Discussion on the Three Backjumping Schemes Existing in ADOPT-ng

Marius C. Silaghi[†] and Makoto Yokoo[‡]

[†]Florida Institute of Technology
[‡]Kyushu University

Abstract. The original ADOPT-ng has three major versions, corresponding to three different classes of feedback possibilities. The first version is identical to the scheme of the original ADOPT, where messages with feedback are communicated only to the variable of one's parent node in the DFS of the constraint graph. It is similar to the Graph-Based Backjumping concept common in Constraint Satisfaction (CSPs), except that the asynchronous computation paradigm makes the term "backjumping" less intuitively accurate.
The second major version of ADOPT-ng communicates costs to higher priority agents based on dependencies detected dynamically. The third version combined dependencies detected dynamically with statically analyzed constraint graph structure. These versions are related to Conflict-Based Backjumping schemes in CSPs in the way conflicts are announced to earlier variables. Here we discuss and experiment in more detail the advantages and drawbacks of the different backjumping schemes and of some of their variations. While past experiments have shown that sending more feedback is better than sending the minimal information needed for correctness, new experiments show that one should not exaggerate sending too much feedback and that the best strategy is at an intermediary point.

1 Introduction

Distributed Constraint Optimization (DCOP) is a formalism that can model naturally distributed problems. These are problems where agents try to find assignments to a set of variables that are subject to constraints. Typically research has focused on techniques in which reluctance is manifested toward modifications to the distribution of the problem (modification accepted only when some reasoning infers it is unavoidable for guaranteeing that a solution can be reached). This criteria is widely believed to be valuable and adaptable for large, open, and/or dynamic distributed problems [17, 4, 9, 1, 12]. It is also perceived as an alternative approach to privacy requirements [16, 7, 10].

ADOPT-ng [14] is a recent optimization algorithm for DCOPs using a type of nogoods, called *valued nogoods* [3], that besides automatically detecting and exploiting the DFS tree of the constraint graph coherent with the current order, can exploit additional communication leading to significant improvement in efficiency. The examples given of additional communication are based on allowing

each agent to send feedback via valued nogoods to several higher priority agents in parallel. The usage of nogoods is a source of much flexibility in asynchronous algorithms. A nogood specifies a set of assignments that conflict with existing constraints [15]. A basic version of the valued nogoods consists of associating each nogood to a threshold, namely a cost limit violated due to the assignments of the nogood.

We start by defining the general DCOP problem, followed by introduction of the immediately related background knowledge consisting in the ADOPT algorithm and the use of Depth-First Search trees in optimization. In Section 3 we present the ADOPT-ng algorithm that unifies ADOPT with the older Asynchronous Backtracking (ABT). ADOPT-ng is introduced by first describing the goals of its design in terms of the three backjumping schemes that it uses. We provide a more detailed description of used data structures and of their function. Several different new and old variations mentioned during the description are compared experimentally in the last section.

2 Distributed Valued CSPs

Constraint Satisfaction Problems (CSPs) are described by a set X of variables and a set of constraints on the possible combinations of assignments to these variables with values from their domains.

Definition 1 (DCOP). *A distributed constraint optimization problem (DCOP), aka distributed valued CSP, is defined by a set of agents* $A_1, A_2, ..., A_n$*, a set* X *of variables,* $x_1, x_2, ..., x_n$*, and a set of functions* $f_1, f_2, ...f_i, ..., f_n$*,* $f_i : X_i \rightarrow \mathbb{R}_+$*,* $X_i \subseteq X$*, where only* A_i *knows* f_i*. We assume that* x_i *can only take values from a domain* $D_i = \{1, ..., d\}$*.*

Denoting with x *an assignment of values to all the variables in X, the problem is to find* $\underset{x}{\operatorname{argmin}} \sum_{i=1}^{n} f_i(x_{|X_i})$*.*

For simplification and without loss of generality, one typically assumes that $X_i \subseteq \{x_1, ..., x_i\}$*.*

By $x_{|X_i}$ we denote the projection the set of assignments in x on the set of variables in X_i.

3 ADOPT with nogoods

Asynchronous Distributed OPTimization with valued nogoods (ADOPT-ng) is a distributed optimization algorithm. It exploits the increased flexibility brought by the use of valued nogoods. The algorithm can be seen as an extension of both ADOPT and ABT.

A nogood, $\neg N$, specifies a set N of assignments that conflict with existing constraints [15]. Valued nogoods have the form $[SRC, c, N]$ and are an extension of classical nogoods. Each valued nogood has a *set of references to a conflict list of constraints* SRC and a threshold c. The threshold specifies the minimal

weight of the constraints in the conflict list SRC given the assignments of the nogood N [3, 14].

A valued nogood $[SRC, c, N \cup \langle x_i, v \rangle]$ applied to a value v of a variable x_i is referred to as the cost assessment (CA) of that value and is denoted (SRC, v, c, N). If the conflict list is missing (and implies the whole problem) then we speak of a valued global nogood. One can combine valued nogoods by sum-inference and min-resolution to obtain new nogoods [3]. If $N = (\langle x_1, v_1 \rangle, ..., \langle x_t, v_t \rangle)$ where $v_i \in D_i$, then we denote by $\overline{N}$ the set of variables assigned in N, $\overline{N} = \{x_1, ..., x_t\}$.

Proposition 1 (min-resolution). *Assume that we have a set of cost assessments for x_i of the form (SRC_v, v, c_v, N_v) that has the property of containing exactly one CA for each value v in the domain of variable x_i and that for all k and j, the assignments for variables $\overline{N_k} \cap \overline{N_j}$ are identical in both N_k and N_j. Then the CAs in this set can be combined into a new valued nogood. The obtained valued nogood is $[SRC, c, N]$ such that $SRC = \cup_i SRC_i$, $c = \min_i(c_i)$ and $N = \cup_i N_i$.*

Proposition 2 (sum-inference). *A set of cost assessments of type (SRC_i, v, c_i, N_i) for a value v of some variable, where $\forall i, j : i \neq j \Rightarrow SRC_i \cap SRC_j = \emptyset$, and the assignment of any variable x_k is identical in all N_i where x_k is present, can be combined into a new cost assessment. The obtained cost assessment is (SRC, v, c, N) such that $SRC = \cup_i SRC_i$, $c = \sum_i (c_i)$, and $N = \cup_i N_i$.*

When an attempt to combine nogoods using sum-inference fails because their SRCs have a non-empty intersection, one of the inputs is retained and the other one is discarded.

As in ABT, agents communicate with **ok?** messages proposing new assignments of the variable of the sender, **nogood** messages announcing a nogood, and **add-link** messages announcing interest in a variable. As in ADOPT, agents can also use **threshold** messages, but their content can be included in **ok?** messages.

For simplicity we assume in this algorithm that the communication channels are FIFO (as enforced by the Internet transport control protocol). Attachment of counters to proposed assignments and nogoods also ensures this requirement (i.e., older assignments and older nogoods for the currently proposed value are discarded).

3.1 Exploiting DFS trees for Feedback

Here we recall the feedback schemes of ADOPT-ng and introduce the new variants ADOPT-A_ and ADOPT-D__. In ADOPT-ng, agents are totally ordered as in ABT, A_1 having the *highest priority* and A_n the lowest priority. The *target* of a valued nogood is the position of the lowest priority agent among those that proposed an assignment referred by that nogood. Note that the basic version of ADOPT-ng does not maintain a DFS tree, but each agent can send messages with valued nogoods to any predecessor. ADOPT-ng also has hybrid versions that can spare network bandwidth by exploiting an existing DFS tree. It has

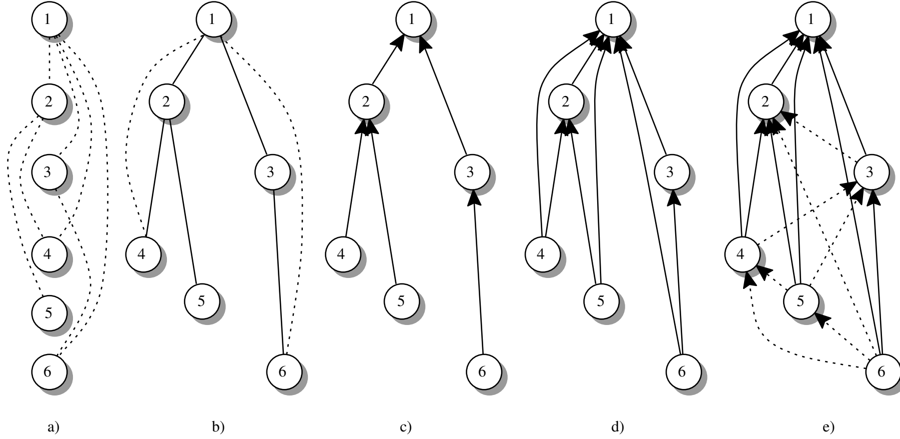

Fig. 1. Feedback modes in ADOPT-ng. a) a constraint graph on a totally ordered set of agents; b) a DFS tree compatible with the given total order; c) ADOPT-p__: sending valued nogoods only to parent (graph-based backjumping); d) ADOPT-d__ and ADOPT-D__: sending valued nogoods to any ancestor in the tree; e) ADOPT-a__ and ADOPT-A__: sending valued nogoods to any predecessor agent.

two ways of exploiting such an existing structure. The first is by having each agent send its valued nogood only to its parent in the tree and it is roughly equivalent to the original ADOPT. The other way is by sending valued nogoods only to ancestors. This later hybrid approach can be seen as a fulfillment of a direction of research suggested in [11], namely communication of costs to higher priority parents.

The versions of ADOPT-ng are differentiated using the notation **ADOPT-XYZ**. **X** shows the destinations of the messages containing valued nogoods. X has one of the values $\{p, a, A, d, D\}$ where p stands for *parent*, a and A stand for *all predecessors*, and d and D stand for *all ancestors in a DFS trees*. The difference between the upper and lower case versions is further explained in Section 3.2. **Y** marks the optimization criteria used by sum-inference in selecting a nogood when the inputs have the same threshold and their SRC intersect. For now we use a single criterion, denoted o, which consists of choosing the nogood whose target has the highest priority. **Z** specifies the type of nogoods employed and has possible values $\{n, s\}$, where n specifies the use of valued global nogoods (without SRCs) and s specifies the use of valued nogoods (with SRCs).

The different schemes are described in Figure 1. The total order on agents is described in Figure 1.a where the constraint graph is also depicted with dotted lines representing the arcs. Each agent (representing its variable) is depicted with a circle. A DFS tree of the constraint graph which is compatible to this total order is depicted in Figure 1.b. ADOPT gets such a tree as input, and each agent sends COST messages (containing information roughly equivalent to a valued

global nogood) only to its parent. As mentioned above, the versions of ADOPT-ng that replicate this behavior of ADOPT when a DFS tree is provided are called ADOPT-p__, where p stands for *parent* and the underscores stand for any legal value defined above for Y and Z respectively. This method of announcing conflicts based on the constraint graph is depicted in Figure 1.c and is related to the classic Graph-based Backjumping algorithm [5, 8].

In Figure 1.d we depict the nogoods exchange schemes used in ADOPT-d__ and ADOPT-D__ where, for each new piece of information, valued nogoods are separately computed to be sent to each of the ancestors in the known DFS tree. As for the initial version of ADOPT, the proof for ADOPT-d__ and ADOPT-D__ shows that the only mandatory nogood messages for guaranteeing optimality in this scheme are the ones to the parent agent. However, agents can infer from their constraints valued nogoods that are based solely on assignments made by shorter prefixes of the ordered list of ancestor agents. The agents try to infer and send valued nogoods separately for all such prefixes.

Figure 1.e depicts the basic versions of ADOPT-ng, when a DFS is not known (ADOPT-a__ and ADOPT-A__), where nogoods can be sent to all predecessor agents. The dotted lines show messages, which are sent between independent branches of the DFS tree, and which are expected to be redundant. Experiments have shown that valued nogoods help to remove the redundant dependencies whose introduction would otherwise be expected from such messages. The provided proof for ADOPT-a__ and ADOPT-A__ shows that the only mandatory nogood messages for guaranteeing optimality in this scheme are the ones to the immediately previous agent. However, agents can infer from their constraints valued nogoods that are based solely on assignments made by shorter prefixes of the ordered list of all agents. As in the other case, the agents try to infer and send valued nogoods separately for all such prefixes.

3.2 Levels of Conflict differentiating ADOPT-a and ADOPT-d from ADOPT-A and ADOPT-D

The valued nogood computed for the prefix $A_1, ..., A_k$ ending at a given predecessor A_k may not be different from the one of the immediately shorter prefix $A_1, ..., A_{k-1}$. Sending that nogood to A_k may not affect the value choice of A_k, since the cost of that nogood applies equally to all values of A_k. Exceptions appear in the case where such nogoods cannot be composed by sum-inference with some valued nogoods of A_k. The new versions ADOPT-D__ and ADOPT-A__ correspond to the case where optional nogood messages are only sent when the target of the payload valued nogood is identical to the destination of the message. The versions ADOPT-d__ and ADOPT-a__ correspond to the case where optional nogood messages are sent to all possible destinations each time that the payload nogood has a non-zero threshold. I.e., in those versions **nogood** messages are sent even when the target of the transported nogood is not identical to the destination agent but has a higher priority.

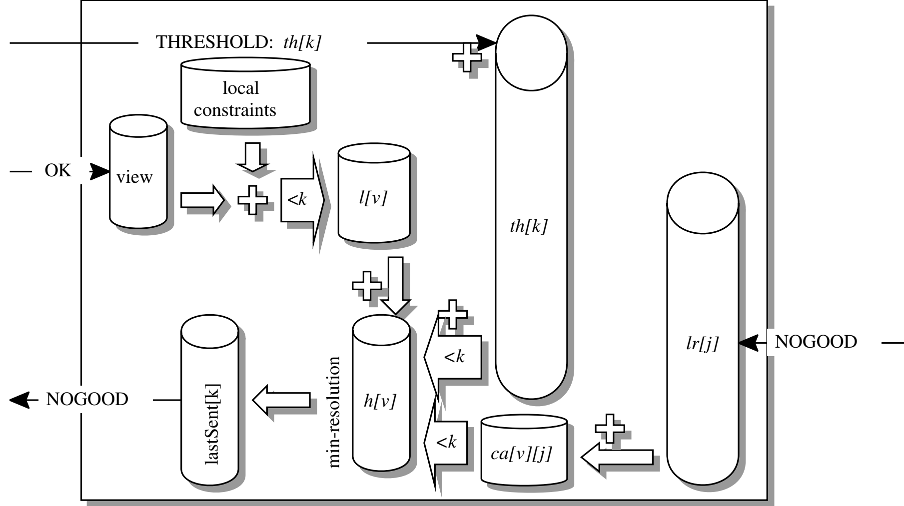

Fig. 2. Schematic flow of data through the different data structures used by an agent A_i in ADOPT-ng.

3.3 Data Structures

Each agent A_i stores its *agent-view* (received assignments), and its outgoing links (agents of lower priority than A_i and having constraints on x_i). The instantiation of each variable is tagged with the value of a separate counter incremented each time the assignment changes. To manage nogoods and CAs, A_i uses matrices l[1..d], h[1..d], ca[1..d][i+1..n], th[1..i], lr[i+1..n] and lastSent[1..i-1] where d is the domain size for x_i. *crt_val* is the current value A_i proposes for x_i. These matrices have the following usage.

- l[k] stores a CA for $x_i = k$, which is inferred solely from the local constraints between x_i and prior variables.
- ca[k][j] stores a CA for $x_i = k$, which is obtained by sum-inference from valued nogoods received from A_j.
- th[k] stores nogoods coming via **threshold/ok?** messages from A_k.
- h[v] stores a CA for x_i=v, which is inferred from ca[v][j], l[v] and th[t] for all t and j.
- lr[k] stores the last valued nogood received from A_k.
- lastSent[k] stores the last valued nogood sent to A_k.

The names of the structures were chosen by following the relation of ADOPT with A* search [13]. Thus, h stands for the “heuristic” estimation of the cost due to constraints maintained by future agents (equivalent to the $h()$ function in A*) and l stands for the part of the standard $g()$ function of A* that is “local” to the current agent. Here, as in ADOPT, the value for $h()$ is estimated by

aggregating the equivalent of costs received from lower priority agents. Since the costs due to constraints of higher priority agents are identical for each value, they are irrelevant for the decisions of the current agent. Thus, the function $f()$ of this version of A* is computed combining solely l and h. We currently store the result of combining h and l in h itself to avoid allocating a new structure for $f()$.

The structures lr and th store received valued nogoods and ca stores intermediary valued nogoods used in computing h. The reason for storing lr, th and ca is that change of context may invalidate some of the nogoods in h while not invalidating each of the intermediary components from which h is computed. Storing these components (which is optional) saves some work and offers better initial heuristic estimations after a change of context. The cost assessments stored in $ca[v][j]$ of A_i also maintain the information needed for **threshold** messages, namely the heuristic estimate for the value v of the variable x_i at successor A_j (to be transmitted to A_j if the value v is proposed again).

The array $lastSent$ is used to store at each index k the last valued nogood sent to the agent A_k. The array lr is used to store at each index k the last valued nogood received from the agent A_k. Storing them separately guarantees that in case of changes in context, they are discarded at the recipient only if they are also discarded at the sender. This property guarantees that an agent can safely avoid retransmitting to A_k messages duplicating the last sent nogood, since if it has not yet been discarded from $lastSent[k]$ then the recipients have not discarded it from $lr[k]$ either.

3.4 Data flow in ADOPT-ng

The flow of data through these data structures of an agent A_i is illustrated in Figure 2. Arrows $\Leftarrow$ are used to show a stream of valued nogoods being copied from a source data structure into a destination data structure. These valued nogoods are typically sorted according to some parameter such as the source agent, the target of the valued nogood, or the value v assigned to the variable x_i in that nogood (see Section 3.3). The + sign at the meeting point of streams of valued nogoods or cost assessments shows that the streams are combined using sum-inference. The $\stackrel{+}{\Leftarrow}$ sign is used to show that the stream of valued nogoods is added to the destination using sum-inference, instead of replacing the destination. When computing a nogood to be sent to A_k, the arrows marked with $\boxed{<k}$ restrict the passage to allow only those valued nogoods containing solely assignments of the variables of agents $A_1, ..., A_k$. Our current implementation recomputes the elements of h and l separately for each target agent A_k by discarding the previous values.

The pseudocode is described in Algorithm 1. The $min_resolution(j)$ function applies the min-resolution over the CAs associated to all the values of the variable of the current agent, but uses only CAs having no assignment from agents with lower priority than A_j. More exactly it first re-computes the array h using only CAs in ca and l that contain only assignments from $A_1, ..., A_j$, and then applies

min-resolution over the obtained elements of h. As mentioned above, in the current implementation we recompute l and h at each call to $min_resolution(j)$, and such a call is separately performed for each ancestor agent A_j.

The order of combining CAs matters. The array h is computed only using cost assessments that are updated solely by sum-inference. To compute h[v]:

1. a) When maintaining DFS trees, for each value v, CAs are combined separately for each set s of agents defining a DFS sub-tree of the current node: tmp[v][s]=sum-inference$_{t \in s}$(ca[v][t]).
   b) Otherwise, with ADOPT-a__ and ADOPT-A__, we act as if we have a single sub-tree:
   tmp[v]=sum-inference$_{t \in [i+1,n]}$(ca[v][t]).
2. CAs from step 1 (a or b) are combined:
   In case (a) this means: $\forall v, s; h[v]$=sum-inference$_{\forall s}$(tmp[v][s]).
   Note that the SRCs in each term of this sum-inference are disjoint and therefore we obtain a valued nogood with threshold given by the sum of the individual thresholds obtained for each DFS sub-tree (or larger).

   For case (b) we obtain h[v]=tmp[v]. This makes sure that at quiescence the threshold of $h[v]$ is at least equal to the total cost obtained at the next agent.
3. Add l[v]: h[v]=sum-inference(h[v], l[v]).
4. Add threshold: h[v]=sum-inference(h[v], th[*]).

3.5 Optimizing valued nogoods

Both for the versions of ADOPT-ng using DFS trees, as well as for the version that does not use such DFS tree preprocessing, if valued nogoods are used for managing cost inferences, then a lot of effort can be saved at context switching by keeping nogoods that remain valid [6]. The amount of effort saved is higher if the nogoods are carefully selected (to minimize their dependence on assignments for low priority variables, which change more often). We compute valued nogoods by minimizing the index of the least priority variable involved in the context. At sum-inference with intersecting SRCs, we keep the valued nogoods with lower priority target agents only if they have better thresholds. Nogoods optimized in similar manner were used in several previous DisCSP techniques [2]. A similar effect is achieved by computing min_resolution(j) with incrementally increasing j and keeping new nogoods only if they have higher thresholds than previous ones with lower targets.

3.6 Example

Now we give a detailed example of a run of ADOPT-ng basic versions ADOPT-aos and ADOPT-Aos. Let us take the problem in Figure 3. Note that in this simple case the two versions do not differ since any optional nogood message can only leave from A_3 to A_1. Such a message is sent in ADOPT-aos only if it has a non-zero threshold, which happens only when A_1 is a target of the

```
when receive ok?(⟨x_j, v_j⟩, tvn) do
    integrate(⟨x_j, v_j⟩);
    if (tvn no-null and has no old assignment) then
        k:=target(tvn); // threshold tvn as common cost;
        th[k]:=sum-inference(tvn,th[k]);
    check-agent-view();
when receive add-link(⟨x_j, v_j⟩) from A_j do
    add A_j to outgoing-links;
    if (⟨x_j, v_j⟩) is old, send new assignment to A_j;
when receive nogood(rvn, t) from A_t do
    foreach new assignment a of a linked variable x_j in rvn do
        integrate(a); // counters show newer assignment;
    if (an assignment in rvn is outdated) then
        if (some new assignment was integrated now) then
            check-agent-view();
        return;
    foreach assignment a of a non-linked variable x_j in rvn do
        send add-link(a) to A_j;
    lr[t]:=rvn;
    foreach value v of x_i such that rvn_|v is not ∅ do
        vn2ca(rvn, i, v) → rca (a CA for the value v of x_i);
        ca[v][t]:=sum-inference(rca,ca[v][t]);
        update h[v] and retract changes to ca[v][t] if h[v]'s cost decreases;
    check-agent-view();
procedure check-agent-view() do
    for every A_j with higher priority than A_i (respectively ancestor in the DFS tree,
    when one is maintained) do
        for every(v ∈ D_i) update l[v] and recompute h[v];
                // with valued nogoods using only instantiations of {x_1, ..., x_j};
        if (h has non-null cost CA for all values of D_i) then
            vn:=min_resolution(j);
            if (vn ≠ lastSent[j])  then
                if (target(vn) == j) then
                    send nogood(vn,i) to A_j;
                    lastSent[j] = vn;
    crt_val=argmin_v(cost(h[v]));
    if (crt_val changed) then
        send ok?(⟨x_i, crt_val⟩, ca2vn(ca[crt_val][k]),i)
            to each A_k in outgoing_links;
procedure integrate(⟨x_j, v_j⟩) do
    discard elements in ca, th, lastSent and lr based on other values for x_j;
    use lr[t]_|v to replace each discarded ca[v][t];
    store ⟨x_j, v_j⟩ in agent-view;
```

Algorithm 1: Receiving messages of A_i in ADOPT-ng

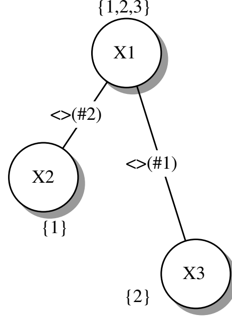

Fig. 3. A DisCOP with three agents and two inequality constraints. The fact that the cost associated with not satisfying the constraint $x_1 \neq x_2$ is 2, is denoted by the notation (#2). The cost for not satisfying the constraint $x_1 \neq x_3$ is 1.

1. A_1 ——— **ok?**$\langle x_1, 1\rangle$ ——→ A_2, A_3
2. A_2 — **nogood**$[|F, T, F|, 2, \langle x_1, 1\rangle]$ → A_1
3. A_1 ——— **ok?**$\langle x_1, 2\rangle$ ——→ A_2, A_3
4. A_3 — **nogood**$[|F, F, T|, 1, \langle x_1, 2\rangle]$ → A_1, A_2
5. A_1 ——— **ok?**$\langle x_1, 3\rangle$ ——→ A_2, A_3
6. A_2 — **nogood**$[|F, F, T|, 1, \langle x_1, 2\rangle]$ → A_1

Fig. 4. Trace of ADOPT-aos and ADOPT-Aos on the problem in Figure 3

message, which means that it will also be sent in ADOPT-Aos. A trace is shown in Figure 4 where identical messages sent simultaneously to several agents are grouped by displaying the list of recipients. The agents start selecting values for their variables and announce them to interested lower priority agents. A_3 has no constraint between x_3 and x_2; therefore the first exchanged messages are **ok?** messages sent by A_1 to both successors A_2 and A_3 and proposing the assignment x_1=1.

After receiving the assignment from A_1, the best (and only) assignment for A_2 is x_2=1 at a cost of 2 due to the conflict with the constraint $x_1 \neq x_2$. Similarly A_3 instantiates x_3 with 2 and with a local cost of 0.

Since the best local cost of A_2 is not null, A_2 performs a min-resolution. Since a single value exists for A_2 and ca is empty, this min-resolution simply obtains a valued nogood defined by the existing local nogood: $h[1] = l[1] = [C_{1,2}, 2, \langle x_1, 1\rangle]$. In our implementation we decide to maintain a single reference for each agent's secret constraints. SRCs are represented as Boolean values in an array of size n. A value on the ith position in the array SRC equal to T signifies that the constraints of A_i are used in the inference of that nogood. A_2 also stores the sent valued nogood in $lastSent[1]$ such that it avoids resending it without modification as a result of receiving other messages. A_1 stores this received valued nogood in

$lr[2]$, from where it is used to update $ca[1][2]$, by sum-inference. Since $ca[1][2]$ is empty, it becomes equal to this valued nogood.

Agent A_1 now updates its $h[1]$ by setting it to $ca[1][2]$ (since $l[1]$ and $ca[1][3]$ are empty). Since the threshold of $h[1]$ becomes 2 and is higher than the threshold of the other two values, {2,3}, in the domain of x_1, A_1 changes the assignment of x_1 to one of them, here 2. This is announced through another **ok?** message to A_2 and A_3.

On the receipt of the **ok?** messages, the agents update their agent-view with the new assignment. Each agent tries to generate valued nogoods for each prefix of its list of predecessor agents: $\{A_1\}$ and $\{A_1, A_2\}$ respectively. This time it is A_2 whose only possible assignment leads to a non-zero local cost. Based on its agent-view and constraints, A_2 generates a corresponding valued nogood $[C_{1,3}, 1, \langle x_1, 2\rangle]$ with threshold 1 due to the weight 1 of its constraint. This valued nogood is sent to the agent A_1 whose assignment is involved in this nogood. To guarantee optimality the nogood is also sent to its immediate predecessor, namely the agent A_2, making sure that at quiescence all the costs of its children are summed.

After receiving this second nogood, A_1 stores it in $lr[3]$, used further by sum-inference to set $ca[2][3]$, and finally used to update $h[2]$. As a result, A_1 now switches its assignment to its value that has the lowest threshold in h, namely the value 3. The new assignment is again sent by **ok?** messages to its successors. Meanwhile, the agent A_2 also processes the valued nogood received from A_3 storing it in its own $lr[3]$, $ca[2][3]$ and $h[2]$. The nogood is not changed by sum inference or min-resolution at this agent; it is sent on to A_1 which stores it in $lr[2]$ and $ca[2][2]$. However, it does not lead to any modification in the $h[2]$ of A_1 since the SRCs of $ca[2][2]$ and $ca[2][3]$ have a nonempty intersection.

After receiving the third assignment from A_1, the other two agents reach quiescence with cost 0; thus an optimal solution is found. Note that the existence of message 6 depends on whether the message 5 (with the last assignment from A_1) reaches A_2 before or after the nogood from A_3, that the message 5 invalidates. The solution is found in 5 half-round-trips of messages (a logic time of 5).

4 Experiments

The algorithms are compared on the same problems that are used to report ADOPT's performance in [11]. To correctly compare our techniques with the original ADOPT, we have used the same order (or DFS trees) on agents for each problem. The impact of the existence of a good DFS tree compatible with the used order is tested separately by comparison with a random ordering. The set of problems distributed with ADOPT and used here contains 25 problems for each problem size. It contains problems with 8, 10, 12, 14, 16, 18, 20, 25, 30, and 40 agents, and for each of these numbers of agents it contains test sets with density .2 and with density .3. The density of a (binary) constraint problem's graph with n variables is defined by the ratio between the number of binary

Agents	ADOPT	aos	Aos	dos	Dos
8	922.2	429.48	427.92	429.2	427.76
10	779.84	354.12	365.76	351.16	357.48
12	1244.56	544.76	562.96	544.24	552.88
14	1591	674.56	704.96	656.24	669.44
16	2453.8	839.92	852.6	814.76	845.48
18	4666.4	1777.44	1815.6	1727.84	1765.16
20	*6264.71	1711.84	1701.6	1718.36	1703.88
25	*33919.5	7499.32	7498.12	7434.96	7276.4
30	*58459.1	16707.48	17618.48	16097.36	17154.4
40	*	96406.76	90747.6	93678.76	90951.56

Fig. 5. Longest causal chain of messages (cycles) used to solve versions of ADOPT using CAs, averaged over problems with density .3. Table entries containing * specify that the corresponding algorithm did not manage to solve all instances of that size in 2 weeks, and the eventually present value is based on the subset of problems solved in that time.

Agents	ADOPT	aos	Aos	dos	Dos
8	45.2	31.4	31.4	31.32	31.32
10	60.2	30.92	29.56	30.24	30.44
12	69.12	39.32	39.6	39.48	39.52
14	75.64	42.32	42.8	42.44	42.72
16	97.84	44.24	46.2	44.04	45.16
18	162.16	75.08	75.36	73.08	74.8
20	71.8	36.48	35.16	36.48	34.84
25	221.44	83.12	83.96	80.64	84.2
30	433.92	112.68	122.64	112.52	114.84
40	720.04	117.28	108.4	107.64	112.24

Fig. 6. Longest causal chain of messages (cycles) used to solve versions of ADOPT using CAs, averaged on 25 problems with density .2.

constraints and $\frac{n(n-1)}{2}$. Results are averaged on the 25 problems with the same parameters.

The length of the longest causal (sequential) chain of messages of each solver, computed as the number of cycles of our simulator and averaged on problems with density .3, is given in Figure 5. Results for problems with density .2 are given in Figure 6. It took more than two weeks for the original ADOPT implementation to solve one of the problems for 20 agents and density .3, and one of the problems for 25 agents and density .3 (at which moment the solver was interrupted). Therefore, it was evaluated using only the remaining 24 problems at those problem sizes.

Nodes	aos	Aos	dos	Dos	pon
14	21981.96	14696.88	15760.4	12427.52	16869.40
16	35710.8	22057.12	24552.24	19553.64	28375.24
18	93368.6	50861.08	64610.96	44328.36	58243.40
20	116468.8	56852.32	85127.44	49630.32	81116.80
25	863145.12	350337.6	602437.08	291927.8	630519.00
30	3640811.3	1137317	1853420	881049.7	830616.88
40	49802812	9046121	22413986.4	7141719	

Fig. 7. Total number of messages used by versions of ADOPT-ng, averaged on problems with density .3.

Nodes	Aos	aos	dos	Dos
16	18	33	19	15
18	56	111	70	45
20	74	161	115	61
25	674	1615	1198	539
30	2889	8474	4907	2101

Fig. 8. Total number of seconds used on a simulator by versions of ADOPT-ng, on the 25 problems with density .3.

Agents	16	18	20	25	30	40
ADOPT-aos	839.92	1777.44	1711.84	7499.32	16707.48	96406.76
no threshold	849.76	1783.6	1763.6	7641.84	16917.72	96406.64
ADOPT-dos	814.76	1727.84	1718.36	7434.96	16097.36	93678.76
no threshold	847.76	1779.6	1741.28	7500.04	16958.28	98932.72

Fig. 9. Impact of threshold valued nogoods on the longest causal chain of messages (cycles) for versions of ADOPT-ng, averaged on problems with density .3.

Agents	16	18	20	25	30	40
DFS compatible	839.92	1777.44	1711.84	7499.32	$16*10^3$	$96*10^3$
random order	$461*10^3$	$1.5*10^6$	$3.7*10^6$	$48*10^6$	$128*10^6$	—

Fig. 10. Impact of choice of order according to a DFS tree on the longest causal chain of messages (cycles) for versions of ADOPT-ng, averaged on problems with density .3.

The use of valued nogoods in ADOPT-ng brought an improvement of approximately 7 times on problems of density 0.2, and an approximately 5 times improvement on the problems of density .3.

Figure 5 shows that, with respect to the number of cycles, the use of SRCs practically replaces the need to maintain the DFS tree since ADOPT-aos and ADOPT-Aos are comparable in efficiency with ADOPT-dos and ADOPT-Dos.

SRCs bring improvements over versions with valued global nogoods, since SRCs allow detection of dynamically obtained independence.

Versions using DFS trees require fewer parallel/total messages, being more network friendly, as seen in Figure 7. Figure 7 shows that refraining from sending too many optional nogoods messages, as done in ADOPT-Aos and ADOPT-Dos, is comparable to ADOPT-pon in terms of total number of messages, while maintaining the efficiency in cycles comparable to ADOPT-aos and ADOPT-dos.

A comparison between the total times required by versions of ADOPT-ng on a simulator is shown in Figure 8. It reveals the computational load of the agents, which, as expected, is proportional to the total number of exchanged messages.

A separate set of experiments was run for isolating and evaluating the contribution of threshold valued nogoods. Figure 9 shows that the contribution of threshold nogoods is higher when a DFS tree is maintained, but still it is no more than 5%.

Another experiment, whose results are shown in Figure 10, is meant to evaluate the impact of the guarantees that the ordering on agents is compatible with some short DFS tree. We evaluate this by comparing ADOPT-aos with an ordering that is compatible with the DFS tree built by ADOPT, versus a random ordering. The results show that random orderings are unlikely to be compatible with short DFS trees and that verifying the existence of a short DFS tree compatible to the ordering on agents to be used by ADOPT-ng is highly recommended.

Figure 5 clearly show that the highest improvement in number of cycles is brought by sending valued nogoods to other ancestors besides the parent. The use of the structures of the DFS tree makes slight improvements in number of cycles (when nogoods reach all ancestors) and slight improvements in total message exchange. To obtain a low total message traffic and to reduce computation at agent level, we found that it is best not to announce any possible valued nogoods to each interested ancestor. Instead, one can reduce the communication without a penalty in number of cycles by only announcing valued nogoods to the highest priority agent to which they are relevant (besides the communication with the parent, which is required for guaranteeing optimality).

5 Conclusions

ADOPT-ng detects and exploits dynamically created independence between subproblems. Such independence can be caused by assignments. Previous experimentation with ADOPT-ng has shown that it is important for an agent to infer and send in parallel several valued nogoods to different higher priority agents. New experiments show that exaggerating this principle by sending each valued nogood to all ancestors able to handle it produces little additional gain while increasing the network traffic and the computational load. Instead, each inferred valued nogood should be sent only to the highest priority agent that can handle it (its target).

We isolated and evaluated the contribution of using threshold valued nogoods in ADOPT-ng, which was found to be at most 5%. In addition, we determined the importance of precomputing and maintaining a short DFS tree of the constraint graph, or at least of guaranteeing that a DFS tree is compatible with the order on agents, which is almost an order of magnitude in our problems. Choosing a strategy of medium aggressiveness for sending valued nogoods to predecessors brings slight improvements in terms of length of longest causal chain of messages (measured as number of cycles of the simulator). It brings an order of magnitude improvements in the total number of messages.

References

1. S. Ali, S. Koenig, and M. Tambe. Preprocessing techniques for accelerating the DCOP algorithm ADOPT. In *AAMAS*, 2005.
2. C. Bessiere, I. Brito, A. Maestre, and P. Meseguer. Asynchronous backtracking without adding links: A new member in the abt family. *Artificial Intelligence*, 161:7–24, 2005.
3. P. Dago. Backtrack dynamique valué. In *JFPLC*, pages 133–148, 1997.
4. J. Davin and P. J. Modi. Impact of problem centralization in distributed cops. In *DCR*, 2005.
5. R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *AI'90*, 1990.
6. M. L. Ginsberg. Dynamic backtracking. *Journal of AI Research*, 1, 1993.
7. R. Greenstadt, J. Pearce, E. Bowring, and M. Tambe. Experimental analysis of privacy loss in dcop algorithms. In *AAMAS*, pages 1024–1027, 2006.
8. Y. Hamadi and C. Bessière. Backtracking in distributed constraint networks. In *ECAI'98*, pages 219–223, 1998.
9. R. Maheswaran, M. Tambe, E. Bowring, J. Pearce, and P. Varakantham. Taking DCOP to the real world: Efficient complete solutions for distributed event scheduling. In *AAMAS*, 2004.
10. R. Mailler and V. Lesser. Solving distributed constraint optimization problems using cooperative mediation. In *AAMAS*, pages 438–445, 2004.
11. P. J. Modi, W.-M. Shen, M. Tambe, and M. Yokoo. Adopt: Asynchronous distributed constraint optimization with quality guarantees. *AIJ*, 161, 2005.
12. A. Petcu and B. Faltings. Odpop: An algorithm for open/distributed constraint optimization. In *AAAI*, 2006.
13. M.-C. Silaghi, J. Landwehr, and J. B. Larrosa. volume 112 of *Frontiers in Artificial Intelligence and Applications*, chapter Asynchronous Branch & Bound and A* for DisWCSPs with heuristic function based on Consistency-Maintenance. IOS Press, 2004.
14. M.-C. Silaghi and M. Yokoo. Nogood-based asynchronous distributed optimization (ADOPT-ng). In *AAMAS*, 2006.
15. R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–193, 1977.
16. R. Wallace and M.-C. Silaghi. Using privacy loss to guide decisions in distributed CSP search. In *FLAIRS'04*, 2004.
17. W. Zhang and L. Wittenburg. Distributed breakout revisited. In *Proc. of AAAI*, Edmonton, July 2002.