Dynamic DFS Tree in ADOPT-ing

Marius C. Silaghi
Florida Institute of Technology
msilaghi@fit.edu

Makoto Yokoo
Kyushu University, Japan
yokoo@is.kyushu-u.ac.jp

Abstract

Several distributed constraint reasoning algorithms employ Depth First Search (DFS) trees on the constraint graph that spans involved agents. In this article we show that it is possible to dynamically detect a minimal DFS tree, compatible with the current order on agents, during the distributed constraint reasoning process of the ADOPT algorithm. This also allows for shorter DFS trees during the initial steps of the algorithm, while some constraints did not yet prove useful given visited combinations of assignments. Earlier distributed algorithms for finding spanning trees on agents did not look to maintain compatibility with an order already used. We also show that announcing a nogood to a single optional agent is bringing significant improvements in the total number of messages. The dynamic detection of the DFS tree brings improvements in simulated time.

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. Several applications are addressed in the literature, such as multi-agent scheduling problems, oil distribution problems, or distributed control of red lights in a city (Modi & Veloso 2005; Marcellino, Omar, & Moura 2007; Walsh 2007). 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. It is also perceived as an alternative approach to privacy requirements (Greenstadt *et al.* 2006).

Several algorithms have been developed for addressing DCOPs, and the most well known basic frameworks are centered around:

- the Asynchronous Distributed Optimization (ADOPT) algorithm based on an opportunistic agent strategy maintaining DFS trees on the constraint graph (Modi *et al.* 2005; Ali, Koenig, & Tambe 2005),
- the DPOP algorithm based on variable elimination along such DFS trees (Petcu & Faltings 2006),
- the Asynchronous Partial Overlay (APO) algorithm, based on merging neighboring subproblems found in conflict (Mailler & Lesser 2004),
- depth first search traversals with branch and bound, also involving DFS trees (Chechetka & Sycara 2006).

Several hybrids of these basic approaches are also known, and we note that the use of DFS trees is common among the most efficient versions. Trade-offs between the different algorithms have been often discussed and DPOP is known to perform well on problems with small induced width, but in general has an exponential space complexity. A way to hybridize the idea behind APO with other solvers is suggested in (Petcu, Faltings, & Mailler 2007).

ADOPT is the first proposed asynchronous algorithm, and has only a polynomial space complexity. Several types of modifications were shown over time to bring large improvements in ADOPT. In particular both, switching to a depth first search traversal strategy and sending feedback to earlier agents, were separately shown to bring order of magnitude improvements to ADOPT (Chechetka & Sycara 2006; Silaghi & Yokoo 2006). The two modifications to ADOPT can be combined and future research is needed to check how the improvements that they provide do compose.

In this article we explore a further improvement to ADOPT, showing how one can dynamically detect the relevant DFS tree of a problem using the inferences of ADOPT itself. We show that ADOPT can start to solve a problem without knowing a DFS tree compatible with the original order on agents. It only integrates a constraint in the DFS tree when the constraint is first used to infer a cost. Therefore it maintains a smaller DFS tree, at least during an initial stage of the search. Together with an improved policy for sending optional messages, this is shown to bring improvements in a hybrid algorithm enabling agents to announce costs to higher priority neighbors in the constraint graph.

2 ADOPT and ADOPT-ing

Without loss of generality, a distributed constraint optimization problem (DCOP) is commonly defined by a set of agents $A_1, ..., A_n$, each agent A_i controlling a variable x_i

and enforcing constraints with other variables. Each constraint violation is associated with a cost. The goal is to assign the variables such as to minimize the total cost.

ADOPT (Modi *et al.* 2005) is an asynchronous complete DCOP solver, which is guaranteed to find an optimal solution. Here, we only show a brief description of ADOPT. Please consult (Modi *et al.* 2005) for more details. First, ADOPT organizes agents into a Depth-First Search (DFS) tree, in which constraints are allowed between a variable and any of its ancestors or descendants, but not between variables in separate sub-trees.

ADOPT uses three kinds of messages: VALUE, COST, and THRESHOLD. A VALUE message communicates the assignment of a variable from ancestors to descendants that share constraints with the sender. When the algorithm starts, each agent takes a random value for its variable and sends appropriate VALUE messages. A COST message is sent from a child to its parent, which indicates the estimated lower bound of the cost of the sub-tree rooted at the child. Since communication is asynchronous, a cost message contains a context, i.e., a list of the value assignments of the ancestors. The THRESHOLD message is introduced to improve the search efficiency. An agent tries to assign its value so that the estimated cost is lower than the given threshold communicated by the THRESHOLD message from its parent. Initially, the threshold is 0. When the estimated cost is higher than the given threshold, the agent opportunistically switches its value assignment to another value that has the smallest estimated cost. Initially, the estimated cost is 0. Therefore, an unexplored assignment has an estimated cost of 0. A cost message also contains the information of the upper bound of the cost of the sub-tree, i.e., the actual cost of the sub-tree. When the upper bound and the lower bound meet at the root agent, then a globally optimal solution has been found and the algorithm is terminated.

If the components of a COST message of ADOPT are bundled together, one obtains a simplified type of valued nogood (where only the justification is missing from the valued nogoods structures proposed by (Dago & Verfaillie 1996)), and the operations of ADOPT are simplified versions of the typical inference rules available for valued nogoods. If one uses instead the full version of the valued nogoods as proposed by Dago & Verfaille then the nogood messages can be sent to any predecessor (Silaghi & Yokoo 2006). That version of ADOPT is called Asynchronous Distributed OPTimization with inferences based on valued nogoods (ADOPT-ing)[1], and has two variants: ADOPT-dos and ADOPT-aos. In ADOPT-dos an agent sends a valued nogood to *all its ancestors in the DFS tree* for which it is relevant and for which it does not have a better nogood. In ADOPT-aos agents are totally ordered and an agent sends a valued nogood to *all its predecessors* for which it is relevant and for which it does not have a *better* nogood. In ADOPT-ing agents compute separately nogoods for each prefix of the ordered list of their predecessors. Both variants bring similar improvements in simulated time. However, ADOPT-dos has the weakness of requiring to know the DFS tree in advance, while ADOPT-aos loads the network with a very large number of concurrent messages (whose handling also leads to heavier local computations).

The variants of ADOPT-ing were differentiated using a notation **ADOPT-$\mathcal{DON}$** where $\mathcal{D}$ shows the destinations of the messages containing valued nogoods (a, d, or p), $\mathcal{O}$ marks the used optimization criteria for deciding a *better* nogood (only one such criteria was available, o), and $\mathcal{N}$ specifies the type of nogoods employed (the simplified valued nogoods, n, or the full version of valued nogoods, s). ADOPT-*pon* therefore indicates the original ADOPT.

Valued Nogoods. A nogood, $\neg N$, specifies a set N of assignments that conflict with existing constraints. Valued nogoods have the form $[R, c, N]$ and are an extension of classical nogoods (Dago & Verfaillie 1996). Each valued nogood has a *set of references to a conflict list of constraints (SRC)* R and a cost c. The cost specifies the minimal cost of the constraints in the conflict list R given the assignments of the nogood N. 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, $\overline{N} = \{x_1, ..., x_t\}$.

A valued nogood $[R, c, N \cup \langle x_i, v\rangle]$ applied to a value v of a variable x_i is referred to as the cost assessment (CA) of that value and is denoted (R, 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 in minimization DCOPs by sum-inference and min-resolution to obtain new nogoods (Dago & Verfaillie 1996).

Sum-inference. A set of cost assessments of type (R_i, v, c_i, N_i) for a value v of some variable, where $\forall i, j : i \neq j \Rightarrow R_i \cap R_j = \emptyset$, 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 (R, v, c, N) such that $R = \cup_i R_i$, $c = \sum_i(c_i)$, and $N = \cup_i N_i$.

Min-resolution. Assume that we have a set of cost assessments for x_i of the form (R_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 $[R, c, N]$ such that $R = \cup_i R_i$, $c = \min_i(c_i)$ and $N = \cup_i N_i$.

3 Basic Ideas

Continuing to use the notation ADOPT-$\mathcal{DON}$ for variants of ADOPT-ing, here we add a new possible value for $\mathcal{D}$, namely Y. Y stands for *all ancestors in a DFS tree* (as with d) but for *a dynamically discovered* DFS tree. Also, with Y, optional nogood messages are only sent when the target of the payload valued nogood is identical to the destination of the message. The *target* of a valued nogood is the position of the lowest priority agent among those that proposed an assignment referred by that nogood.

Let us now assume that at the beginning, the agents only know the address of the agents involved in their constraints (their neighbors), as in ABT (Yokoo *et al.* 1992; Bessiere *et al.* 2005). A DFS tree is *compatible* with a given total order on nodes, if the parent of a node precedes that node in the

[1] Originally ADOPT-ng (Silaghi & Yokoo 2006).

given total order.

We first show a method of computing a DFS tree compatible with a given total order on nodes in a preprocessing phase. Next, we consider a way for dynamically discovering the DFS tree during the search process.

procedure initPreprocessing*()* **do**
1.1 $ancestors \leftarrow neighboringpredecessors$;
foreach A_j *in* $ancestors$ **do**
1.2 **send DFS**($ancestors$) to A_j;
1.3 parent $\leftarrow$ last agent in $ancestors$;
when *receive* **DFS***(induced) from* A_t **do**
1.4 **if** *(*predecessors in $induced$*)* $\not\subseteq$ $ancestors$ **then**
1.5 $ancestors \leftarrow ancestors \cup$(predecessors in $induced$);
foreach A_j *in* $ancestors$ **do**
1.6 **send DFS**($ancestors$) to A_j;
1.7 parent $\leftarrow$ last agent in $ancestors$;

Algorithm 1: Preprocessing for discovery of DFS tree

Preprocessing for computing the DFS tree Each agent only has to perform the procedure in Algorithm 1. Algorithm 1 can be used for preprocessing the distributed problem. Each agent maintains a list with its $ancestors$ and starts executing the procedure **initPreprocessing**. The first step consists of initializing its $ancestors$ list with the neighboring predecessors (Line 1.1). The obtained list is broadcast to the known ancestors using a dedicated message named **DFS** (Line 1.2). On receiving a **DFS** message from A_t, an agent discards it when the parameter is a subset of its already known ancestors (Line 1.4). Otherwise the new ancestors induced because of A_t are inserted in the $ancestors$ list (Line 1.5). The new elements of the list are broadcasted to all ancestors (Line 1.6). The parent of an agent is the last ancestor (Lines 1.3,1.7).

Lemma 1 *Algorithm 1 computes a DFS tree compatible with a problem equivalent to the initial DCOP.*

Proof. Let us insert in the initial constraint graph of the DCOP a new total constraint (constraint allowing everything) for each link between an agent and its parent computed by this algorithm. A constraint allowing everything does not change the problem therefore the obtained problem is equivalent to the initial DCOP. Note that the arcs between each agent and its parent define a tree.

Now we can observe that there exists a DFS traversal of the graph of the new DCOP that yields the obtained DFS tree. Take three agents A_i, A_j, and A_k such that A_i is the obtained parent of both A_j and A_k. Our lemma is equivalent to the statement that no constraint exists between subtrees rooted by A_j and A_k (given the arcs defining parent relations).

Let us assume (trying to refute) that an agent $A_{j'}$ in the subtree rooted by A_j has a constraint with an agent $A_{k'}$ in the subtree rooted by A_k. Symmetry allows us to assume without loss of generality that $A_{k'}$ precedes $A_{j'}$. Therefore $A_{j'}$ includes $A_{k'}$ in its $ancestors$ list and sends it to its parent, which propagates it further to its parent, and so on to all ancestors of $A_{j'}$. Let $A_{j''}$ be the highest priority ancestor of $A_{j'}$ having lower priority than $A_{k'}$. But then $A_{j''}$ will set $A_{k'}$ as its parent (Lines 1.3,1.7), making $A_{k'}$ an ancestor of $A_{j'}$. This contradicts the assumption that $A_{k'}$ and $A_{j''}$ are in different subtrees of A_i. □

Note that for any given total order on agents, Algorithm 1 returns a single compatible DFS tree. This tree is built by construction, adding only arcs needed to fit the definition of a DFS tree. The removal of any of the added parent links leads to breaking the DFS-tree property, as described in the proof of the Lemma. We infer that Algorithm 1 obtains the smallest DFS tree compatible with the initial order (conclusion supported by the next Lemma).

Lemma 2 *If the total order on the agents is compatible with a known DFS tree of the initial DCOP, then all agent-parent arcs defined by the result of the above algorithm correspond to arcs in the original graph (rediscovering the DFS tree).*

Proof. Assume (trying to refute) that an obtained agent-parent relation, A_i–A_j, corresponds to an arc that does not exist in the original constraint graph (for the lowest priority agent A_i obtaining such a parent). The parent A_k of A_i in the known DFS tree must have a higher or equal priority than A_j; otherwise A_i (having A_k in his $ancestors$) would chose it as the parent in Algorithm 1. If A_k and A_j are not identical, it means that A_i has no constraint with A_j in the original graph (otherwise, the known DFS would not be correct). Therefore, A_j was received by A_i as an induced link from a descendant A_t which had constraints with A_j (all descendants being defined by original arcs due to the assumption). However, if such a link exists between a descendant A_t and A_j, then the known DFS tree would be incorrect (since in a DFS pseudo-tree all predecessor neighbors of one's descendants must be ancestors of oneself). This refutes the assumption and proves the Lemma. □

The preprocessing algorithm terminates, and the maximal casual chain of messages it involves has a length of at most n. That is due to the effort required to propagate ancestors from the last agent to the first agent. All messages travel only from low priority agents to high priority agents, and therefore the algorithm terminates after the messages caused by the agents in leaves reach the root of the tree[2].

Dynamic detection of DFS trees Intuitively, detecting a DFS tree in a preprocessing phase has three potential weaknesses which we can overcome. The first drawback is that it necessarily adds a preprocessing of up to n sequential messages. Second, it requires an additional termination detection algorithm. Third, it uses all constraints up-front, while some of them may be irrelevant for initial assignments of the agents (and shorter trees can be used to speed up search in the initial stages). Here we show how we address these issues in our next technique.

[2]Or roots of the forest.

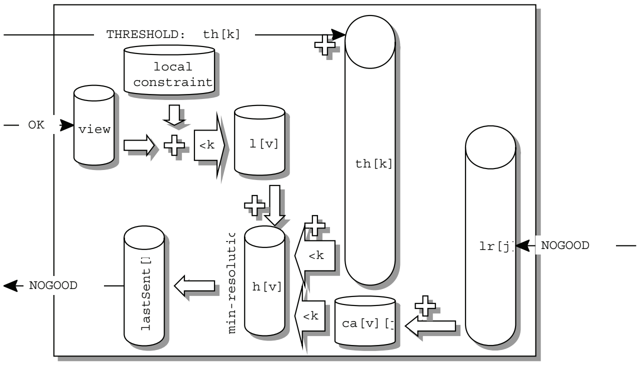

Figure 1: Schematic flow of data through the different data structures used by an agent A_i in ADOPT-ing.

Therefore, we propose to build a DFS tree only for the constraints used so far in the search. In the version ADOPT-Y$_{--}$, agents do not start initializing their $ancestors$ with all neighboring predecessors, but with the empty set. Neighboring predecessors are added to the $ancestors$ list only when the constraint defining that neighborhood is actually used to increase the cost of a valued nogood[3]. On such an event, the new $ancestor$ is propagated further as on a receipt of new induced ancestors with a **DFS** message in Algorithm 1. The handling of **DFS** messages is also treated as before. The dynamic detection is run concurrently with the search and integrated with the search, thus circumventing the mentioned weaknesses of the previous version based on preprocessing.

Another problem consists of dynamically detecting the children nodes, and how descendants are currently grouped in subtrees by the dynamic DFS tree. In our solution, A_i groups agents A_k and A_t in the same subtree if it detects that its own descendants in the received lists of induced links from A_k and A_t do intersect. This is done as follows. A check is performed each time when there is a new agent A_u in the lists of induced links received from a descendant A_k. If A_u was not a known descendant, a new subtree is created for it. Otherwise, the previous subtree containing A_u is merged with the subtree containing A_k. Also, a new subtree is created for each agent from which we receive a nogood and that was not previously known as a descendant. The data structure employed by an agent A_i for this purpose consists of a vector of n integers called $subtrees$. $subtrees[j]$ holds the ID of the subtree containing A_j, or 0 if A_j is not currently considered to be a descendant of A_i. Each agent generates a different ID for each of its subtrees.

4 The ADOPT-Yos algorithm

Data Structures. Besides its $ancestors$ and $subtrees$ arrays, 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.

ADOPT-ing also used a structure called lvn containing optional nogoods that increased the space complexity of the algorithm while having absolutely no effect on efficiency (as shown by our experiments). We propose to discard it.

The flow of data through these data structures of an agent A_i is illustrated in Figure 1. 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 4). 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 $A_1, ..., A_k$.

Pseudocode. The pseudocode showing how the dynamic detection is integrated in ADOPT-ing is given in Algorithm 2. No change is performed to the other procedures of ADOPT-ing. The procedures for dynamically building the DFS tree are inserted at Lines 2.1, 2.22, 2.23 and 2.25. Induced ancestors are received at Line 2.1 or detected al Line 2.22 and are propagated further at Lines 2.23 and 2.25. Please note that the *induced ancestors* need to be attached only to the first message towards each agent after a change to $ancestors$ (not to all messages).

The procedure described in the following remark is used in the proof of termination and optimality.

Remark 1 *The order of combining CAs to get h at Line 2.17 matters. To compute h[v]:*

1. *h[v]=sum-inference*$_{t\in[i+1,n]}$*(ca[v][t]).*
2. *Add l[v]: h[v]=sum-inference(h[v], l[v]).*
3. *Add threshold: h[v]=sum-inference(h[v], th[*]).*

[3] i.e., when a message is sent to that neighboring agent.

```
when receive nogood(rvn, t, inducedLinks) from A_t do
    insert new predecessors from inducedLinks in ancestors, on change making sure interested predecessors will be (re-)sent nogood messages;
    foreach assignment a of linked variable x_j in rvn do
        integrate(a)
    lr[t]:=rvn;
    if (an assignment in rvn is outdated) then
        if (some new assignment was integrated now) then
            check-agent-view();
        return;
    foreach assignment a of non-linked variable x_j in rvn do
        send add-link(a) to A_j;
    foreach value v of x_i such that rvn_|v is not ∅ do
        vn2ca(rvn, i, v) → rca (a CA for value v of x_i);
        ca[v][t]:=sum-inference(rca,ca[v][t]);
        update h[v];
    check-agent-view();
procedure check-agent-view() do
    for every ancestor A_j in the current DFS tree do
        for every(v ∈ D_i) update l[v] and recompute h[v];
                //with nogoods using only {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) or (j is parent)) then
                    add j to ancestors (updating parent);
                    send nogood(vn,i,ancestors) to A_j;
                    lastSent[j] = vn;
                on new ancestors, send nogood(∅,i,ancestors) to each ancestor not yet announced;
    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 2: Procedures of A_i in ADOPT-Yos

Note that (Silaghi & Yokoo 2006) proposed a more complicated scheme for ADOPT-dos that also exploited known DFS trees by first combining nogoods coming from the same subtrees and then combining those intermediary results. However our experiments show that particular optimization to have only a small (1%) effect on efficiency.

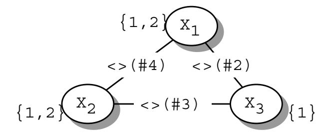

Figure 2: A DCOP with three agents and three inequality constraints. The fact that the cost associated with not satisfying the constraint $x_1 \neq x_2$ is 4, is denoted by the notation (#4). The cost for not satisfying the constraint $x_2 \neq x_3$ is 3.

5 Example and Proof

Assume a problem that involves three agents A_1, A_2, A_3. First, agents A_2 and A_3 discover the relevance of links A_1-A_2 and A_1-A_3. These define a DFS of depth 1. The search can perform a lot of efficient progress with this shallow intermediary DFS. Some time later, A_3 discovers the relevance of link A_2-A_3. The new tree is A_1-A_2-A_3 (depth 2), which is deeper and slower than the previous tree. Fortunately earlier we have done lots of progress, when A_3's reasoning effort went only straight to A_1 without cascading through A_2.

Let us now explore in more detail a possible run of ADOPT-ing's version ADOPT-Yos on the problem in Figure 2. A trace is shown in Figure 3. Identical messages sent simultaneously to several agents are grouped by displaying the list of recipients. In the messages of Figure 3, SRCs are represented as Boolean values in an array of size n. A value at index i in the array of SRCs set to T signifies that the constraints of A_i are used in the inference of that nogood. The agents start selecting values for their variables and announce them to interested lower priority agents. 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. Similarly, A_2 sends an **ok?** message to A_3 proposing x_2=2.

A_3 detects a conflict with x_1, inserts A_1 in its *ancestors* list, and sends a nogood with cost 2 to A_1 (message 3). A_1 answers the received nogood by switching its assignment to a value with lower current estimated value, x_1=2 (message 4). A_2 reacts by switching x_2 to its lowest cost value, x_2=1 (message 5). A_3 detects a conflict with x_2 and inserts A_2 in its *ancestors* list, which becomes $\{A_1, A_2\}$. In general a change to *ancestors* is announced to any prior ancestor whose prefix changes, but here the prefix of A_1 does not change and no such announcement is needed. A_3 also anounces the conflict to A_2 using the **nogood** message 6. This nogood received by A_2 is combined by min-resolution with the nogood locally infered by A_2 for its value 2 due to the constraint $x_1 \neq x_2$(#4). That inference also prompts the insertion of A_1 in the ancestors list of A_2. The obtained nogood is therefore sent to A_1 using message 7. A_1 and later A_2 switch their assignments the the values with the lowest cost, attaching the latest nogoods received for those values as threshold nogoods (messages 8, 9 and 10). At this moment the system reaches quiescence.

We note that without the dynamic DFS tree detection, the nogood in message 3 would also be sent to agent A_2. In general, such additional messages generate a cascade of message exchanges and modify the locally stored nogoods of

1. A_1	**ok?**$\langle x_1, 1\rangle$ $\longrightarrow$	A_2, A_3
2. A_2	**ok?**$\langle x_2, 2\rangle$ $\longrightarrow$	A_3
3. A_3	**nogood**$([\|F, F, T\|, 2, \langle x_1, 1\rangle], 3, \{A_1\})$ $\longrightarrow$	A_1
4. A_1	**ok?**$\langle x_1, 2\rangle$ $\longrightarrow$	A_2, A_3
5. A_2	**ok?**$\langle x_2, 1\rangle$ $\longrightarrow$	A_3
6. A_3	**nogood**$([\|F, F, T\|, 3, \langle x_2, 1\rangle], 3, \{A_1, A_2\})$ $\longrightarrow$	A_2
7. A_2	**nogood**$([\|F, T, T\|, 3, \langle x_1, 2\rangle], 2, \{A_1\})$ $\longrightarrow$	A_1
8. A_1	**ok?**$\langle x_1, 1\rangle$ $\longrightarrow$	A_2
9. A_1	**ok?**$\langle x_1, 1\rangle$**threshold** $[\|F, F, T\|, 2, \langle x_1, 1\rangle]$ $\longrightarrow$	A_3
10. A_2	**ok?**$\langle x_2, 2\rangle$ $\longrightarrow$	A_3

Figure 3: Trace of ADOPT-Yos on the problem in Figure 2.

involved agents.

Lemma 3 (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 costs of the elements of its h cannot be modified in an infinite cycle due to incoming valued nogoods.*

Proof. Valued nogoods that are updated solely by sum-inference have costs that can only increase (which can happen only a finite number of times). For a given cost, 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 CAs used to infer h will not be modified any longer and therefore h will no longer be modified. □

Corollary 3.1 *If ADOPT-ing uses the procedure in Remark 1, then for a given agent-view, the elements of the array h for that agent cannot be modified in an infinite cycle.*

Remark 2 *Since lr contains the last received valued nogoods, that array is updated by replacement with recently received nogoods, without sum-inference. Therefore, it cannot be used directly to infer h (optimization unfortunately proposed in (Silaghi & Yokoo 2006)).*

Theorem 4 *ADOPT-Yos terminates in finite time.*

Proof. Given the list of agents $A_1, ..., A_n$, define the suffix of length m of this list as the last m agents.

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

Induction step. Let us now assume that the induction assertion is true for a suffix of k 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 cost increases. Costs for CAs associated with the values of A_{n-k} will eventually stop being modified as a consequence of Lemma 3. □

Theorem 5 *The ADOPT-Yos algorithm is optimal.*

Proof. Our algorithm implements all the features used to prove optimality of ADOPT-ing in (Silaghi & Yokoo 2006). Therefore, the proof presented there applies here as is. Since it is lengthy we do not replicate it here and kindly ask readers to retrieve it from (Silaghi & Yokoo 2006). □

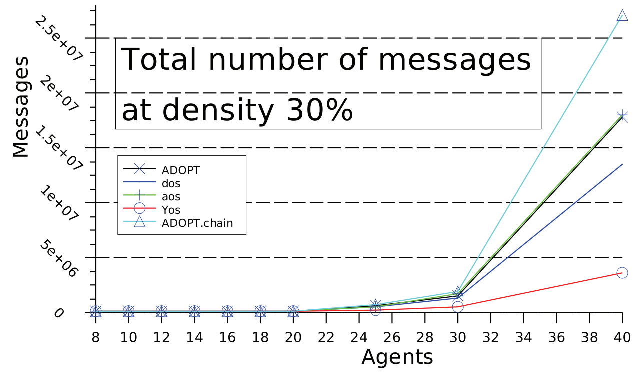

Figure 4: Total number of messages.

6 Experiments

We performed experiments comparing the simulated time required by our implementation of ADOPT, ADOPT-aos, ADOPT-dos, and ADOPT-Yos. The experiments use the set of random problems distributed with the original implementation of ADOPT (Modi *et al.* 2005). Those sets of problems contain instances containing between 8 and 40 agents, and densities of the constraint graph (ratio between the number of binary constraints and the total number of pairs of variables) of 20% and 30% and 40%. The set of problems at density 40% made available with ADOPT (Modi *et al.* 2005) is smaller that the sets for other densities, the largest instances having only 25 agents. At each problem size the set contains 25 instances. The order of the variables was set compatible with the DFS tree built by ADOPT. We also tested the efficiency of running ADOPT on the obtained chain of variables rather than on the DFS tree (version denoted ADOPT.chain). The simulator used random latencies for messages, generated uniformly between 150ms and 250ms (sample variation for an international link over optical cable (Neystadt & Har'El 1997)).

A metric we display here is the total number of messages exchanged, showing the load placed on the network bandwidth (see Figure 4). Figure 4 reveals that ADOPT-Yos overcomes the lack of knowledge of a DFS tree, problem present in ADOPT-aos and ADOPT.chain. At 40 agents ADOPT-Yos requires 4 times less messages than ADOPT-dos, and 7.5 times less messages than ADOPT.chain (5 times less messages than ADOPT on the known DFS tree).

Another measure we use is the equivalent non-

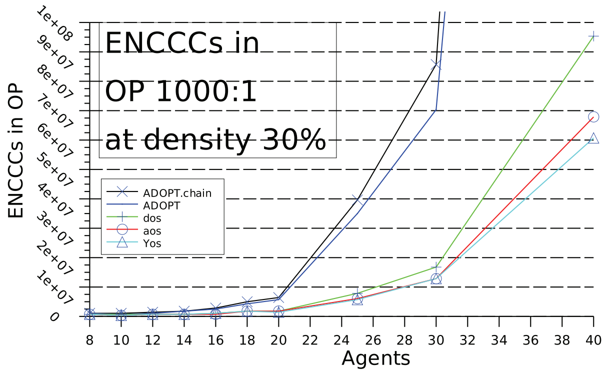

Figure 5: ENCCCs in the operating point (ENCCCOPs).

concurrent constraint checks (ENCCCs), recently employed in (Chechetka & Sycara 2006; Silaghi & Faltings 2004). The graph in Figure 5 shows the behavior in the *operating point* (OP) assumed for developing the algorithm (ENCCCOPs). Our OP is based on assuming intercontinental computations over the Internet, where the latency of a message is approximately equivalent to the cost associated with 1000 constraint checks (the simulator used between .1ms to 1ms per constraint check)[4]. At 40 agents ADOPT-Yos is shown to compete well with previous algorithms being 12% faster than ADOPT-aos, and 14 times faster than ADOPT.chain (11 times faster than ADOPT).

The improvements on the large problems at density 40% are very similar to the ones at density 30%, while the improvements on problems at density 20% are smaller (there ADOPT-Yos is only 5% faster than the second best algorithm which at density 20% is ADOPT-dos).

7 Conclusions

We proposed an algorithm for dynamically detecting the DFS tree in ADOPT, based on constraints used so far in the reasoning process. The new version, ADOPT-Yos, differentiates itself from earlier optimization algorithms by the fact that agents do not initially need to know neither a DFS tree on the constraint graph of the problem, nor the addresses of other agents than the ones with which they share constraints. This property was held previously only by the ABT algorithm for distributed constraint satisfaction. ADOPT-Yos also sends a nogood with optional **nogood** messages only to the most relevant ancestor.

The ADOPT-Yos can find shorter DFS trees during the initial steps of the algorithm, while some constraints did not yet prove useful given visited combinations of assignments. We showed that the techniques proposed in ADOPT-Yos bring an order of magnitude improvements in the total number of messages over the previous winner for this measure (ADOPT-dos), even without knowing a DFS tree in advance. They also bring some smaller (12%) improvements in simulated time over the previous winner (ADOPT-aos). Thus, ADOPT-Yos not only unifies the advantages of ADOPT-aos and ADOPT-dos, but also improves over them.

[4]The ratio obtained is even larger, 10000:1, after additional optimization of our code.

8 Acknowledgement

This research was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research (B), 17300049, 2006–2007.

References

Ali, S.; Koenig, S.; and Tambe, M. 2005. Preprocessing techniques for accelerating the DCOP algorithm ADOPT. In *AAMAS*.

Bessiere, C.; Brito, I.; Maestre, A.; and Meseguer, P. 2005. Asynchronous backtracking without adding links: A new member in the abt family. *Artificial Intelligence* 161:7–24.

Chechetka, A., and Sycara, K. 2006. No-commitment branch and bound search for distributed constraint optimization. In *AAMAS*.

Dago, P., and Verfaillie, G. 1996. Nogood recording for valued constraint satisfaction problems. In *ICTAI*.

Greenstadt, R.; Pearce, J.; Bowring, E.; and Tambe, M. 2006. Experimental analysis of privacy loss in dcop algorithms. In *AAMAS*, 1024–1027.

Mailler, R., and Lesser, V. 2004. Solving distributed constraint optimization problems using cooperative mediation. In *AAMAS*, 438–445.

Marcellino, F. M.; Omar, N.; and Moura, A. V. 2007. The planning of the oil derivatives transportation by pipelines as a distributed constraint optimization problem. In *IJCAI-DCR Workshop*.

Modi, P., and Veloso, M. 2005. Bumping strategies for the multiagent agreement problem. In *AAMAS*.

Modi, P. J.; Shen, W.-M.; Tambe, M.; and Yokoo, M. 2005. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *AIJ* 161.

Neystadt, J., and Har'El, N. 1997. Israeli internet guide (iguide). http://www.iguide.co.il/isp-sum.htm.

Petcu, A., and Faltings, B. 2006. ODPOP: an algorithm for open/distributed constraint optimization. In *AAAI*.

Petcu, A.; Faltings, B.; and Mailler, R. 2007. PC-DPOP: A new partial centralization algorithm for distributed optimization. In *IJCAI*.

Silaghi, M.-C., and Faltings, B. 2004. Asynchronous aggregation and consistency in distributed constraint satisfaction. *Artificial Intelligence Journal* 161(1-2):25–53.

Silaghi, M.-C., and Yokoo, M. 2006. Nogood-based asynchronous distributed optimization. In *AAMAS*.

Walsh, T. 2007. Traffic light scheduling: a challenging distributed constraint optimization problem. In *DCR*.

Yokoo, M.; Durfee, E. H.; Ishida, T.; and Kuwabara, K. 1992. Distributed constraint satisfaction for formalizing distributed problem solving. In *ICDCS*, 614–621.