AdjustNogood$ message to clean the agent's nogood stores by removing any nogood with the constraint as justification.
When $A1$ receives a remove constraint message ($RC(A1, C_{13}, 0)$) as Fig. 6 indicates, it removes $A3$ from its neighbors list, removes C_{13} from its constraints list and sends the $RC(A3, C_{13}, stage++)$ message to $A3$. The agent $A3$ also removes $A1$ from its neighbors list, updates its $AgentView$, removes C_{13} from its constraints list and sends the $RC(A1, C_{13}, stage++)$ message to $A1$, informing to it that the propagation of removing the constraint is finished. Then, $A1$ forwards this information to the system, generally the master, by the $RC(system, C_{13}, stage++)$ message. Therefore, to remove one constraint from the network, 4 messages are needed.

b- Updating the nogood stores:

Each agent, when receiving the $AdjustNogood$ message ($AdjNg\ C_{13}$ as shown in Fig. 7), checks his nogood store, tries to clean it and updates his domain. Thus, in this example, $A2$ removes the C_{13} justification from its nogood. However, it keeps the nogood, as C_{13} is not the sole justification. $A3$ restores 1 in his domain and cleans his nogood store (deletes the first nogood). Moreover, $A4$ keeps his nogood store after the check.

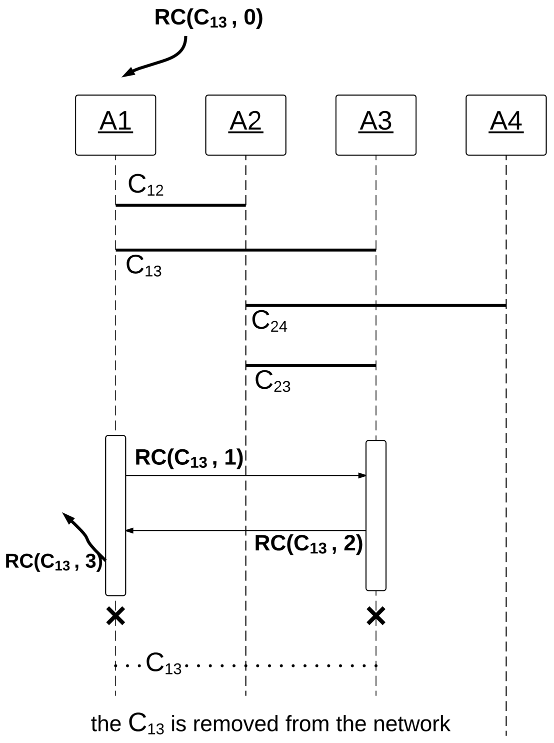

Figure 6: sequence diagram of a constraint removal 1/2

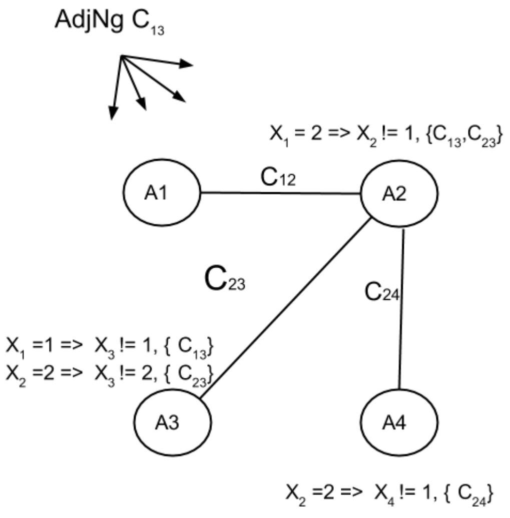

Figure 7: Constraint removal 2/2

To clean all the nogood stores, n messages, such as n is the number of agents in the network, are needed.

3.4 Execution

The description of $LiveABT$ is shown in Fig.8 and Fig.9. The $LiveABT$ kernel is the main procedure (lines 1-11). It is implemented in each agent.
Each agent detects its parents (AgentView) and its children (line 1) and listens to the network.

```
Procedure $LiveABT\_kernel()$
1.  $compute\ \Gamma + (self)\ and\ \Gamma - (self)$;
2.  $CheckAgentView(null)$;
3.  $end \leftarrow false$;
4.  while ($\neg end$) do
5.    $msg \leftarrow getMsg()$;
6.    switch ($msg.type$)
7.      Stop : $end \leftarrow true$;
8.      AddConstraint : $AC(con.getUpper, con, 0)$;
9.      RemoveConstraint : $RC(con.getUpper, con, 0)$;
10      Adjustngood : $IncoherentConstraint(con)$;
11.     else : $CheckAgentView(msg)$;

Procedure CheckAgentView(msg)
12. if msg is relevant then
13.   update local context ;
14. if Consistent(myValue, myContext) then Return;
15. myValue ← ChooseValue();
16. if myValue then
17.   for each child $\in \Gamma + (self)$ do
18.     sendMsg : Info(child, myValue);
19. else Backtrack;

Procedure Backtrack()
20. newNogood ← Solve(myNogoods) ;
21. if newNogood = empty then
22.   end ← true ;
23.   sendMsg : Stop(system);
24. else
25.   sendMsg : Back(newNogood);
26.   update local context;
27.   CheckAgentView(null);

Function ChooseValue()
28. for each $v \in D(self)$ not eliminated by myNogoods do
29.   if Consistent($v$, myContext) then Return ($v$) ;
30.   else
31.     Add($X = valX \Rightarrow \neg\ v$;$C\{ci\}$ , $myNogoods$);
32. Return ($null$) ;
```

Figure 8: Execution of algorithm $LiveABT$ 1/2

If the agent receives a $CheckAgentView$ message, it launches the $CheckAgentView$ function (lines 12-19). If the message comes from a higher priority agent, the receiver agent updates its $agentView$ (lines 12-13). If its value is not consistent (lines 14-15), it calls the $ChooseValue$ function (lines 28-32) to instantiate its variable. If the agent's variable is instantiated, the agent sends an $Info$ message to its lower priority agents, informing them to update their agentviews (lines 16-18). Else, the $Backtrack$ is launched, to inform the $value - sending$ agent to change its value, due to a domain wipe out (lines 20-27).

Throughout, each agent can receive an $AddConstraint$ or $RemoveConstraint$ perturbation

```
Procedure AddConstraint(con)
33. switch (stage)
34.   0 : Add con.neighbour to my Neighbours List;
35.       Add con to Constraints List;
36.       sendMsg : AC(con.getlower,con,stage++);
37.   1 : Add con.neighbour to Neighbours List;
38.       Add con to Constraints List;
39.       sendMsg : AC(con.getupper, con, stage++);
40.   2 : sendMsg : Info(con.getlower(), myvalue);

Procedure RemoveConstraint(con)
41. IncoherentConstraint(con);
42. switch (stage)
43.   0 : Remove con.neighbour from Neighbours List;
44.       Remove unique neighbour from AgentViews;
45.       Remove con from Constraint List;
46.       sendMsg : RC(con.getlower, con, stage++);
47.   1 : Remove con.neighbour from Neighbours List;
48.       Remove unique neighbour from AgentView ;
49.       Remove con from Constraint List;
50.       sendMsg : RC(con.getupper, con, stage++);
51.   2 : sendMsg : RC(system, con, stage++);

Procedure IncoherentConstraint(con)
52. for each nogood in nogoodStore do
53.   if nogood.justification.Contains(con) then
54.     nogood.justification.Remove(con);
55.     if nogood.justification.isEmpty() then
56.       restore nogood.eliminatedValue to domain;
57.       nogoodStore.Remove(nogood);
```

Figure 9: Execution of algorithm $LiveABT$ 2/2

message (lines 8-9).

If an agent receives the $AddConstraint$ message (line 8), $LiveABT$ reacts by calling the $AddConstraint$ process (lines 33-40), which updates all information about the two agents involved in the constraint, and integrates it into the network. $stage$[1] represents the progress of adding (or removing) a constraint. The AC and $info$ messages are exchanged between agents concerned by the added constraint to update their contexts and to activate the new constraint in the network.

If a constraint is removed (line 9), $LiveABT$ proceeds to remove it using $RemoveConstraint$ (lines 41-51). As for $AddConstraint$, RC messages are exchanged between concerned agents to accomplish the removal. Once the constraint is deleted, an $AdjustNogood$ message is broadcasted to clean all the stored nogoods that contain the removed constraint as unique justification (lines 52-57), in order to return the removed values to their domains. $IncoherentConstraint$ ensures that the invalidated values by removed constraints are restored to their domains.

[1]stage is a counter related to each perturbation, it is used by the agent to determine what the action must be executed because one agent can be involved in many perturbations

The $Stop$ message (line 7) is broadcasted from the master to all the network agents, with success, when a quiescence is detected, if a solution is found, or with a failure, if an empty nogood is generated.

3.5 Termination and Completeness

The termination of $LiveABT$ is assured since ABT terminates (Yokoo, 2012). In fact, $LiveABT$ respects the paradigm and uses its same mechanism to communicate all the pop-up perturbations in the network ($Add/Remove$ Constraint messages become like ok or $nogood$ messages), therefore, the approach terminates. Concerning the completeness, if the final problem (after the perturbations) has a solution, $LiveABT$ finds it. In fact, after integrating all the perturbations, $LiveABT$ protocol becomes the standard ABT algorithm, which is complete (Yokoo, 2012).

4 Experiments

We choose to compare experimentally $LiveABT$ to $DynABT$ since it is the latest $DDisCSP$ approach. Furthermore, $DynABT$ has already proved its large outperformance compared to $DynDBA$ (Mailler, 2005) in terms of runtime and network load. Algorithms are evaluated on uniform random binary $DDisCSPs$, on Graph Coloring Problems and also on Meeting Scheduling Problems.
To simulate a quantity p of perturbations like in reality, we choose to divide them over a various q randomly (standard normal distribution) moments of injections [t_1, t_q], which are generated when the resolution is running, and in each moment $t_i \in [t_1, t_q]$, the problem is disturbed with $\frac{p}{q}$ perturbations.
In fact, different ways to disturb problems exist, and as problems density is fixed, we choose the same scenario of perturbation presented in (Omomowo et al., 2008) that keeps density intact. Thus, perturbations are divided into two parts of $\frac{p}{2\times q}\%$ of added constraints and $\frac{p}{2\times q}\%$ of removed constraints. Then, the initial problem and the perturbed one have the same density.
For example, lets consider a problem with 60 constraints and we have p=10% is the rate of perturbation, based on the choosen scenario, 6 constraints will be changed 3 will be removed, and 3 will be added. We randomly generate q moments of time (injection moment). The distribution can be as $q = 3$ (1,1,1) three-time injections was generated, for each time one constraint will be added and one other will be removed, or $q = 2$=(1,2), i.e., for the first time one constraint will be added and one other will be removed and at the second time two constraints will be added/removed, we can have also q= 1(3), i.e., all the perturbations occur at one moment.
All experiments are performed on the $Dischoco$ platform (Wahbi, Ezzahir, Bessiere and Bouyakhf, 2011). We evaluate the performance of the algorithms by the total number of exchanged messages among agents ($\#Msg$) (Lynch, 1996) and non-concurrent computation. The non-concurrent computation is measured by the number of non-concurrent constraint checks ($\#NCCCs$) (Meisels, 2008).

4.1 Experiments on Random Problem

A uniform random binary $DDisCSP$ P is a sequence of uniform random binary $DisCSPs$: $P_0, P_1, ..., P_{\alpha-1}, P_\alpha$. The problem P_0 is the original $DisCSP$, and for each problem P_i, new constraints are added/removed to/from the previous one. In these experiments, we assume that the rates of perturbations are respectively 6% that represents an environment with lower perturbations and 30% witch represents the oscillated one.
$DisCSPs$ are characterized by $< n, d, p_1, p_2 >$, where n is the number of agents/variables, d the number of values in each domain, p_1 the network connectivity that is defined as the ratio of existing binary constraints to possible binary constraints, and p_2 the constraints' tightness, that represents a proportion of the conflicts within the constraints.
Tests are performed on problems from $<20, 10, 0.2, 0.1>$ to $<20, 10, 0.2, 0.9>$ [2] with a p_2 step of 0.1.
For each fixed tightness p_2, 50 different instances are disturbed randomly and treated by each approach. The presented results are the average of the 50 runs.

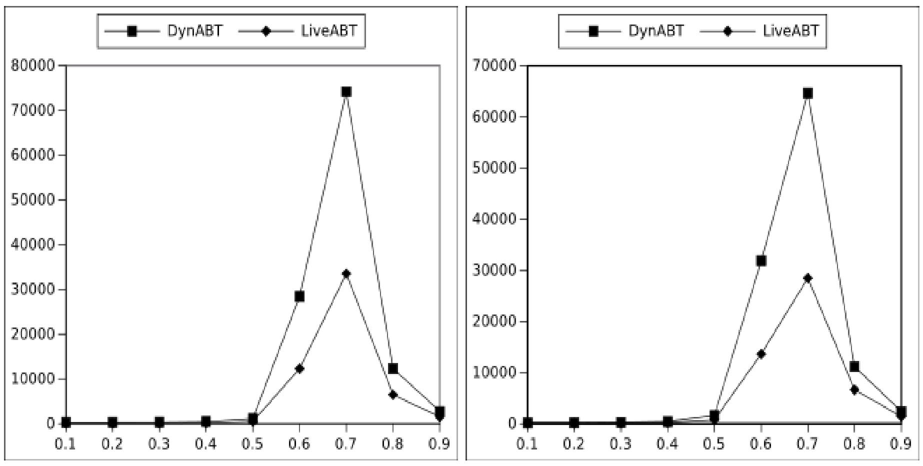

Figure 10: Random Problem: Number of sent messages and NCCCs for 6% of perturbation

Problems with constraints tightness between 0.1 and 0.6 are solvable, problems with a constraints' tightness equal to 0.7 are a mixture of solvable and unsolvable problems and beyond 0.7, problems are unsolvable.
The results show that $LiveABT$ outperforms $DynABT$, for both $NCCCs$ and $Msgs$ (Figs. 10 and 11), regardless of the problems solvability, especially when perturbations are important (Fig. 11).

4.2 Experiments on Graph Coloring

This problem can represent easily many real problems such as timetabling and channel allocation problems in mobile communication systems, in which adjoining cells can not use the same channels to avoid interference.

[2]we performed tests also on problems from $<30, 12, 0.7, 0.1>$ to $<30, 12, 0.7, 0.9>$. The results showed the same improvement

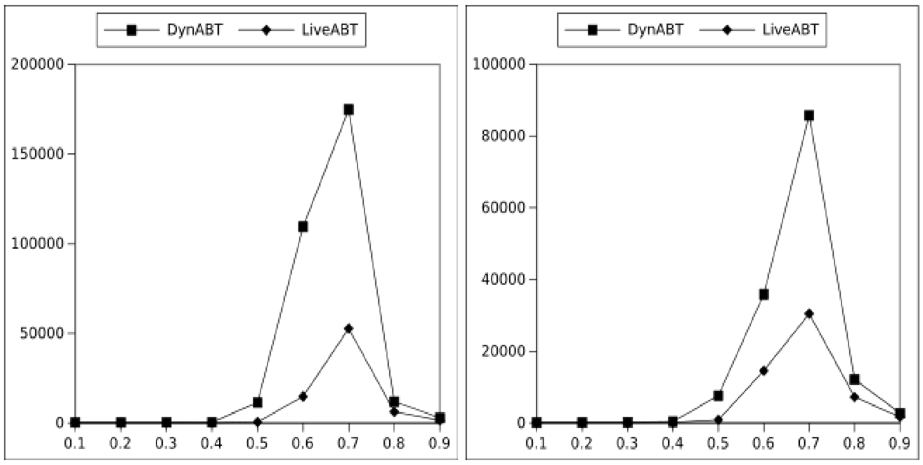

Figure 11: Random Problem: Number of sent messages and NCCCs for 30% of perturbation

A distributed graph-coloring problem is characterized by three parameters, i.e., the number of agents/variables n, the number of colors of each agent k, and the number of links (constraints) between agents m.

We randomly generated problems with 50 agents/variables and m arcs by the method described in (Minton, Philips, Johnston and Laird, 1993). The problems are connected and have a solution, the setting of k=3 and m=n*2.7 is the critical area (Cheeseman, Kanefsky and Taylor, 1991) i.e., the area where problems are difficult. We study the behavior of both $LiveABT$ and $DynABT$ when the perturbation rate increases.

For each rate of perturbation, 50 different instances are disturbed randomly from 2% to 40% by a step of 2% and treated with each approach. The presented results are the averages of these 50 runs.

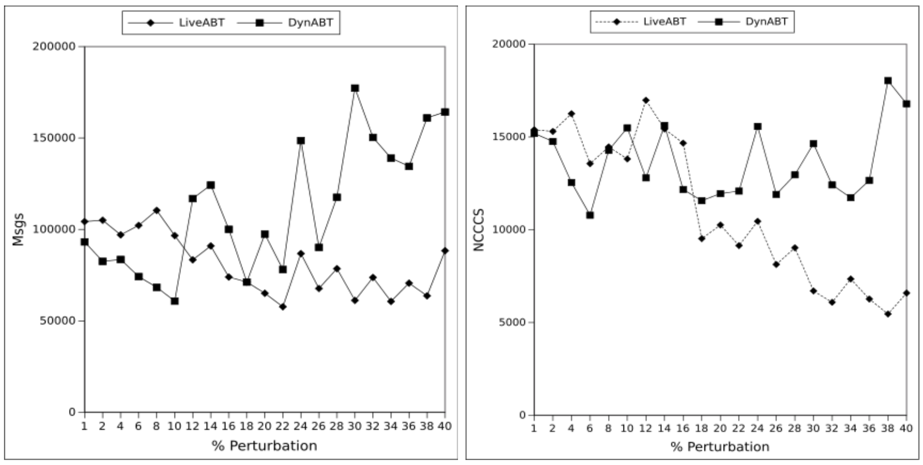

Figure 12: Graph Coloring Problem: behavior of $LiveABT$ and $DynABT$ according to the rate of perturbation

The results in Fig.12 show that $DynABT$ outperforms slightly $LiveABT$ when the perturbations are low. However $LiveABT$ makes less non-concurrent constraints checks ($NCCCs$) and sends less messages than $DynABT$ in middle and high perturbations.

We also note that when the problem is enough disturbed, $LiveABT$ deploys less effort to find a solution, contrary to $DynABT$ for which the effort is linearly increased according to the perturbations rate.

4.3 Experiments on Meeting Scheduling Problems

The Meeting Scheduling Problem (MSP) is one of most popular distributed problems, it consists of searching for a time and place when and where all the meeting participants are free and available.
Formally, a MSP is defined as following :

- $P = \{p_1, p_2, ...\}$, the set of participants.
- $S = \{s_1, s_2, ...\}$, the set of calendar for each participant.
- $M = \{m_1, m_2, ...\}$, the set of meetings.
- $At = \{at_1, at_2, ...\}$, the set of collections of participants that define which attendees must participate in each meeting, i.e. people in at_i must participate in the meeting m_i such as, $1 < i < k$ and $at_i \in P$.
- $L = \{l_1, l_2, ...\}$, the set of locations where meetings can be scheduled.

Distributed MSP ($DisMSP$) can be translated to $DisCSP$ as follows:

- Set of agents/participants.
- Set of n variables X: $\{$meetings to be scheduled for each participant is at most x_{mp} with $m \in M$ and $p \in P$ (i.e. number of variables is PM)$\}$.
- Set of Domains D: $\{$weekly time slots for each participant$\}$.
- Set of constraints C:
  - Intra-agent constraint : Arrival Time Constraint
    For every pair of variables ($x_m p$, x_{np} with $m \neq n \in M$) there is an Arrival Time constraint $t_{nm}(|x_{mp} x_{np}| \geqslant t_{nm})$, if there is a participant p which attends in both meetings m and n.
  - Inter-agent constraint : Equality Constraint
    For every pair of variables (x_{mp}, x_{mq} with $p \neq q \in M$) there is an equality constraint $x_{mp} = x_{mq}$, if there is two participants p and q attending a same meeting m.

The Dynamic Distributed MSP ($DynDisMSP$) can introduce two different types of changes:

- Additional Arrival Time constraint (traffic jams, train rescheduling, etc.)
- New links between meetings previously unconnected (a participant of meeting m_i is required to attend another meeting m_j as well).

Our experimental evaluation of $DynDisMSP$ introduces the first type of change.

We generated a set of 50 random instances of $DisMSP$ in order to compare the performances of $LiveABT$ and $DynABT$, in terms of $NCCCs$ and $MSGs$.
Distributed Meeting Scheduling Problem are characterized by
$< n, m, k, d, h, a >$, where n is the number of agents, m the number meetings, k the number of meetings per agent, d the number of days and h the number of hours per day, and a is a percentage of availability for each participant.
We present our results for the class $< 20, 4, 11, 3, 10, 1, 80 >$ and we vary the rate of perturbation constraints (i.e. Arrival Time Constraint) from 2% to 30%. We generated 50 different instances solved by each algorithm and the results are the averages of these 50 runs.

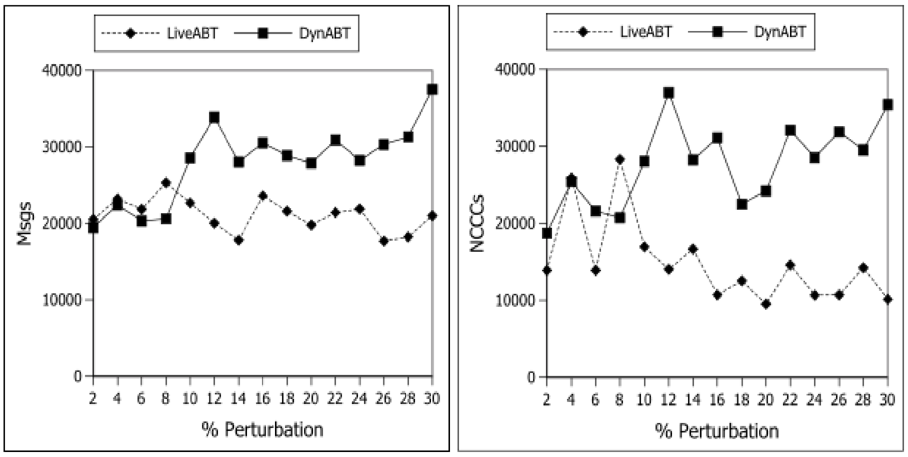

Figure 13: Meetings Scheduling Problem: behavior of $LiveABT$ and $DynABT$ according to the rate of perturbation

The results in Fig.13 show that when the perturbation rate is low, $DynABT$ and $LiveABT$ have the same behavior in terms of checking constraints and sending messages, however, when the perturbation rate increases (from $\geqq 10$), $LiveABT$ sends less messages and checks less constraints compared to $DynABT$.

4.4 Discussion

Firstly, for all perturbation rates on the random problems, namely 6%, and 30%, the results show that $LiveABT$ makes less $NCCCs$ and sends fewer messages than $DynABT$ to treat a problem. In fact, $LiveABT$ needs 4 messages when injecting a constraint, and $n_A + 4$ messages for a constraint removal, where n_A is the number of agents in the network. However, in addition to this quantity (n_A+4+4) to support a perturbation (one addition and one removal), $DynABT$ needs a larger number of broadcasted messages to stop and restart the resolution ($2 \times n_A$), to propagate the perturbation in the network.
Concerning the number of $NCCCs$, when the $DynABT$ solving phase is restarted to repair the solution, each agent launches the ABT process, i.e., evaluates its constraints, thus, $DynABT$ checks at least (if the perturbation have no effect on the current solution) n_C constraints more than $LiveABT$, such as n_C is the number of constraints in the problem.

Secondly, the behavior of $LiveABT$ and $DynABT$ on the meetings scheduling and graph coloring problems shows that when the problem is more disturbed, the effort of $LiveABT$ decreases. In fact, this is due to the impact of the constraint removal, which is much more important than adding a constraint. More precisely, when removing a constraint, all the agents are impacted, and they should clean their nogoods and recover the wasted values, however adding a constraint in real time effects generally only the two involved agents.
Thus, if the problem is disturbed in real time, it is relaxed during resolution.
Furthermore, the effort of $DynABT$ increases when the perturbation rates increase because $DynABT$ cumulates the efforts of the successive solving phases.
Finally, the major drawback of $DynABT$ is the propagation phase, in which the algorithm propagates the perturbations, therefore, for the small perturbations (a bit of propagation phase), in Fig.12 and Fig.13 $DynABT$ is a serious competitor for $LiveABT$.

5 Conclusion and perspectives

In this paper, we introduce a new repair protocol for $DDisCSP$, based on ABT approach, to solve the problems, which are continuously disturbed in real time. Based on the results of experiments on different rates of disturbance over different problem kinds, we show that this protocol exceeds $DynABT$, regardless of problems solvability. Also, we demonstrate by studying the $LiveABT$ behavior that in real time, the removal of constraints has a global impact contrary to the constraints' addition. As perspectives, we intend to develop $LiveABT$ agent to detect autonomously the different agents' perturbations (address changing, disconnection or addition of agents) and react appropriately without waiting for the Master instructions, and we also plan to upgrade and apply $LiveABT$ in other realistic use especially distributed robotic control system and the smart grid problem.

References

Benamrane, A., Acodad, Y., Benelallam, I., El Houssine, B. and Mohammed, B. O. 2014. Modeling trainings in faculty of medicine and pharmacy of casablanca as constraint satisfaction problem, CP 2014 : The Thirteenth International Workshop on Constraint Modelling and Reformulation, pp. 14–20.

Bessière, C., Maestre, A., Brito, I. and Meseguer, P. 2005. Asynchronous backtracking without adding links: a new member in the abt family, *Artificial Intelligence* **161**(1): 7–24.

Cheeseman, P., Kanefsky, B. and Taylor, W. M. 1991. Where the really hard problems are., *IJCAI*, Vol. 91, pp. 331–340.

Dechter, R. and Dechter, A. 1988. *Belief maintenance in dynamic constraint networks*, University of California, Computer Science Department.

Ferber, J. 1999. *Multi-agent systems: an introduction to distributed artificial intelligence*, Vol. 1, Addison-Wesley Reading.

Lynch, N. A. 1996. *Distributed algorithms*, Morgan Kaufmann.

Mailler, R. 2005. Comparing two approaches to dynamic, distributed constraint satisfaction, *Proceedings of the fourth international joint conference on Autonomous agents and multi-agent systems*, ACM, pp. 1049–1056.

Meisels, A. 2008. Message delays and discsp search algorithms, *Distributed Search by Constrained Agents*, Springer, pp. 143–158.

Minton, S., Philips, A., Johnston, M. D. and Laird, P. 1993. Minimizing conflicts: A heuristic repair method for constraint-satisfaction and scheduling problems, *Journal of Artificial Intelligence Research* **1**: 1–15.

Omomowo, B., Arana, I. and Ahriz, H. 2008. Dynabt: Dynamic asynchronous backtracking for dynamic discsps, *Artificial Intelligence: Methodology, Systems, and Applications*, Springer, pp. 285–296.

Precup, R., Preitl, S. and Korondi, P. 2007. Fuzzy controllers with maximum sensitivity for servosystems, *IEEE Transactions on Industrial Electronics* **54**(3): 1298–1310.

Salido, M. A. and Giret, A. 2008. Feasible distributed csp models for scheduling problems, *Engineering Applications of Artificial Intelligence* **21**(5): 723–732.

Türkşen, Ö. and Tez, M. 2016. An application of nelder-mead heuristic-based hybrid algorithms: Estimation of compartment model parameters, *International Journal of Artificial Intelligence* **14**(1): 112–129.

Wahbi, M. 2013. *Agile Asynchronous Backtracking (Agile-ABT)*, John, Wiley and Sons, Inc., pp. 111–130.
URL: *http://dx.doi.org/10.1002/9781118753620.ch7*

Wahbi, M., Ezzahir, R., Bessiere, C. and Bouyakhf, E. H. 2011. Dischoco 2: A platform for distributed constraint reasoning, *Proceedings of the IJCAI'11 workshop on Distributed Constraint Reasoning*, DCR'11, Barcelona, Catalonia, Spain, pp. 112–121.
URL: *http://dischoco.sourceforge.net/*

Wallace, R. J., Grimes, D. and Freuder, E. C. 2009. Solving dynamic constraint satisfaction problems by identifying stable features., *IJCAI*, Vol. 9, Citeseer, pp. 621–627.

Yokoo, M. 2012. *Distributed constraint satisfaction: foundations of cooperation in multi-agent systems*, Springer Science & Business Media.

Yokoo, M., Durfee, E. H., Ishida, T. and Kuwabara, K. 1998. The distributed constraint satisfaction problem: Formalization and algorithms, *IEEE Transactions on knowledge and data engineering* **10**(5): 673–685.

Zhang, W., Wang, G., Xing, Z. and Wittenburg, L. 2005. Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks, *Artificial Intelligence* **161**(1): 55–87.

Zhang, W., Xing, Z., Wang, G. and Wittenburg, L. 2003. An analysis and application of distributed constraint satisfaction and optimization algorithms in sensor networks, *AAMAS*, Vol. 3, pp. 185–192.

Zivan, R. and Meisels, A. 2006. Dynamic ordering for asynchronous backtracking on discsps, *Constraints* **11**(2-3): 179–197.