

Revisiting ADOPT-ing and its Feedback Schemes

Marius C. Silaghi
Florida Institute of Technology

Makoto Yokoo
Kyushu University

Abstract

Here we revisit ADOPT-ing and bring two new contributions. One contribution consists of developing variations on the algorithms keeping the improvement in length of chain of causal messages without an increase in the total number of messages. 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. This brings large efficiency improvements.

We also find that one of the nogood storages of ADOPT-ing can be removed without effects on efficiency while decreasing the space complexity by a factor given by the number of agents. We also provide a more general proof showing which types of nogood storages can be used in the inference of feedback without compromising correctness. In particular we show that all such structures can be updated by sum-inference, and from **threshold** messages.

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 [7, 1, 2, 10]. It is also perceived as an alternative approach to privacy requirements [12, 5, 8].

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

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

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

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

2 ADOPT with nogoods

Asynchronous Distributed OPTimization with inferences based on valued nogoods (ADOPT-ing)¹ 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 [9] and asynchronous backtracking (ABT) [13].

A nogood, $\neg N$, specifies a set N of assignments that conflict with existing constraints. 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, 11].

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 $N = (\langle x_1, v_1 \rangle, \dots, \langle x_t, v_t \rangle)$ where $v_i \in D_i$, then we denote by \overline{N} the set of variables assigned in N , $\overline{N} = \{x_1, \dots, x_t\}$.

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 $N = \cup_i N_i$.

¹Introduced with the acronym ADOPT-ng in [11].

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 $N = \cup_i N_i$.

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. 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).

2.1 Exploiting DFS trees for Feedback

Here we recall the feedback schemes of ADOPT-ing and introduce the new variants ADOPT-A_{...} and ADOPT-D_{...}. In ADOPT-ing, 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-ing does not maintain a DFS tree, but each agent can send messages with valued nogoods to any predecessor. ADOPT-ing also has hybrid versions that can spare network bandwidth by exploiting an existing DFS tree. It has 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 [9], namely communication of costs to higher priority parents.

The versions of ADOPT-ing 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*. **Y** marks the optimization criteria used by sum-inference in selecting a nogood when the inputs have the same threshold. 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 to-

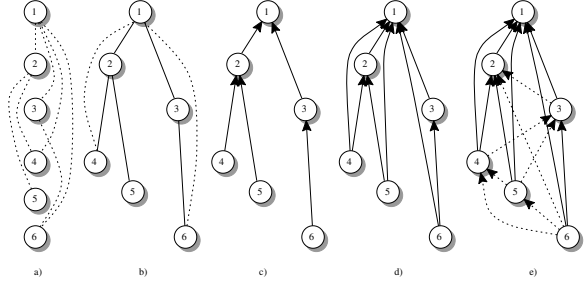


Figure 1. Feedback modes in ADOPT-ing.

tal 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-ing 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 [4, 6].

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-ing, 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 as-

signments 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.

A, D vs. a, d: The valued nogood computed for the prefix A_1, \dots, A_k ending at a given predecessor A_k may not be different from the one of the immediately shorter prefix A_1, \dots, 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 (except when 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.

2.2 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 . cr_val is the current value A_i proposes for x_i .

- $l[k]$ stores a CA for $x_i = k$, 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 follow the relation of ADOPT with A* search. 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 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.

2.3 Data flow in ADOPT-ing

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, \dots, 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. 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:

```

when receive ok?( $\langle x_j, v_j \rangle, tvn$ ) do
  // tvn is a threshold valued nogood
  integrate( $\langle x_j, v_j \rangle$ );
  if ( $tvn$  no-null and has no old assignment) then
     $k := \text{target}(tvn)$ ; // threshold tvn as common cost;
     $th[k] := \text{sum-inference}(tvn, th[k])$ ;
  check-agent-view();

when receive add-link( $\langle x_j, v_j \rangle$ ) from  $A_j$  do
  add  $A_j$  to outgoing-links;
  if ( $\langle x_j, v_j \rangle$ ) 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  $\emptyset$  do
     $vn2ca(rv_n, i, v) \rightarrow rca$  (a CA for the value  $v$  of  $x_i$ );
     $ca[v][t] := \text{sum-inference}(rca, ca[v][t])$ ;
    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 ( $v \in D_i$ ) update  $l[v]$  and recompute  $h[v]$ ;
      // with valued nogoods using only instantiations
      // of  $\{x_1, \dots, x_j\}$ ;
    if ( $h$  has non-null cost CA for all values of  $D_i$ ) then
       $vn := \text{min-resolution}(j)$ ;
      if ( $vn \neq \text{lastSent}[j]$ ) then
        if ( $\text{target}(vn) == j$ ) then
          send nogood( $vn, i$ ) to  $A_j$ ;
           $\text{lastSent}[j] = vn$ ;
       $crt\_val = \text{argmin}_v(\text{cost}(h[v]))$ ;
      if ( $crt\_val$  changed) then
        send ok?( $\langle x_i, crt\_val \rangle, ca2vn(ca[crt\_val][k], i)$ )
          to each  $A_k$  in outgoing_links;

procedure integrate( $\langle x_j, v_j \rangle$ ) do
  discard elements in  $ca, th, \text{lastSent}$  and  $lr$ 
    based on other values for  $x_j$ ;
  use  $lr[t]|_v$  to replace each discarded  $ca[v][t]$ ;
  store  $\langle x_j, v_j \rangle$  in agent-view;

procedure init do
   $h[v] := l[v]$ ; // initialize CAs from unary constraints;
   $crt\_val = \text{argmin}_v(\text{cost}(h[v]))$ ;
  send ok?( $\langle x_i, crt\_val \rangle, \emptyset$ ) to all agents in outgoing-links;

```

Algorithm 1: Receiving messages of A_i in ADOPT-ing

$\forall v, s; h[v] = \text{sum-inference}_{\forall s}(\text{tmp}[v][s])$.

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 $h[v] = \text{tmp}[v]$. 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 $l[v]$: $h[v] = \text{sum-inference}(h[v], l[v])$.
4. Add threshold: $h[v] = \text{sum-inference}(h[v], th[*])$.

The following is a new result:

Lemma 1 (Infinite Cycle) *At a given agent, assume that the agent-view no longer changes and that its array h (used for min-resolution and for deciding the next assignment) is computed only using cost assessments that are updated solely by sum-inference. In this case the thresholds of the elements of its h cannot be modified in an infinite cycle.*

Proof. Valued nogoods that are updated solely by sum-inference have thresholds that can only increase (which can happen only a finite number of times). For a given threshold, modifications can only consist of modifying assignments to obtain lower target agents, which again can happen only a finite number of times. Therefore, after a finite number of events, the cost assessments used to infer h will not be modified any longer and therefore h will no longer be modified. \square

Corollary 1.1 *If ADOPT-ing uses the aforementioned procedure, then for a given agent-view, the elements of the array h for that agent cannot be modified in an infinite cycle.*

Remark 1 *Since lr contains the last received valued nogoods via messages other than **ok?** messages which change the agent-view, that array is updated by assignment with recently received nogoods without sum-inference. Therefore, we do not use it directly to infer h . We use lr only to initialize ca after a change of the agent-view.*

Note that with the described procedure, a newly arriving valued nogood can decrease the threshold of certain elements of h . This is because, while increasing the threshold of some element in ca , it can also modify its SRC and therefore forbid its composition by sum-inference with other cost assessments.

Remark 2 (Obtaining Monotonic Increase) *The implementation used for the experiments reported here avoids the undesired aforementioned effect, where incoming nogoods decrease thresholds of cost assessments in h . Namely, after*

a newly received valued nogood is added by sum-inference to the corresponding element of $ca[v]$ for some value v , if the threshold of $h[v]$ decreases then the old content of $ca[v]$ is restored. Each new valued nogood is used for updating lr . However, on each change of the agent-view (set of known valid assignments), all values of ca are updated using the valued nogoods found in lr and th .

Lemma 2 Revised ADOPT-ing terminates in finite time.

Proof.

Given the list of agents A_1, \dots, A_n , define the suffix of length m of this list as the last m agents. Then the result follows immediately by induction for an increasingly growing suffix (increasing m), assuming the other agents reach quiescence.

The basic case of the induction (for the last agent) follows from the fact that the last agent terminates in one step if the previous agents do not change their assignments.

Let us now assume that the induction assertion is true for a suffix of k agents. Based on this assumption we now prove the induction step, namely that the property is also true for a suffix of $k+1$ agents: For each assignment of the agent A_{n-k} , the remaining k agents will reach quiescence, according to the assumption of the induction step; otherwise, the assignment's CA threshold increases. By construction, thresholds for CAs associated with the values of A_{n-k} can only grow (see Remark 2). Even without the technique in Remark 2, thresholds for CAs associated with the values of A_{n-k} will eventually stop being modified as a consequence of Lemma 1. After values are proposed in turn and the smallest threshold reaches its highest estimate, agent A_{n-k} selects the best value and reaches quiescence. The other agents reach quiescence according to the induction step. \square

The following results were proven in [11] and remain true for the described generalization.

Lemma 3 The last valued nogoods sent by each agent additively integrate the non-zero costs of the constraints of all of the agent's successors.

Theorem 4 ADOPT-ing returns an optimal solution.

Note that the lvn data structure mentioned in [11] is no longer used. That data structure holds n^2 valued nogoods and its removal has little impact on the experimental results. Since remaining arrays have dimensions that are only linear in n , the space complexity of the revisited version is smaller than the one of the original version by the factor n .

2.4 Example

Now we give a detailed example of a run of ADOPT-ing basic versions ADOPT-aos and ADOPT-Aos. Let us take

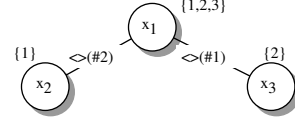


Figure 2. The penalty of conflicting the constraint $x_1 \neq x_2$ is 2, denoted ($\#2$).

1. A_1	$\xrightarrow{\text{ok?}\langle x_1, 1 \rangle}$	A_2, A_3
2. A_2	$\xrightarrow{\text{nogood}[F, T, F], 2, \langle x_1, 1 \rangle}$	A_1
3. A_1	$\xrightarrow{\text{ok?}\langle x_1, 2 \rangle}$	A_2, A_3
4. A_3	$\xrightarrow{\text{nogood}[F, F, T], 1, \langle x_1, 2 \rangle}$	A_1, A_2
5. A_1	$\xrightarrow{\text{ok?}\langle x_1, 3 \rangle}$	A_2, A_3
6. A_2	$\xrightarrow{\text{nogood}[F, F, T], 1, \langle x_1, 2 \rangle}$	A_1

Figure 3. Trace of ADOPT-aos and ADOPT-Aos on the problem in Figure 2

the problem in Figure 2. 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 message, which means that it will also be sent in ADOPT-Aos. A trace is shown in Figure 3 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 i^{th} 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

Agents	ADOPT	aos	Aos	dos	Dos
14	75.64	42.32	42.8	42.44	42.72
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

Table 1. Cycles at density 20%.

$h[1]$ 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_3 whose only possible assignment leads to a non-zero local cost. Based on its agent-view and constraints, A_3 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).

3 Experiments

The algorithms are compared on the same problems that are used to report ADOPT’s performance in [9]. To correctly compare our techniques with the original ADOPT and ADOPT-ing, 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

Nodes	aos	Aos	dos	Dos	pon
14	21.981	14.696	15.760	12.427	16.869
25	863.14	350.33	602.43	291.92	630.51
30	3640.81	1137.31	1853.42	881.04	830.61
40	49803	9046.12	22414	7141.7	

Table 2. Thousands messages at density .3.

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 up to 40 agents, and it contains test sets with density .2 and with density .3.

The length of the longest causal (sequential) chain of messages of each solver, computed as the number of cycles of our simulator is given in Tables 3 and 1. 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).

The use of valued nogoods in ADOPT-ing 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.

Table 3 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 total messages, being more network friendly, as seen in Table 2. Table 2 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.

Another experiment, whose results are shown in Table 4, 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-ing is highly recommended.

Table 3 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

Agents	ADOPT	aos	Aos	dos	Dos
14	1591	674.56	704.96	656.24	669.44
18	4666.4	1777.44	1815.6	1727.84	1765.16
30	*58459.1	16707.48	17618.48	16097.36	17154.4
40	*	96406.76	90747.6	93678.76	90951.56

Table 3. Cycles at density .3. Entries with * show algorithm had to be interrupted.

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	$461*10^3$	$1.5*10^6$	$3.7*10^6$	$48*10^6$	$128*10^6$	—

Table 4. Impact of choice of order at density .3.

message traffic and to reduce computation at agent level, we found that it is best not to announce any possible valued no-goods 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, required for guaranteeing optimality).

4 Conclusions

The Asynchronous Distributed Optimization with no-goods (ADOPT-ing) algorithm is revisited and its space complexity is reduced by one factor without a negative impact on efficiency. Its proof is generalized to allow for extensions of its nogood storage schemes. Previous experimentation with ADOPT-ing has shown that it is important for an agent to infer and send in parallel several valued no-goods to different higher priority agents [11]. That conclusion was drawn by experimenting with the two extreme settings: 1) where no optional message with valued nogoods is sent (similar to ADOPT), and 2) where any inferred valued nogood is sent to all ancestors able to handle it. The first extreme was found to be an order of magnitude better than the other extreme in terms of total number of messages. The second extreme is an order of magnitude better than the first one in length of the longest causal chain of messages.

Experiments show that an intermediary setting has the advantages of both extremes. If each inferred valued no-good is sent only to the highest priority agent that can handle it (its target), then the protocol is comparable to ADOPT in the total number of messages and comparable to the other extreme in length of longest causal chain of messages.

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 the 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, as in ADOPT.

References

- [1] S. Ali, S. Koenig, and M. Tambe. Preprocessing techniques for accelerating the DCOP algorithm ADOPT. In *AAMAS*, 2005.
- [2] A. Chechotka and K. Sycara. No-commitment branch and bound search for distributed constraint optimization. In *AA-MAS*, 2006.
- [3] P. Dago and G. Verfaillie. Nogood recording for valued constraint satisfaction problems. In *ICTAI*, 1996.
- [4] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *AI'90*, 1990.
- [5] R. Greenstadt, J. Pearce, E. Bowring, and M. Tambe. Experimental analysis of privacy loss in dcop algorithms. In *AAMAS*, pages 1024–1027, 2006.
- [6] Y. Hamadi and C. Bessière. Backtracking in distributed constraint networks. In *ECAI'98*, pages 219–223, 1998.
- [7] 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 *AA-MAS*, 2004.
- [8] R. Mailler and V. Lesser. Solving distributed constraint optimization problems using cooperative mediation. In *AAMAS*, pages 438–445, 2004.
- [9] P. J. Modi, W.-M. Shen, M. Tambe, and M. Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *AIJ*, 161, 2005.
- [10] A. Petcu and B. Faltings. ODPOP: an algorithm for open/distributed constraint optimization. In *AAAI*, 2006.
- [11] M.-C. Silaghi and M. Yokoo. Nogood-based asynchronous distributed optimization. In *AAMAS*, 2006.
- [12] R. Wallace and E. Freuder. Constraint-based multi-agent meeting scheduling: Effects of agent heterogeneity on performance and privacy loss. In *DCR*, pages 176–182, 2002.
- [13] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE TKDE*, 10(5):673–685, 1998.