Nogood Learning in DisCSP Algorithms

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

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

El Houssine BOUYAKHF
LIMIARF Laboratory
Rabat, Morocco
bouyakhf@mtds.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:
Ghizlane EL KHATTABI, Imade BENELALLAM, and El Houssine BOUYAKHF. 1997. Nogood Learning in DisCSP Algorithms. In *Proceedings of International Conference on Applied Research in Computer Science and Engineering (ICAR'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.475/123_4

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 planification problems, ***ii)*** Multi-agent system[6], that represent a set of agents (or a set of computer systems) that colaborate 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 affectaion of rules is not definitive, it can be changed during the resolution process. The contribution details will be described more accuratly 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 ϕ 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 affectation that can not be extended to a solution. In DisCSP algorithms, the nogood takes the form $x_i = v_i \wedge x_j = v_j \wedge \ldots \rightarrow x_k = v_k$ 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 → symbol is called the left-hand-side (lhs) and the other is called the right-hand-side (rhs).

Definition 2.6 (Conccurent 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 *self* has an AgentView structure, to save the assignments of higher priority agents than self, and a NogoodStore structure to store the nogoods sent by the lower priority agents to self.

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

```
procedure ABT myvalue ← empty;end ← false;
CheckAgentView();
    while ¬$end$ do
        msg ← getMsg();
        Switch (msg.type)
         Ok? : ProcessInfo($msg$);
         ngd : ResolveConflict($msg$);
         stp : $end \leftarrow true$;
         adl : SetLink($msg$);
    end while
end procedure

procedure CheckAgentView($msg$)
    if ¬$consistent(myValue; myAgentView)$ then
        $myValue \leftarrow ChooseValue()$;
        if myValue then
            for each child ∈ $\Gamma^+(self)$ do
                sendMsg : Ok? (child, myValue);
            end for
        else Backtrack();
        end if
    end if
end procedure

procedure ProcessInfo($msg$)
    Update(myAgentView, msg.Assig);
    CheckAgentView();
end procedure

procedure ResolveConflict($msg$)
    if Coherent($msg.Nogood$, $\Gamma^-(self) \cup \{self\}$) then
        CheckAddLink(msg);
        add(msg.Nogood,myNogoodStore);
        $myValue \leftarrow empty$;
        CheckAgentView();
    elseif msg.sender ∈ $\Gamma^+(self)$ ∧ Coherent(msg.Nogood, self) then
        SendMsg : Ok? (msg.Sender, myValue);
    end if
end procedure

procedure SetLink($msg$)
    add(msg.sender, $\Gamma^+(self)$);
    sendMsg:ok?(msg.sender, myValue);
end procedure
procedure CheckAddLink(msg)
    for each $var \in$ lhs(msg.Nogood) do
        if $var \notin \Gamma^-(self)$ then
            sendMsg:adl($var$,self);
            add($var$,$\Gamma^-(self)$)
            Update (myAgentView, $var$ ← varValue);
        end if
    end for
end procedure

procedure Backtrack
    $newNogood \leftarrow solve(myNogoodStore)$;
    if $newNogood = empty$ then
        $end \leftarrow true$;
        sendMsg:stp(system);
    else
        sendMsg:ngd(newNogood);
        Update (myAgentView, rhs(newNogood) ← unknown);
        CheckAgentView();
    end if
end procedure

function chooseValue()
    for each $v \in D(self)$ not eliminated by $myNogoodStore$ do
        if consistent($v$,$myAgentView$) then return (v);
        else
            add($x_j = val_j \rightarrow self \neq v$, myNogoodStore);
                                          ▷ $v$ is inconsistent with $x_j$'s value
        end if
    end forreturn ($empty$)
end function

procedure Update($myAgentView$, $newAssig$)
    add(newAssig, myAgentView);
    for each $ng \in myNogoodStore$ do
        if ¬$Coherent(lhs(ng), myAgentView)$ then
            remove(ng, myNogoodStore);
        end if
    end for
end procedure
procedure Coherent($nogood$, $agents$)
    for each $var \in nogood \cup agents$ do
        if $nogood[var] \neq myAgentView[var]$ then
      return false;
        end if
    end for
      return true;
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 affectation 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 an 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

```
procedure AFC-NG
    end ← false; AgentView.Consistent ← true ;
    if $A_i$ = IA then
        Assign();
    end if
    while $\neg end$ do
        msg ← getMsg();
        Switch (msg.type) do
         cpa : ProcessCPA($msg$);
         ngd : ProcessNogood($msg$);
         stp : $end \leftarrow true$;
    end while
end procedure

procedure InitAgentView
    for each $x_j < x_i$ do
        $AgentView[j] \leftarrow \{(x_j, empty, 0)\}$;
    end for
end procedure
procedure Assign
    if $D(x_i) \neq \emptyset$ then
        $v_i$ ←ChooseValue();
        $t_i \leftarrow t_i + 1$;
        CPA← {AgentView ∪ $(x_i, v_i, t_i)$};
    else
        Backtrack();
    end if
end procedure

procedure SendCPA(CPA)
    if size(CPA) = $n$ then
        Report $SOLUTION$;
        $end \leftarrow true$;
    else
        for each $(x_k \in \Gamma^+(x_i))$ do
            sendMsg:cpa(CPA) to $A_k$;
        end for
    end if
end procedure

procedure ProcessCPA(CPA)
    if msg.CPA is stronger than AgentView then
        UpdateAgentView(msg.CPA);
        AgentView.Consistent ← true;
        Revise();
        if $D(x_i) = \emptyset$ then
            Backtrack();
        else
            CheckAssign(msg.Sender);
        end if
    end if
end procedure

procedure CheckAssign(sender)
    if predecessor($A_i = sender$) then
        Assign();
    end if
end procedure
```

```
procedure Backtrack
    $ngd \leftarrow solve(myNogoodStore)$;
    if $ngd = empty$ then
        Report $FAILURE$;
        $end \leftarrow true$;
    else                                    ▷ Let $x_j$ denote the variable in rhs(ng)
        for k =j+1 to i-1 do
            AgentView[k].value ← empty;
        end for
        for each (nogood ∈ NogoodStore) do
            if (¬Compatible(nogood,AgentView) ∨ $x_j \in nogood$) then
                remove (nogood, NogoodStore);
            end if
        end for
        AgentView.Consistent ← false;
        $v_i \leftarrow empty$;
        sendMsg:ngd(ng) to $A_j$;
    end if
end procedure

procedure ProcessNogood($msg$)
    if Compatible( msg.nogood, AgentView ) then
        add(msg.nogood,NogoodStore);
                                            ▷ according to the HPLV []
        if (rhs(msg.nogood).value = $v_i$) then
            $v_i \leftarrow empty$;
            Assign();
        end if
    end if
end procedure

procedure UpdateAgentView(CPA)
    AgentView ← CPA;                        ▷ update values and tags
    for each (ng ∈ NogoodStore) do
        if ¬ Compatible(ng, AgentView) then
            remove(ng, myNogoodStore);
        end if
    end for
end procedure

procedure Revise
    for each ($v \in D^0(x_i)$) do
        if $v$ is ruled out by $AgentView$) then
            Store the best nogood for $v$;
                                            ▷ according to the HPLV
        end if
    end for
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 a 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 stucture**: 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, ngd_AFCng (nogood message sent using AFCng algorithm), Ok?, ngd_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.

Algorithm 3 Learning-1

```
procedure ABT-AFCng sharing 1
    end ← false; AgentView_AFCng.Consistent ← true ;
    if $A_i$ = IA then
        Assign();
    end if
    CheckAgentView();
    while ¬$end$ do
        msg ← getMsg();
        Switch (msg.type) do
         cpa          : ProcessCPA($msg$);
         ngd_AFCng    : ProcessNogood($msg$);
         Ok?          : ProcessInfo($msg$);
         ngd_ABT      : ResolveConflict($msg$);
         stp          : $end \leftarrow true$;
    end while
end procedure

procedure Backtrack_AFCng
    $ngd \leftarrow solve(myNogoodStore)$;
    if $ngd = empty$ then
        The same AFC-ng Treatment;
    else

        The same AFC-ng Treatment;
        add(ngd, SentNogoodsByAFCng);
    end if
end procedure

procedure Backtrack_ABT
    $newNogood \leftarrow solve(myNogoodStore)$;
    if $newNogood = empty$ then
        the same ABT treatment;
    else
        the same ABT treatment;
        add(newNogood, SentNogoodsByABT)
    end if
end procedure

function chooseValue_ABT()
    for each ngd ∈ SentNogoodsByAFCng do
        if (istheAgentViewAlreadyInconsistent(ngd) then
            Clear NogoodStore_ABT;
            for each ($v \in D(self)$) do
                nogood ← ngd.lhs∧ngd.rhs→ $x_i = v$;
                add (nogood, nogoodStore_ABT);
            end for
            return null;
        end if
    end for
    for each $v \in D$ not eliminated by $NogoodStore\_ABT$ do
        if $v$ is eliminated by a coherent nogood from $NogoodStore\_AFCng$ then
            add(ngd, NogoodStore_ABT);
        else
            if consistent($v$,$myAgentView$) then return (v);
            else
                add($x_j = val_j \rightarrow self \neq v$, NogoodStore_ABT);
                                        ▷ $v$ is inconsistent with $x_j$'s value
            end if
        end if
    end for return ($empty$)
end function

procedure Revise
    for each ngd ∈ SentNogoodsByABT do
        if (istheAgentViewAlreadyInconsistent(ngd) then
            for each ($v \in D(self)$) do
                ngd ← ngd.lhs∧ngd.rhs→ $x_i = v$;
                if $v$ is eliminated by nogoodStore_AFCng then
                    keep the best nogood between the eliminating nogood and ngd;
                                        ▷ According to the HPLV
                else
                    add (ngd, nogoodStore_AFCng;
                end if
            end forreturn null;
        end if
    end for
    for each ($v \in D^0(x_i)$) do
        if ($v$ is ruled out by $AgentView$ or eliminated by $nogoodStore\_ABT$ ) then
            Store the best nogood for $v$;
                                        ▷ according to the HPLV[7]
        end if
    end for
end procedure
```

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

If so, ChooseValue_ABT procedure clears the NogoodStore_ABT, generates 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_ABT). 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_ABT) (the same as ABT ChooseValue procedue without sharing nogoods), if so, it checks if there is a nogood in the other nogoodStore (NogoodStore_AFCng) which is compatible with the AgentView_ABT and eliminates the tested value. If so, it adds the found nogood in the NogoodStore_ABT. Otherwise, it tests if the value is consistent with AgentView_ABT, 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 SentNogoodByABT 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 HPLV 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_ABT, 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 algorithm which finds the solution the first, using the Learning-1 method).

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, p_1, p_2) where, n is the number of agents, d is the domain size, p_1 is the problem density and p_2 is the tightness of constraints.

Algorithm 4 Learning-2

```
procedure BACKTRACK_AFCNG
    $ngd \leftarrow solve(myNogoodStore)$;
    if $ngd = empty$ then
        The same AFC-ng Treatment;
    else
        if ¬ SentNogoodsByABT contains ngd then
            The same AFC-ng Treatment;
            add(ngd, SentNogoodsByAFCng);
        end if
    end if
end procedure

procedure BACKTRACK_ABT
    $newNogood \leftarrow solve(myNogoodStore)$;
    if $newNogood = empty$ then
        the same ABT treatment;
    else
        if ¬ SentNogoodsByAFCng contains ngd then
            the same ABT treatment;
            add(newNogood, SentNogoodsByABT)
        end if
    end if
end procedure
```

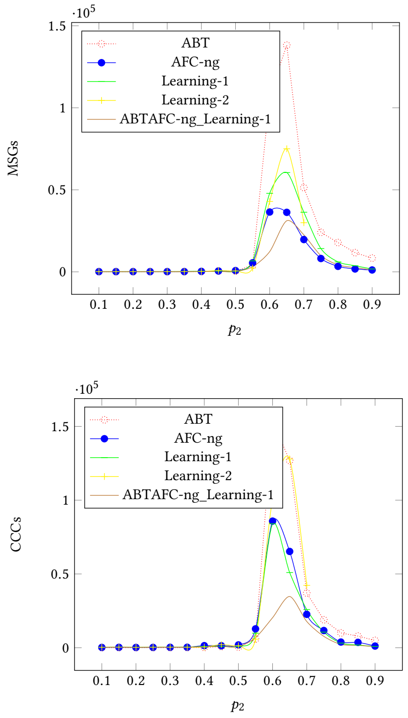

Figure 1: Benchmarking with $n = 20$, $v = 10$, $P_1 = 0.2$, p_2)

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 $p_1 = 0.2$ and with high density value $p_1 = 0.7$.For the two kinds of problems, we variate the tightness p_2 from 0.1 to 0.9 by 0.05 as a step.

Figure 1 shows the number of exchanged messages and Conccurent Constraint Checks by the five methods, for sparse graphs (20, 10, 0.2,p_2). 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.

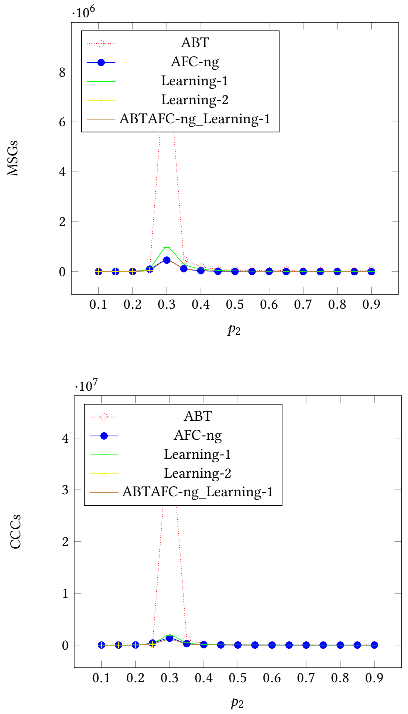

Figure 2: Benchmarking with $n = 20$, $v = 10$, $P_1 = 0.7$, p_2)

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

[1] David W Aha, Dennis Kibler, and Marc K Albert. 1991. Instance-based learning algorithms. *Machine learning* 6, 1 (1991), 37–66.
[2] Ramón Béjar, Carmel Domshlak, Cèsar Fernández, Carla Gomes, Bhaskar Krishnamachari, Bart Selman, and Magda Valls. 2005. Sensor networks and distributed CSP: communication, computation and complexity. *Artificial Intelligence* 161, 1-2 (2005), 117–147.
[3] Imade Benelallam, Zakarya Erraji, Ghizlaneg Elkhattabi, Jaouad Ait Haddou, and El-Houssine Bouyakhf. 2015. JChoc DisSolver-Bridging the Gap Between Simulation and Realistic Use.. In *ICAART (1)*. 66–74.
[4] Imade Benelallam, Zakarya Erraji, Ghizlane EL Khattabi, and El Houssine Bouyakhf. 2015. Dynamic JChoc: A Distributed Constraints Reasoning Platform for Dynamically Changing Environments. In *International Conference on Agents and Artificial Intelligence*. Springer, 20–36.
[5] Christian Bessière, Arnold Maestre, Ismel Brito, and Pedro Meseguer. 2005. Asynchronous backtracking without adding links: a new member in the ABT family. *Artificial Intelligence* 161, 1-2 (2005), 7–24.
[6] Jacques Ferber. 1999. *Multi-agent systems: an introduction to distributed artificial intelligence*. Vol. 1. Addison-Wesley Reading.
[7] Katsutoshi Hirayama and Makoto Yokoo. 2000. The effect of nogood learning in distributed constraint satisfaction. In *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*. IEEE, 169–177.
[8] Hyuckchul Jung, Milind Tambe, and Shriniwas Kulkarni. 2001. Argumentation as distributed constraint satisfaction: Applications and results. In *Proceedings of the fifth international conference on Autonomous agents*. ACM, 324–331.
[9] Rajiv T Maheswaran, Milind Tambe, Emma Bowring, Jonathan P Pearce, and Pradeep Varakantham. 2004. Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*. IEEE Computer Society, 310–317.
[10] Amnon Meisels and Roie Zivan. 2007. Asynchronous forward-checking for DisCSPs. *Constraints* 12, 1 (2007), 131–150.
[11] Adrian Petcu and Boi Faltings. 2004. A value ordering heuristic for local search in distributed resource allocation. In *International Workshop on Constraint Solving and Constraint Logic Programming*. Springer, 86–97.
[12] Francesca Rossi, Peter Van Beek, and Toby Walsh. 2006. *Handbook of constraint programming*. Elsevier.
[13] Mohamed Wahbi, Redouane Ezzahir, Christian Bessiere, and El Houssine Bouyakhf. 2013. Nogood-based asynchronous forward checking algorithms. *Constraints* 18, 3 (2013), 404–433.
[14] Richard J Wallace and Eugene C Freuder. 2002. Constraint-based multi-agent meeting scheduling: Effects of agent heterogeneity on performance and privacy loss. (2002).
[15] Makoto Yokoo. 1995. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 88–102.
[16] Makoto Yokoo, Toru Ishida, Edmund H Durfee, and Kazuhiro Kuwabara. 1992. Distributed constraint satisfaction for formalizing distributed problem solving. In *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*. IEEE, 614–621.