Nogood Learning in DisCSP Algorithms

Ghizlane EL KHATTABI
LIMIARF Laboratory
Rabat, Morocco
elkhattiabi.ghizlane@gmail.com

Imade BENELALLAM
SI2M Laboratory
Rabat, Morocco
imade.benelallam@ieee.org

El Houssine BOUYAKHF
LIMIARF Laboratory
Rabat, Morocco
bouyakhf@mtdns.com

ABSTRACT
The artificial intelligence (AI) is one of the powerful research area. AI gathers several topics such as Constraint Programming CP, Machine Learning ML, and Multi-Agent System MAS. Our contribution is inspired by these last AI topics: CP, ML and MAS. We therefore consider Distributed Constraint Satisfaction Problem formalism DisCSP, which is a Constraint Programming problem, distributed among several autonomous agents in a MAS system. To solve such problems, many algorithms are proposed in the literature. Most of them, rely on message exchanging to find a global solution that satisfies the set of constraints of each agent. Generally, two types of messages are used, the first type is used by each agent to inform others of its chosen value, and the second type (nogood) is used by the same agent to inform others that their choices had blocked it. The Machine Learning principle is used by launching asynchronously two DisCSP algorithms in the same DisCSP problem and sharing the nogoods of the two algorithms. Experimental results exhibit that each algorithm learns from the nogoods of the other algorithm.

KEYWORDS
Multi Agent System, Artificial Intelligence, algorithms, nogoods, Distributed Constraint Problem

ACM Reference Format:

1 INTRODUCTION
In recent years, ones of the most used areas in the artificial intelligence are i) the constraint programming[12], which allows to present and solve combinatorial real problems, such as scheduling and planning problems. ii) Multi-agent system[6], that represent a set of agents (or a set of computer systems) that collaborate in order to do a set of tasks, independently of humans. The two latter domains have been used to create another subdomain DisCSP (for Distributed Constraint Problems)[16], that is a mathematical problem, distributed among several autonomous agents, aiming to solve the problem together. And iii) the Machine Learning[1] that appeared a long time ago, but takes the hard meaning recently by the big data arrival. It bases on examples to learn new techniques.

DisCSP can represent several real distributed problems, As distributed Meeting Scheduling[9, 14], Distributed Resource Allocation Problem[11] and Sensor Networks[2, 8]. These problems, including big problems, require more treatment. So, we have to think to reduce the treatments, when using DisCSPs algorithms.

While there are several DisCSP algorithms, as ABT[5], AFC[10], AFC-ng[13] and AWC[15]. We do not have to create new algorithms, to solve the very big DisCSP problems.

The idea is to use the machine learning principle, which is adapted to big data, in the existing algorithms. In our contribution we use the two algorithms ABT and AFC-ng.

We will not designate, statically, which is the training algorithm and which is the testing one. The choice will be done dynamically and intelligently during the resolution process. At a given point, the fast algorithm which finds a failure the first will be the training one and the other the testing one. This affectionation of rules is not definitive, it can be changed during the resolution process. The contribution details will be described more accurately in the paper.

The paper proceeds as follows: the section 2 contains some useful definitions. The detailed description of the two used algorithms (ABT and AFC-ng) will be done in section 3. Then, the main contribution is presented in section 4. And section 5 will show the experimental results. Finally we resume by a conclusion in section 6.

2 SOME DEFINITIONS
We recall briefly some fundamental definitions in the context of constraint programming domain.

Definition 2.1 (MAS). A Multi-Agent System is a set of autonomous agents, interconnected via certain relations. These agents share a problem and try to solve it collectively.

Definition 2.2 (CSP). A CSP (Constraint Satisfaction Problem) is a mathematical problem, comprised of a set of Variables V, defined in a set of Domains D and should satisfy a set of Constraints C.

Definition 2.3 (DisCSP). A DisCSP (Distributed CSP) is a CSP whose components (variables, domains, and constraints) are distributed among several agents (a MAS). So it is formalized as a set of Agents A, the same three CSP parameters V, D and C and a function $\delta$ that associates each variable to an agent.

Definition 2.4 (Solution). A solution is an assignment of all variables with values from its domains, so as the existing constraints are satisfied.

Definition 2.5 (Nogood). A nogood is a partial assignment that cannot be extended to a solution. In DisCSP algorithms, the nogood takes the form $x_i = v_i \land x_j = v_j \land ... \rightarrow x_k = v_k$ which means that as long as $x_i$ has the value $v_i$ and $x_j$ has the value $v_j$, $x_k$ can not take the value $v_k$. The part which exist before the $\rightarrow$ symbol is called...
the left-hand-side (lhs) and the other is called the right-hand-side (rhs).

Definition 2.6 (Concurrent Constraint Checks CCCs). The Concurrent Constraint Checks is a metric used to evaluate the DisCSP algorithms. It computes the number of the constraints checked concurrently. Each agent handles a constraint counter. Each message sent carries this value. The receiver tests if the received value is greater than the value of its counter. If so, it updates its own counter by the received one. When the resolution is over, the largest counter value is selected as the Concurrent Constraint Checks value.

3 THE PRINCIPLE DISCSP ALGORITHMS

3.1 Asynchronous BackTracking ABT

The Asynchronous Backtracking is a DisCSP algorithm which allows agents to solve the problem asynchronously. The order priority of agents is made statically and lexicographically ($A_i$ is a higher priority than $A_j$ when $i<j$). The algorithm 1 exhibits the different procedures and functions used by an ABT agent to find a solution.

Each agent $s$ has an AgentView structure, to save the assignments of higher priority agents than itself, and a NogoodStore structure to store the nogoods sent by the lower priority agents to itself.

When the ABT protocol starts running, each agent chooses an assignment to its local variables, creates an OK message that carries its choice and sends it to the lower priority agents.

Algorithm 1 ABT algorithm

1. procedure ABT
2. myValue ← empty;
3. while ¬ end do
4. msg ← recvMsg();
5. Switch (msg.type)
6. Ok? : ProcessInfo(msg);
7. ngd : ResolveConflict(msg);
8. stp : end ← true;
9. adl : SetLink(msg);
10. end while
11. end procedure

12. procedure CheckAgentView(msg)
13. if ¬ consistent(myValue, myAgentView) then
14. myValue ← ChooseValue();
15. if myValue then
16. foreach child ∈ Γ*(s) do
17. sendMsg : Ok? (child, myValue);
18. end for
19. else Backtrack();
20. end if
21. end if
22. end procedure

23. procedure ProcessInfo(msg)
24. Update(myAgentView, msg.Assig);
25. CheckAgentView();
26. end procedure

27. procedure ResolveConflict(msg)
28. if Coherent(msg.Nogood, Γ*(s) ∪ {s}) then
29. CheckAddLink(msg);
30. add(msg.Nogood, myNogoodStore);
31. myValue ← empty;
32. CheckAgentView();
33. elseif msg.sender ∈ Γ*(s) ∧ Coherent(msg.Nogood, self) then
34. SendMsg : Ok? (msg.Sender, myValue);
35. end if
36. end procedure

37. procedure SetLink(msg)
38. add(msg.sender, Γ*(s));
39. sendMsg.ok?(msg.sender, myValue);
40. end procedure

41. procedure CheckAddLink(msg)
42. foreach var ∈ lhs(msg.Nogood) do
43. if var ≠ Γ*(s) then
44. sendMsg.adl(var, self);
45. add(var, Γ*(s));
46. Update(myAgentView, var ← varValue);
47. end if
48. end for
49. end procedure

50. procedure BackTrack
51. newNogood ← solve(myNogoodStore);
52. if newNogood = empty then
53. end ← true;
54. sendMsg.stp(system);
55. elseif sendMsg.ngd(newNogood);
56. Update(myAgentView, rhs(newNogood) = unknown);
57. CheckAgentView();
58. end if
59. end procedure

60. function ChooseValue()
61. foreach v ∈ D(s) not eliminated by myNogoodStore do
62. if consistent(v, myAgentView) then return (v);
63. else
64. add($x_j = v_\text{value}$ to self = $s$'s nogood);
65. $\Rightarrow$ v is inconsistent with $x_j$'s value
66. end if
67. end for
68. end function

69. procedure Update(myAgentView, newAssig)
70. add(newAssig, myAgentView);
71. foreach ng ∈ myNogoodStore do
72. if ¬Coherent(lhs(ng), myAgentView) then
73. remove(ng, myNogoodStore);
74. end if
75. end for
76. end procedure

77. procedure Coherent(nogood, agents)
78. foreach var ∈ nogood ∪ agents do
79. if nogood[var] ≠ myAgentView[var] then
80. return false;
81. end if
82. end for
83. return true;
84. end procedure
After receiving the OK message, the receiver updates its AgentView with the new received values and checks if its assignment is still consistent with the content of its updated AgentView.

**Definition 3.1 (Consistent).** An assignment is consistent with other assignments if all the constraints that link this assignment with the others are satisfied.

If this is not the case, it browses its domain, in order to find an assignment which satisfies the consistency condition.

For each tested value, if it does not satisfy a certain number of constraints with some values from the AgentView, a nogood is created and stored in the NogoodStore. This nogood contains in its lhs the values which block the current tested value and the checked value in its rhs. The nogood is used as a justification of why the tested value is not chosen.

If the whole domain is scanned without finding any consistent value, the agent generates a nogood from the stored nogoods (justifications). The rhs of the resultant nogood is the value of the lowest priority agent, and the others values are put into the lhs. The agent self sends this nogood to the lowest priority agent (whose value exists in the rhs). This nogood is used to say to the receiver that as long as the other agents have the values that exist in the lhs, you should change this value which is in the rhs.

The receiver stores the nogood in its nogoodStore, and tries to find a new consistent value which is not removed by one of the stored nogoods in the NogoodStore, taking into account (as after receiving a new OK message) the AgentView values. In the same way as the OK message, if the agent finds the good assignment it sends it via an OK message, otherwise, it sends a nogood message. Without having forgotten that: after receiving a new OK message, even the nogoodStore is updated, by removing the obsolete nogoods (whose lhs contains, at least, a different value that becomes different).

A solution is found when a silence is detected. If a null nogood message is generated by the highest priority agent the problem is unsolvable.

### 3.2 Nogood Based Asynchronous Forward Checking AFC-ng

As ABT, AFC-ng (nogood based Asynchronous Foward Checking) [13] is a DisCSP algorithm. It is based on another algorithm called AFC which has been proposed in [10] by Meisels and Zivan. Actually, this algorithm is hybrid (it is asynchronous and synchronous in the same time).

The synchronous part of AFC-ng is seen when agents try to assign its variables. At one point, just a single agent can affect its variables, it is the one that receives the CPA message (for Current Partial Assignment). It is a data structure which represents the research process state and contains a partial assignment of the DisCSP problem variables, so that each agent tries to extend, until it contains the assignments of all existing variables (all agents), so as the constraints are satisfied. And the asynchronous part is highlighted when an agent spreads its value choices to the lower priority agents. The receivers revise its domains asynchronously.

The algorithm 2 shows the AFC-ng resolution stages. When the protocol starts, it is the highest priority agent which generates the CPA structure, puts in its instantiation and sends it to lower priority agents via CPA message.

#### Algorithm 2 AFC-ng algorithm

1. procedure AFC-ng
2. end ← false; AgentView.Consistent ← true;
3. if \( A_i = 1A \) then
4. Assign();
5. end if
6. while ¬end do
7. msg ← getMsg();
8. Switch (msg.type) do
9. cpa : ProcessCPA(msg);
10. ngd : ProcessNogood(msg);
11. stp : end ← true;
12. end while
13. end procedure
14. procedure INITAGENTVIEW
15. for each \( x_j \in x_d \) do
16. AgentView[j] ← \{(x_j, empty, 0)\};
17. end for
18. end procedure
19. procedure ASSIGN
20. if \( D(x_i) \neq 0 \) then
21. \( v_i \leftarrow \text{ChooseValue()} \);
22. \( t_i \leftarrow t_i + 1 \);
23. CPA ← \{AgentView ∪ (x_i, v_i, t_i)\};
24. else
25. Backtrack();
26. end if
27. end procedure
28. procedure SENDCPA(CPA)
29. if size(CPA) = n then
30. Report SOLUTION;
31. end ← true;
32. else
33. for each \( (x_k, v_k, t_k) \in CPA \) do
34. sendMsg.cpa(CPA) to \( A_k \);
35. end for
36. end if
37. end procedure
38. procedure PROCESSCPA(CPA)
39. if msg.CPA is stronger than AgentView then
40. UpdateAgentView(msg.CPA);
41. AgentView.Consistent ← true;
42. Revise();
43. if \( D(x_i) = 0 \) then
44. Backtrack();
45. else
46. CheckAssign(msg.Sender);
47. end if
48. end if
49. end procedure
50. procedure CHECKASSIGN(sender)
51. if predecessor(\( A_i = \text{sender} \) then
52. Assign();
53. end if
54. end procedure
chooses a value for its variable (the value that is not eliminated by priority agent. Otherwise, if it is the predecessor of the sender, it if yes, it generates a nogood (as ABT) and sends it to the lowest domain becomes empty, then it revises its domain, by adding a justifying nogood, for each inconsistent value. After which, it checks if the domain becomes empty, if yes, it generates a nogood (as ABT) and sends it to the lowest priority agent. Otherwise, if it is the predecessor of the sender, it chooses a value for its variable (the value that is not eliminated by nogood) adds it to the CPA and propagates the message to the lower priority agents.

The nogood message is treated in the same way as in the ABT nogood. The end of protocol can be detected without using another external algorithm. A solution is found when the lowest priority agent adds its assignment to the CPA (the size of the CPA structure is equal to the number of agents). The insolvency is detected by one of the existing agents. When its domain becomes empty and there is no nogood justifying this failure, it stops the resolution, declaring that the problem is insolvable.

4 NOGOOD LEARNING IN THE ALGORITHMS ABT AND AFC-NG

The subsections 3.1 and 3.2 show that there are several common features between ABT and AFC-ng including:

1. The NogoodStore content: the two algorithms have the same NogoodStore structure with the same contents;
2. The AgentView content: the two algorithms have the same AgentView structure. Even if the AFC-ng AgentView contains the counters (the ABT does not use the counters), but it also contains the higher priority agent assignments as it is done by the ABT AgentView;
3. The nogood structure: the two nogoods are generated in the same way.

These common points are the basis to launch the two algorithms in the same DisCSP problem, in order to collaborate with each other and find the solution, either with less message exchanged and fewer tested constraints or else in a minimum of time.

In order to ensure this, we did it in two different ways:

4.1 The first method of ABT & AFC-ng Learning (Learning-1)

In the first method, the two algorithms are launched at the same time. All agents launch the ABT algorithm and just the initial agent which starts the AFC-ng algorithm.

In addition to stp message, each agent can receive and send four other types of messages: CPA, ndg_AFCng (nogood message sent using AFCng algorithm), OK!, ndg_ABT (nogood message sent using ABT algorithm) and stp message.

For each received message the associated procedure is applied as the original algorithm.

The algorithm 3 shows only the procedures and functions that have been changed.

In addition to their structures AgentView_ABT, AgentView_AFCng, NogoodStore_ABT and NogoodStore_AFCng, each agent creates two new structures SentNogoodsByABT to store the nogoods sent by ABT algorithm and SentNogoodsByAFCng which will contain the nogoods sent by AFCng.

Then, when an agent creates and sends a new nogood by ABT(backtrack_ABT procedure), it stores it in the new structure SentNogoodsByABT.

The same thing when it generates a new nogood using AFC_ng (backtrack_AFCng procedure), it stores it in the SentNogoodsByAFCng structure.

```
55: procedure Backtrack
56:   ndg ← solve(myNogoodStore);
57:   if ndg = empty then
58:     Report FAILURE;
59:     end if
60:   else
61:     for k = j+1 to i-1 do
62:       AgentView[k].value ← empty;
63:       end for
64:     for each (nogood ∈ NogoodStore) do
65:       if ¬Compatible(nogood,AgentView) then
66:         remove (nogood, NogoodStore);
67:       end if
68:     end for
69:     AgentView.Consistent ← false;
70:     v_i ← empty;
71:     sendMsg:ngd(ng) to A_j;
72:     end if
73:   end procedure

74: procedure ProcessNogood(msg)
75:   if Compatible(msg,nogood,AgentView) then
76:     add msg to NogoodStore;
77:     "according to the HPLV[]"
78:   if rhs(msg,nogood).value = v_i then
79:     v_i ← empty;
80:     Assign();
81:   end if
82: end if
83: end procedure

84: procedure UpdateAgentView(CPA)
85:   AgentView ← CPA;
86: end if
87: for each (ng ∈ NogoodStore) do
88:   if ¬ Compatible(ng, AgentView) then
89:     remove(ng, myNogoodStore);
90:   end if
91: end procedure

92: procedure Revise
93:   for each (v ∈ D^0(x_i)) do
94:     if v is ruled out by AgentView then
95:       Store the best nogood for v;
96:       "according to the HPLV"
97:     end if
98:   end for
99: end procedure

Each receiver checks if the received message is up to date (using a counter for each agent). If so, it updates its AgentView and NogoodStore. It replaces the AgentView by the received CPA and removes the obsolete nogoods that exists in the NogoodStore. Then it revises its domain, by adding a justifying nogood, for each inconsistent value. After which, it checks if the domain becomes empty, if yes, it generates a nogood (as ABT) and sends it to the lowest priority agent. Otherwise, if it is the predecessor of the sender, it chooses a value for its variable (the value that is not eliminated by
Algorithm 3 Learning-1

1. procedure ABT-AFCng sharing 1
2. end ← false; AgentView_AFCng.Consistent ← true;
3. if $A_j = 1A$ then
4. Assign();
5. end if
6. CheckAgentView();
7. while ¬ end do
8. msg ← getMsg();
9. Switch (msg.type) do
10. cpa : ProcessCPA(msg);
11. ngd_AFCng : ProcessNogood(msg);
12. Ok? : ProcessInfo(msg);
13. ngd_AB : ResolveConflict(msg);
14. sta : end ← true;
15. end while
16. end procedure
17. procedure Backtrack_AFCng
18. ngd ← solve(myNogoodStore);
19. if ngd = empty then
20. the same AFC-ng Treatment;
21. else
22. the same AFC-ng Treatment;
23. add(ngd, SentNogoodsByAFCng);
24. end if
25. end procedure
26. procedure Backtrack_AB
27. newNogood ← solve(myNogoodStore);
28. if newNogood = empty then
29. the same ABT treatment;
30. else
31. the same ABT treatment;
32. add(newNogood, SentNogoodsByABT)
33. end if
34. end procedure
35. function ChooseValue_AB()
36. for each ngd ∈ SentNogoodsByABT do
37. if (istheAgentViewAlreadyInconsistent(ngd)) then
38. ClearNogoodStore_AB;
39. for each ($v$ ∈ Dis($x_j$)) do
40. nogood ← ngd.lhs/ngd.rhs $x_j =v$;
41. add (nogood, nogoodStore_AB);
42. end for
43. return null;
44. end if
45. end for
46. for each $v$ ∈ $D$ not eliminated by NogoodStore_AB do
47. if $v$ is eliminated by a coherent nogood from NogoodStore_AFCng then
48. addNgd, NogoodStore_AB;
49. else
50. if consistent($v$, myAgentView) then return ($v$);
51. else
52. add($x_j = v$) $\rightarrow$ self $\neq v$, NogoodStore_AB;
53. $v$ is inconsistent with $x_j$'s value
54. end if
55. end if
56. end for
57. return (empty)
58. end function
59. procedure Revise
60. for each ngd ∈ SentNogoodsByABT do
61. if (istheAgentViewAlreadyInconsistent(ngd)) then
62. for each ($v$ ∈ Dis($x_j$)) do
63. ngd ← ngd.lhs/ngd.rhs $x_j = v$;
64. if $v$ is eliminated by nogoodStore_AFCng then
65. keep the best nogood between the eliminating no-
66. good and ngd;
67. else
68. add (ngd, nogoodStore_AFCng);
69. end if
70. end for
71. return null;
72. end if
73. for each ($v$ ∈ $D$($x_j$)) do
74. if ($v$ is ruled out by AgentView or eliminated by
75. nogoodStore_AB) then
76. Store the best nogood for $v$;
77. end if
78. end for
79. end procedure

So, When an agent tries to choose a value using the ABT or re-
vising the domain using the AFC-ng algorithm (ChooseValue_AB),
Revise()), Firstly, it checks if there is a sent nogood in the other algo-
rum and the whole elements of the latter exist in the AgentView
of the current algorithm.

If so, ChooseValue_AB procedure clears the NogoodStore_AB,
gen erates a new nogood whose lhs is the nogood components (lhs
and rhs), browses the whole domain, completes the nogood rhs
by the current tested value and adds it to the current algorithm
NogoodStore (NogoodStore_AB). Then it return null, saying that
there is no consistent value, without testing the constraints, because
it is already tested by the other algorithm (AFCng).

After this, if there is no nogood sent by the other algorithm, the
procedure continue its treatments. It browses the domain, value
by value, tests if the value tested is not eliminated by the ABT
NogoodStore (NogoodStore_AB) (the same as ABT ChooseValue
procedure without sharing nogoods), if so, it checks if there is a
nogood in the other nogoodStore (NogoodStore_AFCng) which
is compatible with the AgentView_AB and eliminates the tested
value. If so, it adds the found nogood in the NogoodStore_AB.
Otherwise, it tests if the value is consistent with AgentView_AB,
if so, it returns the value, otherwise, it adds the justifying nogood.

For the Revise() method, it tests if there is a nogood which exists
in the SentNogoodsABt structure and which is coherent with the
AgentView_AFCng.

If so, it browses the domain, in order the construct a new nogood
whose the lhs is the lhs and rhs conjunction of the found nogood
and the rhs is the tested value. If the tested value is already removed
by a nogood (in NogoodStore_AFCng), it keeps the better nogood
(the constructed nogood or the found nogood), according to the
HLPV method. Otherwise, it adds the generated nogood to the
NogoodStore_AFCng.
If there is no sent nogood by the ABT algorithm, the Revise() procedure browses the domain. It checks not only if the value is inconsistent with the AgentView_AFCng but also if the value is eliminated by nogoodStore_AB, if so, it keeps the best nogood using the HPLV method.

4.2 The second method of ABT & AFC-ng Learning (Learning-2)

The second method follows the same methodology as the first. The only difference is highlighted when the ABT or AFC-ng algorithm generates a no empty nogood. Before sending it and adding it into SentNogoods structures (SentNogoodsByABT and SentNogoodsByAFCng), it should check if the generated nogood is not already sent by the other algorithm.

So, if an algorithm finds that its new generated nogood is already sent by the other algorithm, it stops the resolution and the other continues its resolution process.

5 EXPERIMENTAL RESULTS

In this section, we compare the algorithms ABT and AFC-ng with the two learning methods (Learning-1 and Learning-2) and also with ABT/AFC-ng_Learning-1 (i.e. taking into account just the fast reference is highlighted when the ABT or AFC-ng algorithm generates a nogood).

The assessment is made against the number of exchanged messages (# MSGs) and the Concurrent Constraint Checks (# CCCs), using disChoco platform [3].

We experiment the five algorithms in random problems. The problems are characterized by the parameters (n, d, p1, p2) where, n is the number of agents, d is the domain size, p1 is the problem density and p2 is the tightness of constraints.

Algorithm 4 Learning-2

1. procedure Backtrack_AFCng
2.     ngd ← solve(myNogoodStore);
3.     if ngd = empty then
4.         The same AFC-ng Treatment;
5.     else
6.         if ¬SentNogoodsByABT contains ngd then
7.             The same AFC-ng Treatment;
8.             add(ngd, SentNogoodsByAFCng);
9.         end if
10. end procedure

12. procedure Backtrack_AB
13.     newNogood ← solve(myNogoodStore);
14.     if newNogood = empty then
15.         the same ABT treatment;
16.     else
17.         if ¬SentNogoodsByAFCng contains ngd then
18.             the same ABT treatment;
19.             add(newNogood, SentNogoodsByABT)
20.         end if
21.     end if
22. end procedure

Figure 1: Benchmarking with \( n = 20, v = 10, p1 = 0.2, p2 \)

The ABT/AFC-ng_Learning-1 algorithm is obtained by using the Learning-1 method with the ABT and AFC-ng algorithms and computing just the number of MSGs and CCCs in ABT for ABT_Learning-1 and in AFC-ng for the AFC-ng_Learning-1 algorithm, and keep the results of the fast one.

All generated problems have \( n = 20 \) and \( d \). We evaluate the algorithms into two types of problems. With low density value \( p1 = 0.2 \) and with high density value \( p1 = 0.7 \). For the two kinds of problems, we variate the tightness \( p2 \) from 0.1 to 0.9 by 0.05 as a step.

Figure 1 shows the number of exchanged messages and Concurrent Constraint Checks by the five methods, for sparse graphs (20, 10, 0.2, p2). The experimentation results show that Learning-1 and Learning-2 methods are between ABT and AFC-ng, knowing that we compute the number of exchanged message by the two algorithms ABT and AFC-ng, even if, is just one of its which finds the solution (or detect the insolvability). These results show that we could learn the informations between the algorithms, because otherwise, if we do not any nogood learning, the number of exchanged messages will be the sum of the two algorithms, which will be very large, so as the ABT and AFC exchanged message numbers will be Negligible in front of the sum.

For CCCs, we are at least like the better of them.

So when we compute the number of exchanged messages by only the algorithm which finds the solution the first (because it can be find by the ABT or the AFC-ng, according to the most fast algorithm), we obtain very important results. The ABT/AFCng_Learning-1 exchanges less messages and tests less constraints concurrently.
The choice of computing the MSGs and CCCs of only the algorithm which finds the solution is not obsolete. Because, the machine learning principle is serving to store the exchanged nogood messages following each resolution (by an algorithm). And then use the stored messages, to learn and resolve the DisCSP fast. In this case we will not compute the number of MSGs or CCCs of the training step, because it is already done.

The second figure shows the evaluation results (number of MSGs, and CCCs) for dense graphs $(20, 10, 0.7, p_2)$. As the previous figure, it represents the performance of the five algorithms, for $p_2$, ranging from 0.1 to 0.9, except the Learning-2 method which we evaluate just for 0.1 to 0.25. The latter is better than the ABT, AFC-ng and the Learning-1 methods, because, when an algorithm finds that the other has already send the same nogood message, it stops the resolutions.

This is feasible, when problems are simple, but for complex problems, it is less likely, to find the same nogood with the two algorithms. The Learning-1 is so the more useful.

The results make clear that the Learning (Learning-1 or Learning-2) saves us the exchanged messages and the tested constraints, even if we compute the used resources by the two algorithms. The importance of learning becomes more legible, when we plot the ABT/AFC-ng_Learning-2.

6 CONCLUSIONS

In this paper, we have proposed two methods Learning-1 and Learning-2, that allow to at least two DisCSP algorithms at the same DisCSP problem, and share the nogoods content between the participant algorithms. Assessments prove that, by learning nogoods, we save exchanged messages and tested constraints. The results give us the idea, for the future work, to launch the first algorithm (ABT, AFC-ng or another DisCSP algorithm), save the nogoods and recuperate the results. Then run the second algorithm, learning from the saved nogoods.

REFERENCES