Enforcing Full Arc Consistency in Asynchronous Forward Bounding Algorithm
Rachid Adrdor, and Lahcen Koutti

Abstract—The AFB\textsubscript{BJ}\textsuperscript{+}\_DAC\textsuperscript{*} is the latest variant of asynchronous forward bounding algorithms used to solve Distributed Constraint Optimization Problems (DCOPs). It uses Directional Arc Consistency (DAC\textsuperscript{*}) to remove, from domains of a given DCOP, values that do not belong to its optimal solution. However, in some cases, DAC\textsuperscript{*} does not remove all suboptimal values, which causes more unnecessary research to reach the optimal solution.

In this paper, to clear more and more suboptimal values from its domain, we suggest in this paper to upgrade DAC\textsuperscript{*} to the next higher level, which is Full Directional Arc Consistency (FDAC\textsuperscript{*}). The new algorithm is called AFB\textsubscript{BJ}\textsuperscript{+}\_FDAC\textsuperscript{*} and allows agents to perform AC* multiple times and thus remove more suboptimal values from their domains.

Our experiments on different benchmarks show the superiority of AFB\textsubscript{BJ}\textsuperscript{+}\_FDAC\textsuperscript{*} algorithm in terms of communication load and computation effort.

This paper is made up of three main sections. Section II presents an overview of DCOPs, soft arc consistency rules, AFB\textsubscript{BJ}\textsuperscript{+}\_AC* algorithm, and AFB\textsubscript{BJ}\textsuperscript{+}\_DAC\textsuperscript{*} algorithm. Section III gives a description of AFB\textsubscript{BJ}\textsuperscript{+}\_FDAC\textsuperscript{*} algorithm. Section IV exposes the experiments carried out on some benchmarks.

I. INTRODUCTION

There are a large number of multi-agent problems that can be modeled as DCOPs such as meetings scheduling [17], sensor networks [7], [18], and so on. In a DCOP, variables, domains, and constraints are distributed among a set of agents. Each agent has full control over a subset of variables and constraints that involve them [11]. A DCOP is solved in a distributed manner via an algorithm allowing the agents to cooperate and coordinate with each other to find a solution with a minimal cost. A solution of a DCOP is a set of value assignments, each representing the value assigned to one of the variables of that DCOP. Algorithms with various search strategies have been suggested to solve DCOPs [9], [10]. Among them, there are Adopt [19], BnB-Adopt [25], BnB-Adopt\textsuperscript{+} [13], SyncBB [15], AFB [11], [21], AFB\textsubscript{BJ}\textsuperscript{+} [23], AFB\textsubscript{BJ}\textsuperscript{+}\_AC* [1]-[3], AFB\textsubscript{BJ}\textsuperscript{+}\_DAC\textsuperscript{*} [4], [5], etc.

In AFB\textsubscript{BJ}\textsuperscript{+}\_DAC\textsuperscript{*}, to find the optimal solution to a given problem, the agents synchronously exchange a current partial assignment (CPA) containing their assignments. During this process, and to reduce the number of exchanges, each agent uses directional arc consistency (DAC\textsuperscript{*}) to remove any suboptimal values in its domain. The positive behavior of DAC\textsuperscript{*} depends closely on DCOP to be solved in terms of its constraints and costs. This is what sometimes prevents DAC\textsuperscript{*} from behaving better in AFB\textsubscript{BJ}\textsuperscript{+}\_DAC\textsuperscript{*} algorithm. This often occurs in DCOPs where the constraints are sparse or they are dense but most of their costs are zero. For that, we suggest in this paper to upgrade DAC\textsuperscript{*} to the next higher level, which is Full Directional Arc Consistency (FDAC\textsuperscript{*}).

This often occurs in DCOPs where the constraints are sparse or they are dense but most of their costs are zero. For that, we suggest in this paper to upgrade DAC\textsuperscript{*} to the next higher level, which is Full Directional Arc Consistency (FDAC\textsuperscript{*}). The new algorithm is called AFB\textsubscript{BJ}\textsuperscript{+}\_FDAC\textsuperscript{*} and allows agents to perform AC* multiple times and thus remove more suboptimal values from their domains.

Our experiments on different benchmarks show the superiority of AFB\textsubscript{BJ}\textsuperscript{+}\_FDAC\textsuperscript{*} algorithm in terms of communication load and computation effort.

This paper is made up of three main sections. Section II presents an overview of DCOPs, soft arc consistency rules, AFB\textsubscript{BJ}\textsuperscript{+}\_AC* algorithm, and AFB\textsubscript{BJ}\textsuperscript{+}\_DAC\textsuperscript{*} algorithm. Section III gives a description of AFB\textsubscript{BJ}\textsuperscript{+}\_FDAC\textsuperscript{*} algorithm. Section IV exposes the experiments carried out on some benchmarks.

II. BACKGROUND

A. Distributed Constraint Optimization Problem (DCOP)

A DCOP [12] is defined by 4 sets, set of agents $\mathcal{A} = \{A_1, A_2, \ldots, A_k\}$, set of variables $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$, set of domains $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$, each $D_i$ is the possible values of $x_i$ in $\mathcal{X}$, and set of soft constraints $\mathcal{C} = \{C_{ij} : D_i \times D_j \rightarrow R^+\} \cup \{C_i : D_i \rightarrow R^+\}$. In a DCOP, each agent is fully responsible for a subset of variables and the constraints that involve them.

In this paper, while maintaining the generality, we only consider DCOPs in which each agent is responsible for a single variable and that every two variables, at most, are linked by a constraint (i.e., unary or binary constraint) [20].

We consider these notations: $A_j$ is an agent, where $j$ is its level or rank in the default ordering. $(x_j, v_j)$ is an assignment of $A_j$, where $v_j \in D_j$ and $x_j \in \mathcal{X}$. $C_{ij}$ is a binary constraint between $x_i$ and $x_j$. $C_{ij}^{\text{orig}}$ is an identical copy of the $C_{ij}$ constraint, used in AC\textsuperscript{*} process. $C_i$ is a unary constraint on $x_j$. $C_0$ is the global zero-arity constraint that represents a lower bound of any solution of a given DCOP. $C_{ij}$ is the local zero-arity constraint that represents the contribution value of $A_j$ in $C_0$ (i.e., $C_0 = \sum_{A_j \in \mathcal{A}} C_{ij}$). $UB_j$ is the cost of the optimal solution reached so far. $\{A_1, A_2, \ldots, A_n\}$ is the lexicographic ordering of agents (the default ordering). $\Gamma(x_j) = \{\Gamma^- : x_i \in \mathcal{X} | C_{ij} \in C, i < j\} \cup \{\Gamma^+ : x_i \in \mathcal{X} | C_{ij} \in C, i > j\}$ is the set of neighbors of $A_j$. $\Gamma^-$ (resp. $\Gamma^+$) is the set of neighbors with a higher priority (resp. with a lower priority). $Y = Y_j = \{(x_1, v_1), \ldots, (x_j, v_j)\}$ is a current partial assignment (CPA). $v_j^\star$ is the optimal value of $A_j$. $lb_k[\star](Y_j)$ are the lower bounds of a lower neighbor $A_k$ obtained for $Y_j$. $GC$ (resp. $GC^\star$) are the guaranteed costs of $Y$ (resp. in AC\textsuperscript{*}). $DVals$ is a list of $n$ arrays.
containing deleted values. Each array, $\text{DVals}[j]$, contains two elements, $\text{listVals}$ which is the list of values deleted by $A_j$ and $\text{UnvNbrs}$ which is a counter of the $A_j$ neighbors that have not yet processed $\text{listVals}$. $\text{EVals}$ is a list of arrays containing extension values.

The guaranteed cost of $Y$ (1) is the sum of $c_{ij}$ involved in $Y$.

$$\sum_{C_{ij} \in \text{C}} c_{ij}(v_i, v_j), \quad (x_i, v_i), (x_j, v_j) \in Y$$  

If a CPA $Y$ comprises a value assignment for each variable of a given DCOP, then it is called a complete assignment (i.e., a solution). This solution is said to be optimal (2) when the sum of all the constraint costs that it implies is minimal.

$$Y^* = \arg\min_Y \{GC(Y) \mid \text{var}(Y) = \mathcal{X}\}$$

Fig. 1 shows an example of a DCOP in which each agent $A_i \in \mathcal{A} = \{A_1, A_2, A_3\}$ takes control of a single variable $x_i \in \mathcal{X} = \{x_1, x_2, x_3\}$, each being defined on a domain of values $D_i = \{0, 1\}$ in $\mathcal{D} = \{D_1, D_2, D_3\}$. Each pair of variables in $\mathcal{X}$ is connected by a binary constraint $C_{ij} \in \mathcal{C} = \{C_{12}, C_{13}, C_{23}\}$. The costs of the combinations of values of each constraint $C_{ij}$ are indicated in the side tables.

**B. Soft Arc Consistency Techniques**

Soft arc consistency techniques are used when solving a given problem to delete values that are not part of the optimal solution of that problem. To apply these techniques, we use a set of transformations known as equivalence preserving transformations. They allow the exchange of costs between the constraints of the problem according to three manners that are a binary projection, a unary projection, and an extension.

The binary projection (P. 2) is an operation that subtracts, for a value $v_i$ of $D_i$, the smallest cost $\alpha$ of a binary constraint $C_{ij}$ and adds it to the unary constraint $C_i$.

The unary projection (P. 1) is an operation that subtracts the smallest cost $\beta$ of a unary constraint $C_i$ and adds it to the zero-arity constraint $C_\emptyset$.

The extension (P. 3) is an operation that subtracts, for a value $v_i$ of $D_i$, the extension value ($E[v_i]$) of $v_i$ from a unary constraint $C_i$ and adds it to the binary constraint $C_{ij}$, with $0 < E[v_i] \leq c_i(v_i)$.

All of these transformations are applied to a problem under a set of conditions represented by soft arc consistency levels [16], namely:

- **Node Consistency (NC*)**: a variable $x_i$ is NC* if each value $v_i \in D_i$ satisfies $C_{\emptyset} + c_i(v_i) < UB_i$ and there is a value $v_j \in D_j$ with $c_{ij}(v_i, v_j) = 0$. A problem is NC* if each variable $x_i$ of this problem is NC*.

- **Arc Consistency (AC*)**: a variable $x_i$ is AC* with respect to its neighbor $x_j$ if $x_i$ is NC* and there is, for each value $v_j \in D_j$, a value $v_i \in D_i$ which satisfies $c_{ij}(v_i, v_j) = 0$. $v_i$ is called a simple support of $v_j$. A problem is AC* if each variable $x_i$ of this problem is AC*.

- **Directional Arc Consistency (DAC*)**: a variable $x_i$ is DAC* with respect to its lower neighbor $x_j(i>j)$ if $x_i$ is NC* and there is, for each value $v_i \in D_i$, a value $v_j \in D_j$ which satisfies $c_{ij}(v_i, v_j) + c_j(v_j) = 0$. $v_j$ is called a full support of $v_i$. A problem is DAC* if each variable $x_i$ of this problem is DAC* with its lower neighbors $x_j(i>j)$.

- **Full Directional Arc Consistency (FDAC*)**: A problem is FDAC* if this problem is AC* and DAC*.

To make any problem AC*, it is necessary to apply, for each variable of this problem, a binary projection (P. 2), then a unary projection (P. 1), and finally a deletion of non-NC* values. These three instructions are repeated each time a value is deleted. In a distributed case, each agent $A_j$ performs AC* locally (P. 4) and shares its contribution value stored in $C_{\emptyset}(P.
AC

DAC*()

1 foreach (Ai ∈Gamma+) do
2 foreach (vi ∈Dj) do
3 \( P[v_i] \leftarrow \min_{v_k \in D_j} \{c_{jk}(v_j, v_k) + c_j(v_i)\} \);
4 foreach (v_j ∈D_j) do
5 \( E[v_j] \leftarrow \max_{v_i \in D_i} \{P[v_i] - c_{jk}(v_j, v_i)\} \);
6 Extend(x_j, x_i, E);
7 ExtVals[j,k].put(E);
8 ProjectBinary(x_j, x_i);

P. 6: ProcessPruning(msg)

1 if (msg.type = “Ok”) do
2 ExtVals ←msg.ExtVals;
3 foreach (Ai ∈Gamma+) do
4 Extend(x_i, x_j, ExtVals[kj]);
5 ExtVals[k].clear;
6 DelVals ←msg.DelVals;
7 foreach (A_i ∈Gamma+) do
8 foreach (a ∈DelVals[k].listVals) do
9 \( D_k \leftarrow D_k - a; \)
10 ProjectBinary(x_j, x_i);
11 ProjectUnary();
12 /* ----------------------- */
13 if (D_k is changed) do
14 DelVals[k].nbUnvisitedNbrs.decrement(-1);
15 DelVals[k].listVals.clear;
16 \( C_o \leftarrow \max \{C_o, msg.C_o\} + C_o_j; \) \( C_o_j \leftarrow 0; \)
17 if (C_o ≥UB_j) do
18 broadcastMsg : stp(UB_j);
19 end ←true;
20 CheckPruning();
21 DAC*(()); // here FDAC* is achieved
22 ExtendCPA();

P. 7: CheckPruning()

1 foreach (a ∈D_j) do
2 if (c_j(a) + C_o ≥UB_j) \( \lor \)
3 \( ((\bar{A}_j = A_1) \land (lb(Y ∪(x_j, a)) ≥ UB_j)) \) do
4 \( D_j \leftarrow D_j - a; \)
5 DelVals[j].listVals.add(a);
6 if (D_j is changed) do
7 DelVals[j].nbUnvisitedNbrs.decrement(A_j.nbNbrs);
8 foreach (A_i ∈Gamma+) do
9 if (D_j is empty) do
10 broadcastMsg : stp(UB_j);
11 end ←true;

1, line 2) with the other agents in order to calculate the global \( C_o \) (i.e., \( C_o = \sum_{A_i ∈A} C_o_i \)). Each agent \( A_i \) keeps locally for each of its constraints \( C_{ij} \) an identical copy marked by \( C_{ij}^{ac} \) and used in AC* procedure. During AC*, \( C_{ij}^{ac} \) constraints are changed. To keep the symmetry of these constraints in the agents, each agent \( A_i \) applies, on its copy \( C_{ij}^{ac} \), the same action of its neighbor \( A_j \) and vice versa (P. 4, line 3, 5) [14].

In the same way, we can make any problem DAC*. But in this case, we must first extend (P. 3), for each variable, from its unary costs to its binary costs, the minimum cost required to perform again AC* by its lower neighbors (P. 5).

By executing AC* and DAC* successively for each variable, we can make the problem FDAC*.

C. AFB_BJ+_AC* Algorithm

Each agent \( A_j \) carries out the AFB_BJ+_AC* [3] [2] according to three phases. First, \( A_j \) initializes its data structures and performs AC* to delete suboptimal values from its domain \( D_j \). Second, \( A_j \) chooses, for its variable \( x_j \), a value from its previously filtered domain \( D_j \) in order to extend the CPA by its value assignment \( (x_j, v_j) \). If \( A_j \) has successfully extended the CPA, it sends an \ok? message to the next agent asking it to continue the extension of CPA \( Y_j \). This message loads the extended CPA \( Y_j \), its guaranteed cost (3), its guaranteed cost of AC* (4), the \( C_o \), and the list \( DVal \).

\[
GC(Y_j)[j] = GC(Y_j^{-1}) + \sum_{(x_i,x_j) \in Y_j \land i < j} c_{ij}(v_i, v_j) \tag{3}
\]

\[
GC^*(Y_j) = GC^*(Y_j^{-1}) + c_{j}(v_j) + \sum_{c_{ij} \in c} c_{ij}(v_i, v_j) \tag{4}
\]

\[
LB(Y_j)[i] = GC(Y_j)[i] + \sum_{A_k > A_j} LB_k(Y_j)[i] \tag{5}
\]

In case \( A_j \) fails to extend the CPA, either because it doesn’t find a value that gives a valid CPA, or because all the values in its domain are exhausted, it stops the CPA extension and sends a \back? message, containing the same data structures as an \ok? message excluding \( GC \) and \( GC^* \), to the appropriate agent. If such an agent does not exist or the domain of \( A_j \) becomes empty, \( A_j \) stops its execution and informs the others via \stp? messages. A CPA \( Y_j \) is said to be valid if its lower bound (5) does not exceed the global upper bound \( UB_j \), which represents the cost of the optimal solution achieved so far.

\[
LB(Y_j)[i] = GC(Y_j)[i] + \sum_{A_k > A_j} LB_k(Y_j)[i] \tag{5}
\]

Third, \( A_j \) evaluates the extended CPA by sending \flb? messages, which hold the same data structures as an \ok? message excluding \( C_o \) and \( DVal \), to unassigned agents asking them to evaluate the CPA and send the result of the evaluation. When an agent has completed its evaluation, it sends the result directly to the sender agent via an \flb? message. The evaluation is based on the calculation of appropriate lower bounds for the received CPA \( Y_j \). The lower bound of \( Y_j \) (6) is the minimum lower bound over all values of \( D_j \) with respect to \( Y_j \).
D. AFB BJ⁺ DAC* Algorithm

The AFB BJ⁺ DAC* [4] algorithm follows the same steps as AFB BJ⁺ AC* algorithm except that it performs DAC* instead of AC*. With AC*, we can find for each value of a given agent the corresponding simple support in the domains of its lower and higher neighbors. While with DAC* which is the next level of AC* and the best in reducing the domains of a given DCOP, we can find for each value of a given agent the corresponding full support in the domains of its lower neighbors only (§II-B).

III. THE AFB BJ⁺ FDA C* ALGORITHM

In AFB BJ⁺ FDA C* algorithm, instead of using AC* and DAC* separately as in previous versions, we use FDA C* which provides the same effect of both together.

F DAC* as mentioned in section II-B is executed by executing AC* and DAC* successively. This allows getting for each domain value of each variable simple support in the domains of its higher neighbors (Γ⁺) and full support in the domains of its lower neighbors (Γ⁻). With FDA C*, we can continuously exchange costs between agents, from unary constraints to binary ones and vice versa. This allows the unary costs of each agent and the global zero-arity constraint (C₀) to be continuously updated. So, with these updates, we can significantly reduce the agent domain. In short, FDA C* is a technique that allows agents to choose more precisely the best values for their variables by removing more and more invalid values in their domains.
The skeleton of AFB\_BJ\+\_FDAC\(^+\) algorithm is different from those of AFB\_BJ\+\_AC\(^+\) and AFB\_BJ\+\_AC\(^+\) in two things:

The first one is the DAC\(^+\) procedure (P. 5), which is responsible for finding, for each value of an agent, its full support in the domains of its lower neighbors. In DAC\(^+\) procedure, only a part of the unary costs of a given agent is transferred to its lower neighbors as extension values, not the total of those costs as in AFB\_BJ\+\_DAC\(^+\) algorithm. This is so that the AC\(^+\) condition that this agent must keep with its higher neighbors is not violated (P. 5, line 2-5) [16].

The second is the condition (P. 7, line 2) that allows the first agent to permanently delete the values having a global lower bound exceeding the global upper bound and the values that it has already evaluated. This condition remains correct only for the first agent according to the static order of agents. This is because the first agent does not have a previous agent, which allows it to permanently delete any value that proved to be inconsistent.

### A. Description of AFB\_BJ\+\_FDAC\(^+\)

The AFB\_BJ\+\_FDAC\(^+\) (P. 8) is performed by each agent \(A_j\) as follows:

- \(A_j\) starts with the initialization step (P. 8, line 1-10) in which it performs the AC\(^+\) (P. 4). If \(A_j\) is the 1\(^{st}\) agent (P. 8, line 11), it filters its domain by calling CheckPruning() (P. 7), then performs DAC\(^+\)() (P. 5) after AC\(^+\) to ensure the achievement of the FDAC\(^+\), and finally calls ExtendCPA() to generate a CPA \(Y\).

Next, \(A_j\) starts processing the messages (P. 8, line 17). First, it updates \(UB_j\) and \(v_j^*\) (P. 8, line 21). Then, \(A_j\) updates \(Y\) and \(GC\) and erases all unrelated lower bounds if the received CPA \((msg)^*Y\) is fresh compared to the local one \((Y)\) (P. 8, line 22). Thereafter, \(A_j\) restores all temporarily deleted values (P. 8, line 40).

When receiving an ok? message (P. 8, line 28), \(A_j\) authorizes the sending of fb? messages and calls ProcessPruning() (P. 6).

When calling ProcessPruning() (P. 6), \(A_j\) deals initially, for ok? messages only, with extensions of its higher neighbors (P. 6, line 1-5). Afterward, it updates its \(DVals\), then its neighbors’ domains separately in order to keep the same domains as these agents (P. 6, line 6-9). After that, it performs once more the AC\(^+\) (P. 6, line 10-11). Next, \(A_j\) decrements the unvisited neighbors of \(A_k\), \(DVals[k].UnvNbrs\), and then checks whether it is the last visited neighbor of this agent \(A_k\) in order to reset its list of deleted values \(DVals[k].ListV al\)s (P. 6, line 12-15). Then, \(A_j\) updates its global \(C_o\) (P. 6, line 16). If \(C_o\) exceeds the \(UB_j\), \(A_j\) turns off its execution and notifies the others (P. 6, line 17-19). Finally, \(A_j\) calls CheckPruning() to prune its domain, DAC\(^+\)() (P. 5) to achieve FDAC\(^+\), and ExtendCPA() to extend the received CPA (P. 6, line 20-22).

When calling DAC\(^+\)() (P. 5), \(A_j\) performs the proper extensions from \(C_j\) to each \(C_{ij}\) (P. 5, line 6-7). To do that, \(A_j\) calculates, for each value \(v_j\) of \(D_j\), its extension value (P. 5, line 4-5) based on the prior computation of the values of the later projections on its lower neighbors (P. 5, line 2-3) [16]. Once completed, \(A_j\) performs a binary projection to keep the symmetry of \(C_{ij}\) constraints (P. 5, line 8). It should be noted that the direction taken into account by each agent \(A_j\) for the extension of its costs is towards its lower neighbors (\(\Gamma^+(x_j)\)).

When calling CheckPruning() (P. 7), \(A_j\) deletes any value from its domain for which the sum of the \(C_o\) with the unary cost of this value exceeds \(UB_j\). If \(A_j\) is the first agent, it also deletes any value whose global lower bound exceeds \(UB_j\) and any value has already been evaluated (P. 7, line 2-3). With each new deletion, \(A_j\) initializes the number of its neighbors not yet visited (P. 7, line 5-6). Then, it performs a binary projection to keep the symmetry of \(C_{ij}\) constraints (P. 7, line 8). If \(A_j\) domain becomes empty, \(A_j\) turns off its execution and notifies the others (P. 7, line 9-11).
When calling \( \text{ExtendCPA}(\cdot) \) (P. 9), \( A_j \) looks for a value \( v_j \) for its variable \( x_j \) (P. 9, line 1). If no value exists, \( A_j \) returns to the priority agents by sending a **back** message to the contradictory agent (P. 9, line 2-5). If no agent exists, \( A_j \) turns off its execution and notifies the others via **stop** messages (P. 9, line 6-7). Otherwise, \( A_j \) extends \( Y \) by adding its assignment (P. 9, line 9). If \( A_j \) is the last agent (P. 9, line 10) then a new solution is obtained and the \( UB_j \) is updated, which obliges \( A_j \) to call \( \text{CheckPruning}(\cdot) \) to filter again its domain and then \( \text{ExtendCPA}(\cdot) \) to proceed the search (P. 9, line 11-15). Otherwise, \( A_j \) sends an **ok?** message loaded with the extended \( Y \) to the next agent (P. 9, line 17) and **fb?** messages to unassigned agents (P. 9, line 20).

When \( A_j \) receives an **fb?** message, it filters its domain \( D_j \) with respect to the received \( Y \) (P. 8, line 36-40), calculates the appropriate lower bounds (6), and immediately sends them to the sender via **lb** message (P. 8, line 41).

When \( A_j \) receives an **lb** message, it stores the lower bounds received (P. 8, line 43) and performs \( \text{ExtendCPA}(\cdot) \) to modify its assignment if the lower bound calculated, based on the cost of \( Y (5) \), exceeds the \( UB_j \).

**B. Correctness of AFB\_BJ\^+\_FDAC\^+**

**Theorem 1.** AFB\_BJ\^+\_FDAC\^+ is guaranteed to calculate the optimum and terminates.

**Proof.** The AFB\_BJ\^+\_FDAC\^+ algorithm overrides its previous versions by performing both AC\^+ and DAC\^+, which is essentially just a set of cost extensions performed between an agent and its neighbors after performing AC\^+. These extensions have already been proved which are correct in [16] [8], and they are executed by the AFB\_BJ\^+\_FDAC\^+ without any cost redundancy (P. 3, line 4), (P. 5, line 8), and (P. 6, line 1-5).

**IV. Experimental Results**

In this section, we experimentally compare AFB\_BJ\^+\_FDAC\^+ algorithm with its previous versions, AFB\_BJ\^+\_AC\^+, AFB\_BJ\^+\_DAC\^+, and AFB\_BJ\^+\_DAC\^+, and with BnB-Adopt\^+\_DP2 algorithm [6], which is its famous competitor. Three benchmarks are used in these experiments: soft graph coloring, meetings scheduling, and sensors network. All experiments were performed on DisChoco 2.0 platform [22], in which agents are simulated by Java threads that communicate only through message passing.

**Soft graph coloring** [25]: are defined by \((n, c, p_1)\), which are respectively the number of nodes (i.e., variables), the number of possible colors of each node, and the constraint density. The constraints are applied to adjacent nodes. We evaluated two classes of instances \((n = 6 \ldots 14, c = 8, p_1 = 0.4)\) and \((n = 6 \ldots 14, c = 8, p_1 = 0.7)\). For the constraint costs, they were randomly selected from the set \(\{0, \ldots, 100\}\). For each \(p_1\), we randomly generated an average of 30 instances.

**Meetings scheduling** [23]: are defined by \((m, p, ts)\), which are respectively the number of meetings (i.e., variables), the number of participants, and the number of time slots for each meeting. Each participant has a private schedule of meetings and each meeting takes place at a particular location and at a fixed time slot. The constraints are applied to meetings that share participants. We have evaluated 4 cases A, B, C, and D, which are different in terms of targets/sensors [17].

**Sensors network** [7]: are defined by \((t, s, d)\), which are respectively the number of targets (i.e., variables), the number of sensors, and the number of possible combinations of 3 sensors reserved for tracking each target. A sensor can only track one target at most and each combination of 3 sensors must track a target. The constraints are applied to adjacent targets. We have evaluated 4 cases A, B, C, and D, which are different in terms of targets/sensors [17].

To compare the algorithms, we use two metrics which are the total of messages exchanged (\textit{msgs}) that represents the communication load and the total of non-concurrent constraint checks (\textit{ncccs}) that represents the computation effort.

In tables I and II, we display respectively the results of experiments carried out on coloring problems of sparse (\(p_1 = 0.4\)) and dense (\(p_1 = 0.7\)) graphs. The comparison of these results shows an improvement of AFB\_BJ\^+\_FDAC\^+ algorithm reaching 4,000 messages (resp. 5,000 checks) in the sparse case, and reaching 30,000 messages (resp. 600,000 checks) in the dense case. As for BnB-Adopt\^+\_DP2 algorithm, it remains largely delayed compared to the other algorithms.

Regarding meetings scheduling problems (Fig. 2), the results show a clear improvement of AFB\_BJ\^+\_FDAC\^+ compared to others, whether for \textit{msgs} or for \textit{ncccs}. But with regard to sensors network problems (Fig. 3), BnB-Adopt\^+\_DP2 algorithm retains the pioneering role, despite the superiority of AFB\_BJ\^+\_FDAC\^+ algorithm to its previous versions.
By analyzing the results, we can conclude that the AFB_BJ+_FDAC algorithm is better than its prior versions, because of the existence of Full Directional Arc Consistency (FDAC) that allows agents to reapply AC* multiple times and thus remove more suboptimal values. Regarding the superiority of BnB-Adopt+DP over AFB_BJ+_FDAC in sensors network problems, this is mainly due to the arrangement of the pseudo-tree used by this algorithm that corresponds to the structure of these problems, as well as the existence of DP2 heuristic that facilitates the proper choice of values.

V. CONCLUSION

In this paper, we have introduced the AFB_BJ+_FDAC algorithm. It relies on Full Directional Arc Consistency (FDAC) to further reduce the agent domains of a given DCOP and thus quickly reach its optimal solution. FDAC makes it possible to perform more cost extensions from each agent to its neighbors. This allows reapplying over and over again the AC*, which increases the number of deletions carried out by each agent and thus accelerates the process of solving a problem. Experiments on some benchmarks show that the AFB_BJ+_FDAC algorithm behaves better than its previous versions. As future work, we propose to generalize the use of soft arc consistency in its different levels with DCOP algorithms.

REFERENCES


Rachid Adrdor is currently a member of the Laboratory of Computer Information Systems and Vision of the Faculty of Science, Ibn Zohr University, Agadir, Morocco. He received the Ph.D. degree in Computer Science at the Faculty of Science of the same university. His research interests include Applied Artificial Intelligence, in particular, Distributed Constraint Optimization Problems (DCOPs) and Soft Arc Consistency.

Lahcen Koutti is currently a Professor at Department of Computer Science and a member of the Laboratory of Computer Information Systems and Vision of the Faculty of Science, Ibn Zohr University, Agadir, Morocco. He received the Ph.D. degree in Computational Physics from Paul Verlaine University, France and the Habilitation degree from Ibn Zohr University, Morocco. His research interests include Artificial Intelligence and Computer Vision.