ADOPT-ing: Unifying Asynchronous Distributed Optimization with Asynchronous Backtracking

Marius C. Silaghi
Florida Institute of Technology
MSILAGHI@FIT.EDU

Makoto Yokoo
Kyushu University
YOKOO@IS.KYUSHU-U.AC.JP

Abstract

This article presents an asynchronous algorithm for solving Distributed Constraint Optimization problems (DCOPs). The proposed technique unifies asynchronous backtracking (ABT) and asynchronous distributed optimization (ADOPT) where valued nogoods enable more flexible reasoning and more opportunities for communication, leading to an important speed-up. While feedback can be sent in ADOPT by COST messages only to one predefined predecessor, our extension allows for sending such information to any relevant agent. The concept of valued nogood is an extension by Dago and Verfaille of the concept of classic nogood that associates the list of conflicting assignments with a cost and, optionally, with a set of references to culprit constraints.

DCOPs have been shown to have very elegant distributed solutions, such as ADOPT, distributed asynchronous overlay (DisAO), or DPOP. These algorithms are typically tuned to minimize the longest causal chain of messages as a measure of how the algorithms will scale for systems with remote agents (with large latency in communication). ADOPT has the property of maintaining the initial distribution of the problem. To be efficient, ADOPT needs a preprocessing step consisting of computing a Depth-First Search (DFS) tree on the constraint graph. Valued nogoods allow for automatically detecting and exploiting the best DFS tree compatible with the current ordering. To exploit such DFS trees it is now sufficient to ensure that they exist. Also, the inference rules available for valued nogoods help to exploit schemes of communication where more feedback is sent to higher priority agents. Together they result in an order of magnitude improvement.

1. Introduction

Distributed Constraint Optimization (DCOP) is a formalism that can model problems distributed due to their nature. These are problems where agents try to find assignments to a set of variables that are subject to constraints. The reason for the distribution of the solving process comes from the assumption that only a subset of the agents has knowledge of each given constraint. Nevertheless, in DCOPs it is assumed that agents try to maximize their cumulated satisfaction by the chosen solution. This is different from other related formalisms where agents try to maximize the satisfaction of the least satisfied among them (Yokoo, 1993). It is also different from formalisms involving self-interested agents (which wish to maximize their own utility individually).

The application of distributed constraint optimization framework to modeling and solving multi-agent meeting scheduling problems is detailed in (Modi & Veloso, 2005; Franzin, Rossi, E.C., & Wallace, 2004; Maheswaran, Tambe, Bowring, Pearce, & Varakantham, 2004; Sultanik, Modi, & Regli, 2006). The application to Distributed Generator Maintenance is described in (Petcu & Faltings, 2006a). An application to oil pipelines is described in (Marcellino, Omar, & Moura, 2007), while an application to traffic light scheduling is described in (Walsh, 2007). These problems have in common the fact that some constraints are originally distributed among involved agents and are difficult to centralize due to privacy or due to other structural issues.

Several synchronous and asynchronous distributed algorithms have been proposed for solving DCOPs in a distributed manner. Since a DCOP can be viewed as a distributed version of the common centralized Weighted Constraint Satisfaction Problems (WCSPs / Σ-VCSP) (Bistarelli, Fargier, Montanari, Rossi, Schiex, & Verfaillie, 1996; Bistarelli, Montanari, & Rossi, 1995; Schiex, Fargier, & Verfaillie, 1995; Bistarelli, Montanari, Rossi, Schiex, Verfaillie, & Fargier, 1999), it is normal that successful techniques for WCSPs were ported to DCOPs. However, the effectiveness of such techniques has to be evaluated from a different perspective (and using different measures) as imposed by the new requirements. 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 (Silaghi & Faltings, 2002; Wallace & Silaghi, 2004; Yokoo, Suzuki, & Hirayama, 2002; Silaghi & Mitra, 2004).

A synchronous algorithm, synchronous branch and bound, was the first known distributed algorithm for solving DCOPs (Hirayama & Yokoo, 1997). Stochastic versions have also been proposed (Zhang & Wittenburg, 2002). From the point of view of efficiency, a distributed algorithm for solving DCOPs is typically evaluated with regard to applications to agents on the Internet, namely, where latency in communication is significantly more time consuming than local computations (in most algorithms). A common measure that represents this assumption well is given by the number of cycles of a simulator that lets each agent in turn process all the messages that it receives (Yokoo, Durfee, Ishida, & Kuwabara, 1992). Within the mentioned assumption, for real solvers this measure is shown to be equivalent to the longest causal chain of sequential messages, as used in (Silaghi, Sam-Haroud, & Faltings, 2001c). However, we also provide results computed in a scenario simulating random latency in communication. Sometimes, local computation time is also factored into the evaluation metric by weighting the computation associated with each constraint check as a fraction (e.g., between one tenth and one millionth) of the latency of a message (Yokoo et al., 1992; Silaghi & Faltings, 2004; Chechetka & Sycara, 2006). The current value of this fraction for the Internet is around one thousand, estimating approximately 10^{-4} seconds/constraint-check and 0.5 seconds/message. The technological trend predicts improvements of the computational speeds of machines (reducing the duration of constraint-checks), while communication latencies are stable, being close to the physical limit set by the speed of light (Neystadt & Har'El, 1997). Therefore, the fraction is expected to be reduced even further in the future, reducing the relevance of constraint checks.

From the point of view of this measure, a very efficient currently existing DCOP solver is DPOP (Petcu & Faltings, 2005b, 2005a), which is linear in the number of variables. However, that algorithm generally has message sizes and local computation costs that are exponential in the induced width of a chosen depth-first search tree of the constraint graph of the problem. This clearly invalidates the assumptions that lead to the acceptance of the number of cycles as an efficiency measure. Some of the agents are also very disadvantaged in DPOP with respect to their privacy (Greenstadt, Pearce, Bowring, & Tambe, 2006). Effort is currently directed toward reducing these drawbacks (Petcu & Faltings, 2006b).

Two other algorithms competing as efficient solvers of DCOPs are the asynchronous distributed optimization (ADOPT) and the distributed asynchronous overlay (DisAO). DisAO works by incrementally joining the sub-problems owned by agents found in conflict (Mailler & Lesser, 2004). ADOPT can be described as a parallel version of (Iterative Deepening) A* (Silaghi, Landwehr, & Larrosa, 2004). While DisAO is typically criticized for its significant abandon of the maintenance of the local distribution of the problem at the first conflict (and expensive local computations invalidating the above assumptions as for DPOP (Davin & Modi, 2005; Maheswaran et al., 2004; Ali, Koenig, & Tambe, 2005)), ADOPT can be criticized for its strict message pattern that only provides reduced reasoning opportunities. ADOPT works with orderings on agents dictated by some Depth-First Search tree on the constraint graph, and allows cost communication from an agent only to its parent node.

It is easy to construct huge problems whose constraint graphs are forests and which can be easily solved by DPOP (in linear time), but are unsolvable with the other known algorithms. It is also easy to construct relatively small problems whose constraint graph is full and therefore require unacceptable (exponential)

space with DPOP, while being easily solvable with algorithms like ADOPT, e.g., for the trivial case where all tuples are optimal with cost zero.

In this work we address the aforementioned critiques of ADOPT, showing that it is possible to define a message scheme based on a type of nogoods, called *valued nogoods* (Dago & Verfaillie, 1996; Dago, 1997), which besides automatically detecting and exploiting the DFS tree of the constraint graph coherent with the current order, help to 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 (Stallman & Sussman, 1977). A basic version of the valued nogoods consist of associating each nogood with a cost, namely a cost limit violated due to the assignments of the nogood. Valued nogoods that are associated with a list of culprit constraints produce important efficiency improvements. Each of these incremental concepts is described in the following sections.

We start by defining the general DCOP problem, followed by introduction of the immediately related background knowledge consisting of the ADOPT algorithm and use of Depth-First Search trees in optimization. In Section 2.4 we also describe valued nogoods together with the simplified version of valued global nogoods. In Section 3 we present our new algorithm that unifies ADOPT with the older Asynchronous Backtracking (ABT). The algorithm is introduced by first describing the goals in terms of new communication schemes to be enabled. Then the data structures needed for such communication are explored together with the associated flow of data. Finally the pseudo-code and the proof of optimality are provided before discussing other existing and possible extensions. Several different versions mentioned during the description are compared experimentally in the last section.

2. Background and Preliminaries

Now we introduce in more detail the distributed constraint optimization problems, the ABT and ADOPT algorithms, as well as the theory behind the versions of valued nogoods used in this work.

2.1 Distributed Constraint Optimization

Constraint Satisfaction Problems (CSPs) are described by a set X of n variables and a set C of m constraints on the possible combinations of assignments to these variables with values from their domains, D. A common extension of the CSP framework for modeling applications to minimization of time, space, or number of resources is known as Weighted CSP (Bistarelli et al., 1996, 1999; Larrosa, 2002).

Definition 1 (WCSP (Larrosa, 2002; Bistarelli et al., 1996)) *A Weighted CSP is defined by a triplet of sets* (X, D, C) *and a bound* B. X *and* D *are defined as in CSPs. In contrast to CSPs,* $C=\{c_1, ..., c_m\}$ *is a set of functions,* $c_i : D_{i_1} \times ... \times D_{i_{m_i}} \to \mathbb{N}^\infty$ *where* m_i *is the arity of* c_i.
Its solution is $\epsilon* = \underset{\epsilon \in D_1 \times ... \times D_n}{\operatorname{argmin}} \sum_{i=1}^m c_i(\epsilon|_{X_i})$, *if* $\sum_{i=1}^m c_i(\epsilon * |_{X_i}) < B$, *where* $X_i = D_{i_1} \times ... \times D_{i_{m_i}}$.

The specification of the bound B in the above definition is common only to a few versions (Larrosa, 2002), and other articles use its default value $B = \infty$ (Bistarelli et al., 1999)[1]. WCSPs are an important instance of more general frameworks, such as Valued CSPs (VCSPs) and Semiring CSPs (SCSPs) (Bistarelli et al., 1999) that we do not address here[2]. A framework equivalent to WCSPs, where each violation of a constraint of the CSP is associated with a cost, is called $\Sigma - VCSP$ (Schiex et al., 1995). Other equivalent frameworks are known, such as the Partial CSPs of (Freuder & Wallace, 1992). Distributed Constraint Optimization Problems, the problems addressed in this work, are a generalization of WCSPs.

---

1. DCOP definitions could also include it to help specify branch and bound solvers.
2. Algorithmic ideas presented in this paper could be easily reused with a fuzzy version of DCOP (with a + replaced by a MIN in the definition of the optimal solution), where the goal would be to minimize the highest unsatisfaction among agents.

Definition 2 (DCOP) *A distributed constraint optimization problem (DCOP), is defined by a set of agents $A_1, A_2, ..., A_n$, and a set X of variables, $x_1, x_2, ..., x_n$. Each agent A_i has a set of k_i functions $C_i = \{c_i^1, ..., c_i^{k_i}\}$, $c_i^j : X_{i,j} \to \mathbb{R}_+$, $X_{i,j} \subseteq X$, where only A_i knows C_i. We assume that x_i can only take values from a domain $D_i = \{1, ..., d\}$.*

Denoting with ϵ an assignment of values to all the variables in X, the problem is to find $\underset{\epsilon}{\text{argmin}} \sum_{i=1}^n \sum_{j=1}^{k_i} c_i^j(\epsilon_{|X_{i,j}})$.

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

By $\epsilon_{|X_{i,j}}$ we denote the projection of the set of assignments in ϵ on the set of variables in $X_{i,j}$.

Our idea can be easily applied to general weighted CSPs. For example, our simulator of the ADOPT-ing algorithm for DCOPs (presented later) is a solver of weighted CSPs. This solver receives a WCSP as input and returns an optimal solution (after converting the WCSP to a DCOP in an intermediary step). This solver is parametrized with a random number generator that decides the latencies of each message. For constant latencies (the implementation based on rounds), a centralized forward checking-like paradigm is obtained. Namely, each new assignment of a variable x_k is followed in the next step by cost inferences on the domain of all variables $x_j, j > k$ (which in their turn are then immediately backward propagated in the following round to the variable x_k, and to other earlier variables).

Reducing general DCOPs to the used framework. With the aforementioned definition of DCOPs one often says that each agent A_i controls one variable, x_i, since it knows all constraints between x_i and previous variables $x_j, j < i$. There exist several common extensions to this definition of DCOPs providing for several variables per agent, and for agents that hold constraints involving only variables that they do not control. While non-trivial optimizations are possible when considering such frameworks (Yokoo & Hirayama, 1998; Silaghi & Faltings, 2004), any solution for the version discussed here can be easily applied to those cases.

- The case of several variables per agent can be addressed by aggregating all variables of an agent into a new variable, which can take as value any tuple allowed by the constraints between the original variables. Another straightforward solution is to replace each original agent by several new agents, one for each variable (where the original agents act under several false names (Modi, Shen, Tambe, & Yokoo, 2005)).

- The case of agents holding constraints involving only variables that they do not control can be modeled by adding a new variable for each such constraint, adding it to that constraint (with a total relation), assigning it to be controlled by the corresponding agent, and then applying one of the approaches mentioned for several variables per agent.

Example 2.1 *Consider a problem with 3 variables x_1, x_2, and x_3 and 3 constraints c_{12} (between x_1 and x_2), c_{23} (between x_2 and x_3), and c_{31} (between x_3 and x_1), where Alice knows c_{12}, Bob knows c_{23}, and Carol knows c_{31}.*

This problem can be modeled as a DCOP with 4 agents. Alice uses two agents, A_1 and A_2 (typically called pseudo-agents in ADOPT). Bob uses the agent A_3 and Carol uses an agent A_4. The new variable x_4 of the agent A_4 is involved in a ternary constraint c_{431} with x_1 and x_3. The constraint c_{431} is constructed such that its projection on x_1 and x_2 is c_{31}.

2.2 DFS-trees

The primal graph of a DCOP is the graph having the variables in X as nodes and having an arc for each pair of variables linked by a constraint (Dechter, 2003). A Depth-First Search (DFS) tree associated with a DCOP is a spanning tree generated by the arcs used for first visiting each node during some Depth-First Traversal of its primal graph. DFS trees were first successfully used for distributed constraint satisfaction problems in (Collin, Dechter, & Katz, 2000). The property exploited there is that separate branches of the DFS-tree are completely independent once the assignments of common ancestors are decided. Two examples of DFS trees for a DCOP primal graph are shown in Figure 1.
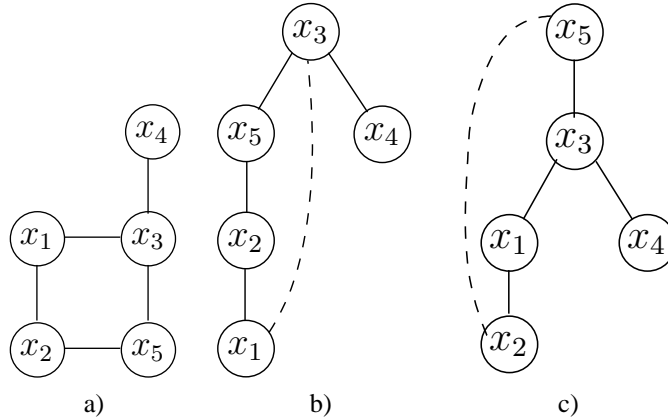
Figure 1: For a DCOP with primal graph depicted in (a), two possible DFS trees (pseudo-trees) are (b) and (c). Interrupted lines show constraint graph neighboring relations not in the DFS tree.

Nodes directly connected to a node in a primal graph are said to be its *neighbors*. In Figure 1.a, the neighbors of x_3 are $\{x_1, x_4, x_5\}$. The *ancestors* of a node are the nodes on the path between it and the root of the DFS tree, inclusively. In Figure 1.b, $\{x_3, x_5\}$ are ancestors of x_2. x_3 has no ancestors. If a variable x_i is an ancestor of a variable x_j, then x_j is a *descendant* of x_i. For example, in Figure 1.b, $\{x_1, x_2\}$ are descendants of x_5.

2.3 ADOPT and ABT

ADOPT. 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.

ABT. Distributed constraint satisfaction problems are special cases of DCOPs where the constraints c_i^j can return only values in $\{0, \infty\}$. The basic asynchronous algorithm for solving distributed constraint satisfaction problems is asynchronous backtracking (ABT) (Yokoo, Durfee, Ishida, & Kuwabara, 1998). ABT uses a total priority order on agents where agents announce new assignments to lower priority agents using **ok?**
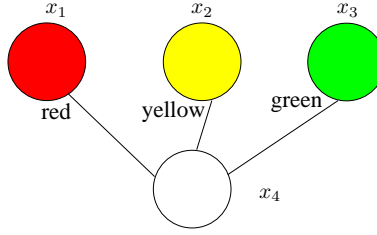
Figure 2: MIN resolution on valued global nogoods

messages, and announce conflicts to lower priority agents using **nogood** messages. New dependencies created by dynamically learned conflicts are announced using **add-link** messages. An important difference between ABT and ADOPT is that, in ABT, conflicts (the equivalents of cost) can be freely sent to any higher priority agent.

ABT performs a kind of forward checking. Immediately after a variable is instantiated, after the latency of a message, all future variables are immediately checked for values consistent with that assignment (as their controlling agents receive the new assignment in parallel). Therefore ABT has a transparent look-ahead behavior implicit in its asynchronism and feedback mechanisms. This explanation is experimentally confirmed by the fact that explicit forward checking does not improve over ABT (Meseguer & Jiménez, 2000).

2.4 Cost of nogoods

Previous flexible algorithms for solving distributed constraint satisfaction problems exploit the inference power of nogoods (e.g., ABT, AWC, ABTR (Yokoo et al., 1992, 1998; Silaghi, Sam-Haroud, & Faltings, 2001b))[3]. A nogood $\neg N$ stands for a set N of assignments that was proven impossible, by inference, using constraints. 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, $\overline{N} = \{x_1, ..., x_t\}$.

2.4.1 VALUED GLOBAL NOGOODS

In order to apply nogood-based algorithms to DCOP, one redefines the notion of nogoods as follows. First, we attach a value to each nogood obtaining a *valued global nogood*. These are a simplified version of Dago&Verfaille's valued nogoods introduced next, and are basically equivalent to the content of COST messages in ADOPT.

Definition 3 (Valued Global Nogood) *A valued global nogood has the form* $[c, N]$*, and specifies that the (global) problem has cost at least* c*, given the set of assignments* N *for distinct variables.*

Example 2.2 *For the graph coloring problem in Figure 2 (assume it has a constraint* $x_1 \neq x_4$ *with weight 10), a possible valued global nogood is* $[10, \{(x_1, r), (x_4, r)\}]$*. It specifies that if* $x_1 = r$ *and* $x_4 = r$ *then there exists no solution with a cost lower than* 10*.*

Given a valued global nogood $[c, (\langle x_1, v_1 \rangle, ..., \langle x_t, v_t \rangle)]$, one can infer a *global cost assessment (GCA)* for the value v_t from the domain of x_t given the assignments $S = \langle x_1, v_1 \rangle, ..., \langle x_{t-1}, v_{t-1} \rangle$. This GCA is denoted (v_t, c, S) and is semantically equivalent to an applied valued global nogood (i.e., the inference):

$$(\langle x_1, v_1 \rangle, ..., \langle x_{t-1}, v_{t-1} \rangle) \rightarrow (\langle x_t, v_t \rangle \text{ has cost } c).$$

---

3. Other algorithms, like AAS, exploit generalized nogoods (i.e., extensions of nogoods to sets of values for a variable), and the extension of the work here for that case is suggested in (Silaghi, 2002).

Remark 1 *Given a valued global nogood $[c, N]$ known to some agent, that agent can infer the GCA (v, c, N) for any value v from the domain of any variable x, where x is not assigned in N, i.e., $x \notin \overline{N}$.*

For example, if A_3 knows a valued global nogood $[10, \{(x_1, r), (x_2, y)\}]$, then it can infer for the value r of x_3 the GCA $(r, 10, \{(x_1, r), (x_2, y)\})$.

Proposition 1 (min-resolution) *Given a minimization WCSP, assume that we have a set of GCAs of the form (v, c_v, N_v) that has the property of containing exactly one GCA 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 one can resolve a new valued global nogood: $[\min_v c_v, \cup_v N_v]$.*

Example 2.3 *For the graph coloring problem in Figure 2 (weighted constraints are not shown), x_1 is colored red (r), x_2 yellow (y) and x_3 green (g). Assume that the following valued global nogoods are known for each of the values $\{r, y, g\}$ of x_4:*

(r): $[10, \{(x_1, r), (x_4, r)\}]$, *obtaining for x_4 the GCA $(r, 10, \{(x_1, r)\})$*

(y): $[8, \{(x_2, y), (x_4, y)\}]$, *obtaining for x_4 the GCA $(y, 8, \{(x_2, y)\})$*

(g): $[7, \{(x_3, g), (x_4, g)\}]$, *obtaining for x_4 the GCA $(g, 7, \{(x_3, g)\})$*

By min-resolution on these GCAs, one obtains the valued global nogood $[7, \{(x_1, r), (x_2, y), (x_3, g)\}]$, meaning that given the coloring of the first 3 nodes, there is no solution with (global) cost lower than 7.

Min-resolution can be applied to valued global nogoods:

Corollary 1.1 *Assume $\mathcal{S}$ is a set of nogoods associated with the variable x_i, such that for each $[c_v, S_v]$ in $\mathcal{S}$, $\exists \langle x_i, v \rangle \in S_v$. If $\mathcal{S}$ contains exactly one global valued nogood $[c_v, S_v]$ for each value v in the domain of variable x_i of a minimization WCSP, then one can resolve a new valued global nogood: $[\min_v c_v, \cup_v (S_v \setminus \langle x_i, v \rangle)]$.*

2.4.2 DAGO AND VERFAILLE'S VALUED NOGOODS

We would like to allow free sharing of nogoods between agents. The operator for aggregating the weights of constraints in DCOPs is $+$, which is not idempotent (i.e., in general $a + a \neq a$). Therefore a constraint cannot be duplicated and implied constraints cannot be added straightforwardly without modifying the semantic of the problem (which was possible with distributed CSPs (Schiex et al., 1995; Bistarelli et al., 1999))[4]. Two solutions are known. One solution is based on DFS trees (used by ADOPT), while the second is based on justifications. We will use both of them.

Remark 2 (DFS sub-trees) *Given two GCAs (v, c'_v, S'_v) and (v, c''_v, S''_v) for a value v in the domain of variable x_i of a minimization WCSP, if one knows that the two GCAs are inferred from different constraints, then one can infer a new GCA: $(v, c'_v + c''_v, S'_v \cup S''_v)$. This is similar to what ADOPT does to combine cost messages coming from disjoint problem sub-trees (Modi, Tambe, Shen, & Yokoo, 2002; Collin et al., 2000).*

This powerful reasoning can be applied when combining a nogood obtained from the local constraints with a valued nogood received from other agents (and obtained solely by inference from other agents' constraints). When a DFS tree of the constraint graph is used for constraining the message pattern as in ADOPT, this powerful inference applies, too.

The question is how to determine that the two GCAs are inferred from different constraints in a more general setting. This can be done by tagging cost assessments with the identifiers of the constraints used to infer them (the justifications of the cost assessments).

Definition 4 *A set of references to constraints (**SRC**) is a set of identifiers, each for a distinct constraint.*

---

4. The aggregation method for fuzzy CSPs (a kind of VCSPs) (Schiex et al., 1995) is MIN, being idempotent. Therefore inferred global valued nogoods can be freely added in that framework.

Note that several constraints of a given problem description can be composed in one constraint (in a different description of the same problem).[5]

SRCs help to define a generalization of the concept of *valued global nogood* named *valued nogood* (Dago & Verfaillie, 1996; Dago, 1997).

Definition 5 (Valued Nogood) *A valued* nogood *has the form* $[R, c, N]$ *where R is a set of references to constraints having cost at least c, given a set of assignments, N, for distinct variables.*

Valued nogoods are generalizations of valued global nogoods. Valued global nogoods are valued nogoods whose SRCs contain the references of all the constraints.

Once we decide that a nogood $[R, c, (\langle x_1, v_1 \rangle, ..., \langle x_i, v_i \rangle)]$ will be applied to a certain variable x_i, we obtain a cost assessment tagged with the set of references to constraints R[6], denoted $(R, v_i, c, (\langle x_1, v_1 \rangle, ..., \langle x_{i-1}, v_{i-1} \rangle))$.

Definition 6 (Cost Assessment (CA)) *A cost assessment of variable x_i has the form (R, v, c, N) where R is a set of references to constraints having cost with lower bound c, given a set of assignments N for distinct variables where the assignment of x_i is set to the value v.*

As for valued nogoods and valued global nogoods, cost assessments are generalizations of global cost assessments.

Remark 3 *Given a valued nogood $[R, c, N]$ known to some agent, that agent can infer the CA (R, v, c, N) for any value v from the domain of any variable x, where x is not assigned in N, i.e., where $x \notin \overline{N}$.*

For example, denoting by $J_{4,7}$ the reference to the constraint between variables x_4 and x_7, if A_6 knows the valued nogood $[\{J_{4,7}\}, 10, \{(x_2, y), (x_4, r)\}]$, then it can infer the CA $(\{J_{4,7}\}, b, 10, \{(x_2, y), (x_4, r)\})$ for the value b of x_6.

We can now detect and perform the desired powerful reasoning on valued nogoods and/or CAs coming from disjoint sub-trees, mentioned in Remark 2.

Proposition 2 (sum-inference (Dago & Verfaillie, 1996; Dago, 1997)) *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 $N = \cup_i N_i$.*

Example 2.4 *For the graph coloring problem in Figure 3, x_1 is colored red, x_2 yellow, x_3 green, and x_4 red.*
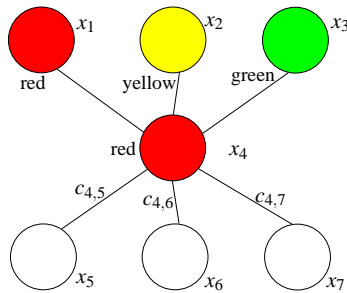

Figure 3: SUM-inference resolution on CAs

Let the justification referring a set of constraints including $c_{i,j}$ and $c_{j,k}$ be denoted $J_{i,j,k}$. Assume that the following valued nogoods are known for (x_4, r):

---

5. For privacy, a constraint can be represented by several constraint references and several constraints of an agent can be represented by a single constraint reference.
6. This is called a *valued conflict list* in (Silaghi, 2002).

- $[\{J_{4,5,2}\}, 5, \{(x_2, y), (x_4, r)\}]$ *obtaining CA* $(\{J_{4,5,2}\}, r, 5, \{(x_2, y)\})$

- $[\{J_{4,6,1}\}, 7, \{(x_1, r), (x_4, r)\}]$ *obtaining CA* $(\{J_{4,6,1}\}, r, 7, \{(x_1, r)\})$

- $[\{J_{4,7,2}\}, 9, \{(x_2, y), (x_4, r)\}]$ *obtaining CA* $(\{J_{4,7,2}\}, r, 9, \{(x_2, y)\})$

Also assume that based on x_4's constraint with x_1, one has obtained for $\langle x_4, r \rangle$ the following valued nogood:

- $[\{J_{1,4}\}, 10, \{(x_1, r), (x_4, r)\}]$ *obtaining CA* $(\{J_{1,4}\}, r, 10, \{(x_1, r)\})$

Then, by sum-inference on these CAs, one obtains for x_4 the CA $[\{J_{1,4}, J_{4,5,2}, J_{4,6,1}, J_{4,7,2}\}, r, 31, \{(x_1, r), (x_2, y)\}]$, *meaning that given the coloring of the first 2 nodes, coloring x_4 in red leads to a cost of at least 31 for the constraints $\{J_{1,4}, J_{4,5,2}, J_{4,6,1}, J_{4,7,2}\}$.*

Remark 4 (sum-inference for valued nogoods) *Sum inference can be similarly applied to any set of valued nogoods with disjoint SRCs and compatible assignments. The result of combining a set of nogoods $[R_i, c_i, S_i]$ is $[\cup_i R_i, \sum_i c_i, \cup_i S_i]$. This can also be extended to the case where assignments are generalized to sets (Silaghi, 2002).*

The min-resolution proposed for GCAs translates straightforwardly for CAs as follows.

Proposition 3 (min-resolution (Dago & Verfaillie, 1996; Dago, 1997)) *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 $N = \cup_i N_i$.*

Example 2.5 *For the graph coloring problem in Figure 2, x_1 is colored red, x_2 yellow, and x_3 green. Assume that the following valued nogoods are known for the values of x_4:*

(r): $[\{J_{1,4}\}, 10, \{(x_1, r), (x_4, r)\}]$ *obtaining CA* $(\{J_{1,4}\}, r, 10, \{(x_1, r)\})$

(y): $[\{J_{2,4}\}, 8, \{(x_2, y), (x_4, y)\}]$ *obtaining CA* $(\{J_{2,4}\}, y, 8, \{(x_2, y)\})$

(g): $[\{J_{3,4}\}, 7, \{(x_3, g), (x_4, g)\}]$ *obtaining CA* $(\{J_{3,4}\}, g, 7, \{(x_3, g)\})$

By min-resolution on these CAs, one obtains the valued global nogood $[\{J_{1,4}, J_{2,4}, J_{3,4}\}, 7, \{(x_1, r), (x_2, y), (x_3, g)\}]$, meaning that given the coloring of the first 3 nodes there is no solution with cost lower than 7 for the constraints $\{J_{1,4}, J_{2,4}, J_{3,4}\}$.

As with valued global nogoods, the min-resolution could be applied directly to valued nogoods:

Corollary 3.1 (min-resolution on nogoods) *From a set of valued nogoods $[R_v, c_v, S_v]$ (such that $\exists v, \langle x_i, v \rangle \in S_v$) containing exactly one valued nogood for each value v in the domain of variable x_i of a minimization problem, one can resolve a new valued nogood: $[\cup_v R_v, \min_v c_v, \cup_v (S_v \setminus \langle x_i, v \rangle)]$.*

3. ADOPT with nogoods

We now present a distributed optimization algorithm whose efficiency is improved by exploiting the increased flexibility brought by the use of valued nogoods. The algorithm can be seen as an extension of both ADOPT and ABT, and will be denoted Asynchronous Distributed OPTimization with inferences based on valued nogoods (ADOPT-ing).

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 can also be used to ensure this requirement (i.e., older assignments and older nogoods for the currently proposed value are discarded).
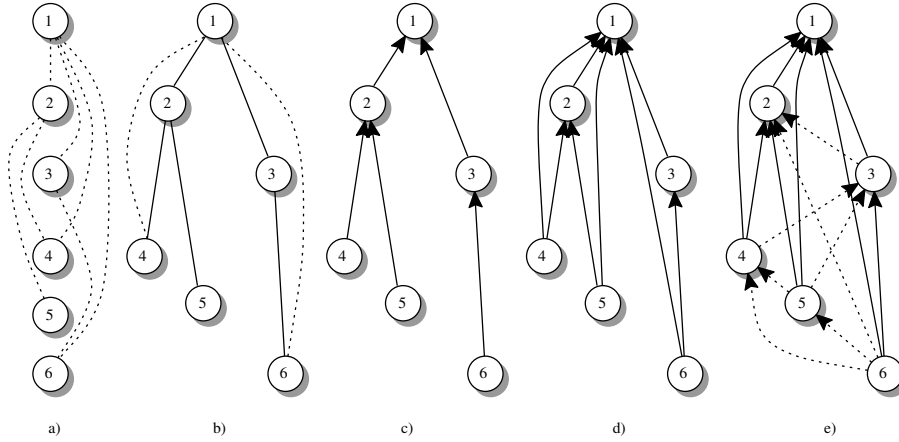
Figure 4: Feedback modes in ADOPT-ing. a) a constraint graph on a totally ordered set of agents; b) a DFS tree compatible with the given total order; c) ADOPT-p⌐: sending valued nogoods only to parent (graph-based backjumping); d) ADOPT-d⌐, ADOPT-D⌐, and ADOPT-Y⌐: sending valued nogoods to any ancestor in the tree; e) ADOPT-a⌐ and ADOPT-A⌐: sending valued nogoods to any predecessor agent.

3.1 Exploiting DFS trees for Feedback

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 and the DFS tree is discovered dynamically. We also propose hybrid versions that can exploit an existing DFS tree. We have identified 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. The obtained algorithm 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 (Modi et al., 2005), namely communication of costs to higher priority parents.

The versions of ADOPT-ing described in this article are differentiated using the notation **ADOPT-$\mathcal{DON}$**. $\mathcal{D}$ shows the destinations of the messages containing valued nogoods. $\mathcal{D}$ has one of the values $\{p, a, A, d, D, Y\}$ where p stands for *parent*, a and A stand for *all predecessors*, while d, D and Y stand for *all ancestors in a DFS tree*. The difference between the upper and lower case versions is further explained in Section 3.2. Y is as D but for a dynamically discovered DFS tree. $\mathcal{O}$ marks the optimization criteria used by sum-inference in selecting a nogood when the alternatives have the same cost. For now we use a single criterion, denoted o, which consists of choosing the nogood whose target has the highest priority. $\mathcal{N}$ 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 4. The total order on agents is described in Figure 4.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 4.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 $\mathcal{O}$ and $\mathcal{N}$ respectively. Sometimes the underscores are dropped to improve readability. This method of announcing conflicts based on the constraint graph is depicted in Figure 4.c and is related to the classic Graph-based Backjumping algorithm (Dechter, 1990; Hamadi & Bessière, 1998).

In Figure 4.d we depict the nogoods exchange schemes used in ADOPT-d__, ADOPT-D__ and ADOPT-Y__ where, for each new piece of information, valued nogoods are separately computed to be sent to each of the ancestors in the currently known DFS tree. These schemes are enabled by valued nogoods and are shown by experiments to bring large improvements. As for the initial version of ADOPT, the proof 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 4.e depicts the basic versions of ADOPT-ing, when a chain of agents is used instead of a DFS tree (ADOPT-a__ and ADOPT-A__), and 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 show that valued nogoods help to remove the redundant dependencies whose introduction would otherwise be expected from such messages. The only mandatory nogood messages for guaranteeing optimality in this scheme are the ones to the immediately previous agent (parent in the chain). 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. Note that the original ADOPT can also run on any chain of the agents, but our experiments show that its efficiency decreases by 20% when it does not know the shortest DFS tree compatible with the current order, and is an order of magnitude less efficient than any of these two variants of ADOPT-ing. When no DFS tree is known in advance, ADOPT-Y__ slightly improves on ADOPT-A__ as it dynamically detects a tree with reduced depth.

3.2 Differentiating ADOPT-a and ADOPT-d from ADOPT-A and ADOPT-D

The valued nogood computed for a 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 according to Remark 3. Exceptions appear in the case where such nogoods cannot be composed by sum-inference with some valued nogoods of A_k. The 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 cost. In other words, 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. From this point of view ADOPT-Y__ works like ADOPT-D__ but on the dynamically found DFS tree.

Example 3.6 *Consider the DCOP whose DFS tree is depicted in Figure 4. Assume that the next nogoods are inferred by A_6:*

[1] For the prefix of predecessor agents $\{A_1\}$: $[\{J_{6,1}\}, 2, (x_1 = 2)]$

[2] For the prefix of predecessor agents $\{A_1, A_2, A_3\}$: $[\{J_{6,1}, J_{6,3}\}, 3, (x_1 = 2)(x_3 = 2)]$

With ADOPT-a, nogood [1] is sent to $\{A_1, A_2\}$, and nogood [2] to $\{A_3, A_4, A_5\}$. With ADOPT-A, nogood [1] is sent to A_1, and nogood [2] to its target A_3 and to the predecessor agent A_5.

Example 3.7 *Consider the DCOP whose DFS tree is depicted in Figure 5. Assume that the next nogoods are inferred by A_7:*

[1] For the prefix of ancestor agents $\{A_1\}$: $[\{J_{7,1}\}, 2, (x_1 = 2)]$

[2] For the prefix of ancestor agents $\{A_1, A_3, A_6\}$: $[\{J_{7,1}, J_{7,6}\}, 3, (x_1 = 2)(x_6 = 2)]$
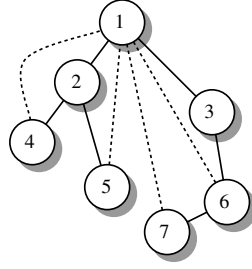
Figure 5: The constraint graph for a problem with 7 agents.

Nogood [1] can be used in inferences by all three ancestor agents $\{A_1, A_3, A_6\}$. Nogood [2] can be used only by A_6. In this situation, with ADOPT-d the agent A_7 sends nogood [1] to A_1 and A_3, and nogood [2] to A_6. With ADOPT-D, A_7 sends nogood [1] only to A_1, and nogood [2] to its target (and parent) A_6.

3.3 Dynamic Discovery of Compatible DFS Tree in ADOPT-Y

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. Finding a DFS tree of a constraint graph is different from the minimal cycle cutset problem, whose distributed solutions have been studied in the past (Jagota & Dechter, 1997). We address the problem of computing a DFS tree *compatible* with a given total order on nodes, namely where the parent of a node precedes that node in the given total order. However, not any given total order on the variables is compatible with a DFS tree of the constraint graph. Given an agreed total order on agents that unknowingly happens to be compatible with a DFS tree, it is relatively simple (less than n rounds) to find the compatible DFS tree. When a compatible DFS tree does not exist, our technique adds a small set of arcs (total constraints) that keep the problem equivalent to the original one and then returns a DFS tree compatible with the new graph.

procedure initPreprocessing*() ***do**

1.1 $ancestors \leftarrow neighboring predecessors$;

     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* $induced$*)* $\nsubseteq ancestors$ **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: Procedures of agent A_i during preprocessing for dynamic discovery of DFS tree.

Preprocessing for computing the DFS tree   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 broadcast to all interested ancestors, namely ancestors that will have these new elements as their ancestors (Line 1.6). The parent of an agent is the last ancestor (Lines 1.3,1.7).

Lemma 4 *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, if no constraint existed already. 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 sub-trees 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 sub-tree rooted by A_j has a constraint with an agent $A_{k'}$ in the sub-tree 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 sub-trees of A_i. $\square$

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. Therefore, we infer that Algorithm 1 obtains the smallest DFS tree compatible with the initial order.

Remark 5 *The trivial approach to using the DFS construction algorithm as a preprocessing technique also requires the detection of the termination, to launch ADOPT-D_ or ADOPT-d_ when the preprocessing terminates. Some of our techniques can be viewed as efficient ways to avoid such detection.*

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[7].

Lemma 5 *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 (Lines 1.3, 1.7). 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 have been incorrect (since in a DFS pseudo-tree all predecessor neighbors of one's descendants must be ancestors of oneself). This contradicts the assumption and proves the Lemma. $\square$

Remark 6 *If one knows that there exists a DFS tree of the initial constraint graph that is compatible with the order on agents, then the parent of each agent in that tree is its lowest priority predecessor neighbor.*

---

7. Or roots of the forest.

The agent can therefore compute its parent from the beginning without any message. This is at the basis of our implementation of ADOPT-D_, ADOPT-d_ and ADOPT-p_, where we know that the input order is compatible with a DFS tree (being the same order as the one used by ADOPT) but we do not bother providing the tree to the solver.

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 uses all constraints up-front while some of them may be irrelevant, at least for initial assignments of the agents (and shorter trees can be used to speed up search in the initial stages). Third, trivial DFS tree detection may also require an additional termination detection algorithm. Here we show how we address these issues in one of our next techniques.

Therefore, we propose to build a DFS tree only for the constraints used so far in the search. Therefore, agents in ADOPT-Y_ 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[8]. 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. The payload of the **DFS** messages is attached to **nogood** messages.

Another problem consists of dynamically detecting the children nodes and how descendants are currently grouped in sub-trees by the dynamic DFS tree. In our solution, A_i groups agents A_k and A_t in the same sub-tree 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 there is a new descendant agent A_u in the lists of induced links received from a descendant A_k. If A_u was not a previously known descendant of A_i, then A_u is inserted in the sub-tree of A_k. Otherwise, the previous sub-tree containing A_u is merged with the sub-tree containing A_k. Also, a new sub-tree 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 sub-tree containing A_j, or 0 if A_j is not currently considered to be a descendant of A_i. Each agent generates a different unique ID (positive number) for each of its sub-trees (e.g., by incrementing a counter).

Remark 7 *If agents start ADOPT-Y by inserting all their predecessor neighbors in their* ancestors *list, the algorithm becomes equivalent to ADOPT-D after less than n rounds.*

3.4 Data Structures

Besides the *ancestors* and *subtrees* structures of ADOPT-Y, 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.

---

8. More exactly, when a message is sent to that neighboring agent.
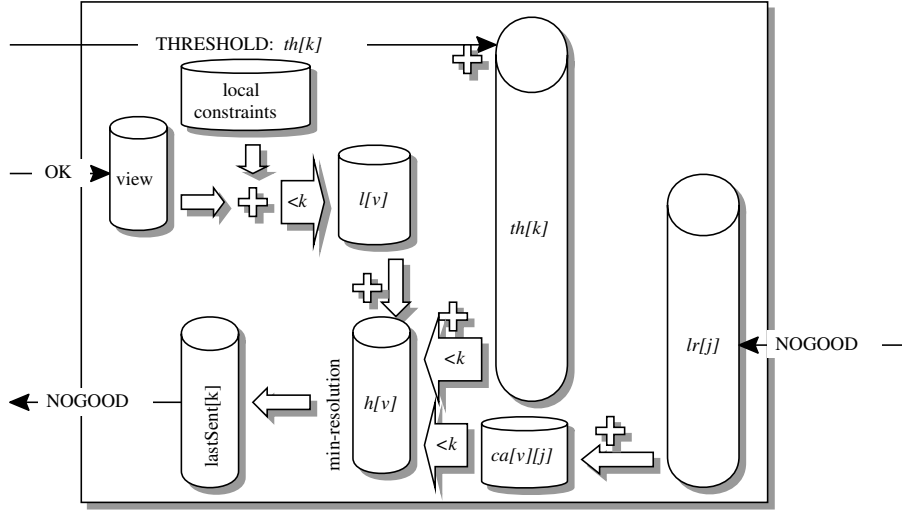
Figure 6: Schematic flow of data through the different data structures used by an agent A_i in ADOPT-ing.

- 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 (Silaghi, 2003a; Silaghi et al., 2004). 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 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.

3.5 Data flow in ADOPT-ing

The flow of data through these data structures of an agent A_i is illustrated in Figure 6. 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 3.4).

The $+$ sign at the meeting point of streams of valued nogoods or cost assessments shows that the streams are combined using sum-inference. The $\overset{\pm}{\Leftarrow}$ 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.

when *receive* **ok?**$(\langle x_j, v_j \rangle$, *tvn)* **do**

2.1	integrate($\langle x_j, v_j \rangle$);
2.2	**if** *(tvn no-null and has no old assignment)* **then**
2.3	k:=target(tvn); // threshold tvn as common cost;
2.4	th[k]:=sum-inference(tvn,th[k]);
2.5	check-agent-view();

when *receive* **add-link**$(\langle x_j, v_j \rangle)$ *from A_j* **do**

2.6	add A_j to outgoing_links;
2.7	**if** $(\langle x_j, v_j \rangle)$ is old, **send** new assignment to A_j;

when *receive* **nogood***(rvn, t, inducedLinks) from A_t* **do**

2.8	insert new predecessors from $inducedLinks$ in $ancestors$, on change making sure interested predecessors will be (re-)sent **nogood** messages; //needed only in ADOPT-Y_;
2.9	**foreach** *new assignment a of a linked variable x_j in rvn* **do**
2.10	integrate(a); // counters show newer assignment;
2.11	lr[t]:=rvn;
2.12	**if** *(an assignment in rvn is outdated)* **then**
2.13	**if** *(some new assignment was integrated now)* **then**
2.14	check-agent-view();
2.15	return;
2.16	**foreach** *assignment a of a non-linked variable x_j in rvn* **do**
2.17	send **add-link**(a) to A_j;
2.18	**foreach** *value v of x_i such that $rvn_{\vert v}$ is not $\emptyset$* **do**
2.19	$vn2ca(rvn, i, v) \rightarrow rca$ (a CA for the value v of x_i);
2.20	ca[v][t]:=sum-inference(rca,ca[v][t]);
2.21	update h[v] and retract changes to ca[v][t] if h[v]'s cost decreases;
2.22	check-agent-view();

Algorithm 2: Receiving messages of A_i in ADOPT-ing

3.6 ADOPT-ing pseudo-code and proof

The pseudo-code for the procedures in ADOPT-ing is given in Algorithms 2 and 3. To extract the cost of a CA, we introduce the function $cost()$, where $cost((R, v, c, N))$ returns c. The $min\_resolution(j)$ function applies the min-resolution over the CAs associated with 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 recomputes 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. In the current implementation, we recompute l and h at each call to $min\_resolution(j)$. An optimization is possible here, reusing the result[9] of computing $min\_resolution(k-1)$ in the computation of $min\_resolution(k)$ for $k < parent$ by adding only nogoods

---

9. From applying Step 2 of Remark 8.

on x_k to it. Experiments show that this brings minor 4% improvements in simulator time (local computations) on hard problems.

The $sum\_inference()$ function used in Algorithm 3 applies the sum-inference to its parameters whenever this is possible (it detects disjoint SRCs). Otherwise, it selects the nogood with the highest cost or the one whose lowest priority assignment has the highest priority (this has been previously used in (Bessiere, Brito, Maestre, & Meseguer, 2005; Silaghi et al., 2001b)). The function $vn2ca(vn, i)$ transforms a valued nogood vn in a cost assessment for x_i. Its inverse is function $ca2vn$. If vn has no assignment for x_i, then a cost assessment can be obtained according to Remark 3. The function $vn2ca(vn, i, v)$ translates vn into a cost assessment for the value v of x_i, using the technique in Remark 3 if needed. The function $target(N)$ gives the index of the lowest priority variable present in the assignment of nogood N. As with file expansion, when "*" is present in an index of a matrix, the notation is interpreted as the set obtained for all possible values of that index (e.g., ca[v][*] stands for $\{ca[v][t] \mid \forall t\}$). Given a valued nogood ng, the notation $ng_{|v}$ stands for vn2ca(ng) when ng's value for x_i is v, and $\emptyset$ otherwise.

3.6.1 PSEUDO-CODE

This sub-section explains line by line the pseudocode in Algorithms 2 and 3. Each agent A_i starts by calling the init() procedure in Algorithm 3, which at Line 3.1 initializes l with valued nogoods inferred from local (unary) constraints. The agent assigns x_i to a value with minimal local cost, crt_val (Line 3.2), announcing the assignment to lower priority agents in *outgoing_links* (Line 3.3). The *outgoing_links* of an agent A_i initially holds the address of the agents enforcing constraints that involve the variable x_i. The agents answer to any received message with the corresponding procedure in Algorithm 2: "**when** receive **ok?**," "**when** receive **nogood**," and "**when** receive **add-link**."

When a new assignment of a variable x_j is learned from **ok?** or **nogood** messages, valued nogoods based on older assignments for the same variables are discarded (Lines 2.1,2.10) by calling the function *integrate()* in Algorithm 3. Within this function, all valued nogoods (cost assignments) stored by the agent are verified and those that contain an old assignment of x_j, which is no longer valid, are deleted (Line 3.17). Any discarded element of ca is recomputed from lr. Namely, if a cost assessment $ca[v][t]$ is deleted in this process while $lr[t]$ remains valid, the agent attempts to apply the nogood in $lr[t]$ to the value v and the obtained cost assessment is copied in $ca[v][t]$ (Line 3.18). This application of the nogood $lr[t]$ to v is possible either if it contains $x_i = v$ or if it contains no assignment for the variable x_i of the current agent (Remark 3). Eventually the new assignment is stored in the agent-view (Line 3.19).

Further, when an **ok?** message is received, it is checked for valid threshold nogoods (Line 2.2). The target k of any such nogood, i.e., the position of the owner of the lowest priority variable, is extracted at Line 2.3 with a procedure called *target*, to detect the place where the nogood should be stored. The newly received threshold nogood is stored at $th[k]$ by sum-inference with the current nogood found there (Lines 2.4,3.21). If no nogood is found in $th[k]$, the new nogood is simply copied there (Line 3.20). If a nogood is already stored in $th[k]$, but its SRC intersects the one in the new nogood, then the behavior depends on the version of ADOPT-ing. Our pseudo-code illustrates the versions ADOPT-_o_, where the valued nogoods with the highest cost are retained (Line 3.22). In case of a tie, the one with the smallest target is maintained (Line 3.23) (Bessiere et al., 2005; Silaghi et al., 2001b).

After receiving a new value, like in ABT, the *check-agent-view* procedure is used to select a value or detect nogoods (Line 2.5). In this procedure, the agent first tries to compute a nogood for each of its predecessors (Line 3.4). For each such destination, a separate nogood is computed in l for each value v by considering only local constraints with that target agent and with its predecessors. Then, by considering these nogoods of l and all cost assessments in ca based only on assignments from the target agent and its predecessors, new elements of h are computed by sum-inference (Line 3.5). The order of the steps used in this computation is important for correctness and is described in detail later, in Remark 8. If all values of x_i have non-zero cost nogoods in h (Line 3.6), then all elements of h are combined via min-resolution and a nogood vn is obtained for the currently targeted destination (Line 3.7). However, the nogood vn is sent only if it is different from the last nogood sent to that same agent (Line 3.8). Repeating its sending would be

redundant since the recipient holds it in its lr vector. A further restriction is set with ADOPT-A__, ADOPT-D__, and ADOPT-Y__ where the nogood is sent only if the lowest priority variable involved in it is the same as the one controlled by the destination (Line 3.9). The nogood is always sent to the parent in the DFS tree (with ADOPT-d__, ADOPT-D__, and ADOPT-Y_) which is the immediate predecessor with ADOPT-a__ and ADOPT-A__. With ADOPT-Y__, when a nogood is sent for the first time to an agent A_k[10], A_k is added to the list $ancestors$ (Line 3.10). After the nogood is sent (Line 3.11), it is stored in $lastSent$ to help avoid immediate retransmission (Line 3.12). If some change was recently made to the $ancestors$ list, the change is propagated at Line 3.13 to all the ancestors that had not already been notified with **nogood** messages at Line 3.11.

The second part of the *check-agent-view* procedure deals with selecting opportunistically a value with the smallest estimated cost (Line 3.14), as common in ADOPT and ABT. We used the common mathematical notation $argmin_v(f(v))$ to denote a computation that returns the value v minimizing the function $f(v)$ passed as the parameter (here $cost(h[v])$). In case of a tie with the old value of x_i, our implementation of $argmin$ prefers to maintain the old value. If the value selected for x_i is different from the old value (Line 3.15), the new value is sent to all agents in $outgoing\_links$ (Line 3.16).

When **nogood** messages are received, in the ADOPT-Y__ version we first insert new received induced links into $ancestors$ (Line 2.8). If the set of $ancestors$ was changed by this operation, we set a flag to make sure that *check-agent-view* is eventually called and will propagate the change to all current ancestors. The agent checks if the transported nogood has newer assignments than the ones it already knows. A new assignment can reach an agent as part of a nogood before the corresponding **ok?** message. This can be handled in two ways:

i The original solution of ADOPT and ABT (Yokoo et al., 1998; Modi et al., 2005) is to consider any assignment in a nogood that is different from the assignment known for that variable as being invalid. Assignments are re-announced after each received valid message. Therefore, later retransmission[11] of the nogood triggered by this scheme is guaranteed to correctly deliver each nogood eventually.

ii The other scheme identifies new assignments in **nogood** messages as such, and validates the nogoods on their first reception. The mechanism was used in several versions of ABT (Silaghi & Faltings, 2004). It works by letting each agent maintain a separate counter for each variable. The counter is incremented when the assignment is changed and tags each sent assignment. Each agent stores the last value of the counter it sees for each variable. An agent detects a new assignment by comparing its tag with the previously seen value of that counter. Once detected (Line 2.9), new assignments in nogoods are integrated as on the arrival of their **ok?** message (Line 2.10).[12]

The last nogood received from some agent A_j is stored in $lr[j]$ (Line 2.11), such that it would not be lost as long as it is stored by A_j in its $lastSent$ (otherwise deadlocks could occur).[13] If some assignment in a nogood is considered old at Line 2.12 (with any mentioned scheme) the handling of the nogood is stopped and the nogood is discarded (Line 2.15). However, if some new assignment was integrated at Line 2.10, then the rest of the processing normally executed on **ok?** messages is performed by calling the *check-agent-view* procedure at Lines 2.13, 2.14.

If a received nogood contains a variable not previously involved in constraints with the variable of the agent (Line 2.16), an **add-link** message is sent to the agent owning that variable (Line 2.17) to announce the creation of a new link between the two agents (Line 2.6) and to request updates on the values of that variable (Line 2.7). In ADOPT-ing, the assignment received in the nogood is attached to the **add-link** message. This allows the owner of that variable to spare a message by not sending this assignment to A_i if the assignment is still valid.

---

10. Because the corresponding constraint increases for the first time the cost of the computed nogood.

11. Assuming no mechanism is used to block immediate retransmission of nogoods, such as our $lastSent$ structure.

12. Assignments having the same value are considered identical, even if their tag differs (allowing for re-using old nogoods).

13. Note that with the first scheme (i), where assignments are not tagged with counters, ADOPT-ing should not delete old nogoods from lr (which is done with the second scheme), but checks them when **ok?** messages are received.

An agent can receive a nogood where its variable is not present and therefore where the nogood can be applied to all its values. Valid nogoods are projected on all values of A_i (Lines 2.18,2.19), and the result is added to the corresponding cost assessments in ca using the sum-inference procedure (Line 2.20). It is possible that by the quirks of the impact of disjoint SRCs on sum-inference, the addition of a new nogood leads to the decrease of the cost of the obtained cost assessment for the corresponding value. We prefer to enforce a monotonic behavior by withdrawing changes to ca in such situations (Line 2.21). For this purpose, the evaluation of the modification of the cost is done by computing h as when messages are prepared for the parent in the DFS tree (or immediate predecessor). After integrating the new nogood, *check-agent-view* is called at Line 2.22 to infer new nogoods and to select the best value of x_i.

3.6.2 PROOF

Received nogoods are stored in matrices lr and th (Algorithm 2). A_i always sets its crt_val to the index with the lowest CA cost in vector h (preferring the previous assignment in case of ties). On each change that propagates to h, and for each ancestor A_j (or higher priority agent in versions not using DFS trees), the elements of h are recomputed separately by min-resolution(j) to generate new nogoods for A_j. The simultaneous generation and use of multiple nogoods is already known to be useful for the constraint satisfaction case (Yokoo & Hirayama, 1998).

The threshold valued nogood tvn delivered with **ok?** messages sets a common cost on all values of the receiver (see Remark 3), effectively setting a threshold on costs below which the receiver does not change its value. This achieves the effect of THRESHOLD messages in ADOPT.

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

Remark 8 *The order of combining CAs to get h at Line 3.5 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: $\forall v, s; h[v]$=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 cost given by the sum of the individual costs obtained for each DFS sub-tree.

   For case (b) we obtain h[v]=tmp[v].
   This makes sure that at quiescence the cost 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[*]).*

Note that method (a) at Step 1 can be applied only to ADOPT-Y$\_$, ADOPT-D$\_$, and ADOPT-d$\_$, while method (b) can be applied to all versions. Experiments show that, when applicable, method (a) works only slightly (i.e. 1%) better than method (b).

Lemma 6 (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 cost assessments used to infer h will not be modified any longer and therefore h will no longer be modified. $\quad\square$

Corollary 6.1 *If ADOPT-ing uses the procedure in Remark 8, then for a given agent-view, the elements of the array h for that agent cannot be modified in an infinite cycle.*

Remark 9 *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, it cannot be used directly to infer h.*

Note that with the described procedure, a newly arriving valued nogood can decrease the cost of certain elements of h (even if it does not decrease the cost of any of the elements from which h is computed). This is because, while increasing the cost of some element in ca, it can also modify its SRC and therefore forbid its composition by sum-inference with other cost assessments.

Remark 10 (Obtaining Monotonic Increase) *One can avoid the undesired aforementioned effect, where incoming nogoods decrease costs of elements 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 cost of $h[v]$ decreases, then the old content of $ca[v]$ can be restored. Each new valued nogood is used for updating lr. On each change to some element in ca, one has to add to ca the elements found in lr and coming from children in the DFS tree (if they do not lead to a decrease in the cost of h). Experiments show that this technique can bring a small improvement of up to 2% in the number of cycles.*

Intuitively, the convergence of ADOPT-ing can be noticed from the fact that valued nogoods can only monotonically increase valuation for each subset of the search space, and this has to terminate since such valuations can be covered by a finite number of values. If agents A_j, $j<i$ no longer change their assignments, valued nogoods can only monotonically increase at A_i for each value in D_i: costs of the nogoods only increase since they only change by sum-inference.

Lemma 7 *ADOPT-ing 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. 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 cost increases. By construction, costs for CAs associated with the values of A_{n-k} can only grow (see Remark 10). Even without the technique in Remark 10, costs for CAs associated with the values of A_{n-k} will eventually stop being modified as a consequence of Lemma 6. After values are proposed in turn and the smallest cost reaches its highest estimate, agent A_{n-k} selects the best value and reaches quiescence. The other agents reach quiescence according to the assumption of the induction step. $\quad\square$

Lemma 8 *The last valued nogoods sent by each agent additively integrate the non-zero costs of the constraints of all of the agent's successors (or descendants in the DFS tree when a DFS tree is maintained).*

procedure *init* **do**

3.1 $h[v] := l[v]$:=initialize CAs from unary constraints;

3.2 $crt\_val = argmin_v(cost(h[v]))$;

3.3   send **ok?**($\langle x_i, crt\_val \rangle$,$\emptyset$) to all agents in outgoing_links;

procedure *check-agent-view()* **do**

3.4   **for every** A_j with higher priority than A_i *(respectively ancestor in the DFS tree, when one is maintained)* **do**

3.5   **for every**($v \in D_i$) update $l[v]$ and recompute $h[v]$;
              // with valued nogoods using only instantiations of $\{x_1, ..., x_j\}$;

3.6   **if** *(h has non-null cost CA for all values of D_i)* **then**

3.7   vn:=min_resolution(j);

3.8   **if** *(vn $\neq$ lastSent[j])* **then**

3.9   **if** *((target(vn) == j) or (j is parent/immediate predecessor))* **then**

3.10   add j to $ancestors$ (updating parent);// for ADOPT-Y_;

3.11   send **nogood**(vn,i,$ancestors$) to A_j;

3.12   lastSent[j] = vn;

3.13   on new ancestors, send *nogood*($\emptyset$,i,$ancestors$) to each ancestor A_h not yet announced;

3.14 $crt\_val = argmin_v(cost(h[v]))$;

3.15   **if** *(crt_val changed)* **then**

3.16   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**

3.17   discard elements in ca, th, $lastSent$ and lr based on other values for x_j;

3.18   use $lr[t]|_v$ to replace each discarded $ca[v][t]$;

3.19   store $\langle x_j, v_j \rangle$ in agent-view;

function *sum-inference(vng1, vng2)*

3.20   **if** *either vng1 or vng2 has cost 0* **then**
           return the other one;

3.21   **if** *vng1 and vng2 have disjoint SRCs* **then**
           return the result of applying sum-inference on them;

3.22   **if** *vng1 and vng2 have different costs* **then**
           return the one with lower cost;

3.23   **if** *vng1 and vng2 have different targets* **then**
           return the one with smaller target;

       return *vng1*;

Algorithm 3: Procedures of A_i in ADOPT-ing

Proof.   At quiescence, each agent A_k has received the valued nogoods describing the costs of each of its successors (or descendants in the DFS tree when a DFS tree is maintained).

The lemma results by induction for an increasingly growing suffix of the list of agents (in the order used by the algorithm): It is trivial for the last agent.

Assuming that it is true for agent A_k, it follows that it is also true for agent A_{k-1} since adding A_{k-1}'s local cost to the cost received from its children in the tree (A_k for ADOPT-A_) will be higher (or equal when removing zero costs) than the result of adding A_{k-1}'s local cost to that of any descendants of those children. Respecting the order in Remark 8 guarantees that this value is obtained (according to the assumption of the induction step, costs from children will be higher than the ones from their descendants and prevail at Step

1, and therefore the result of Step 2 is the sum of the costs of the children). Therefore, the sum between the local cost and the last valued nogood coming from its children defines the last valued nogood sent by A_{k-1}.
□

Theorem 9 *ADOPT-ing returns an optimal solution.*

Proof. We prove by induction on an ever increasing suffix of the list of agents that this suffix converges to a solution that is optimal for the union of the sub-problems of the agents in that suffix.

The induction step is immediate for the suffix composed of the agent A_n alone. Assume now that it is true for the suffix starting with A_k. Following the previous two lemmas, one can conclude that at quiescence, A_{k-1} knows exactly the minimal cumulated cost of the problems of its successors for its chosen assignment, and therefore knows that this cumulated cost cannot be better for any of its other values.

Since A_{k-1} has selected the value leading to the best sum of costs (between its own local cost and the costs of all subsequent agents), it follows that the suffix of agents starting with A_{k-1} converged to an optimal solution for the union of their sub-problems.   □

The space complexity is basically the same as for ADOPT. The SRCs do not change the space complexity of the nogoods. The largest space is required by the data structure used for storing potential payloads of future (equivalents of) THRESHOLD messages.

Theorem 10 *The space complexity of an agent in ADOPT-ing is $O(dn^2)$.*

Proof. In an agent, A_i, the space for storing the *outgoing_links*, and the agent view (assignments) is linear in n, having at most one link and one assignment per agent. Six data structures in ADOPT-ing store valued nogoods ($l[1..d]$, $ca[1..d][i+1..n]$, $th[1..i]$, $h[1..d]$, $lr[i+1..n]$, $lastSent[1..i-1]$). Therefore the space complexity is given by the complexity of the largest of them, ca, which stores $O(dn)$ cost assessments that can be sent as threshold nogoods.

Each valued nogood contains a list of up to n assignments and a list of up to n SRCs, its space being linear in n. Therefore the total space requirement for an agent is $O(dn^2)$.   □

The space complexity for using the simulator of ADOPT-ing as a centralized WCSP solver is given by the sum of all the spaces of the n agents, which is $O(dn^3)$. The simulator also maintains the queues of *traveling* messages, which can be compacted such that only the last sent message is stored for each channel (Silaghi, Sam-Haroud, & Faltings, 2000). There are $O(n^2)$ bidirectional channels, each of them requiring at most a valued nogood (for an optimized simulator); therefore their total size is $O(n^3)$, being smaller than the sum of the sizes of the agents.

We expect that one can further optimize the space of a centralized implementation by abandoning the message-passing paradigm of the simulator and by sharing the ca data structures of the agents, directly storing each inferred valued nogood at its final position in the structure ca. Additional improvements in space complexity are possible by simply discarding the ca storage in favor of more compact aggregations of its nogoods (where h and the structure for $f()$ mentioned in Section 3.4 are used alone without ca, integrating incoming nogoods directly in h), with a total space complexity of $O(dn^2)$. However, some nogoods would be lost and may have to be recomputed, and threshold nogoods would no longer be available.

3.7 Optimizing valued nogoods

Both for the versions of ADOPT-ing using DFS trees, as well as for the version that does not use such DFS trees, 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 (Ginsberg, 1993). 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 costs. Nogoods optimized in a similar manner were used in
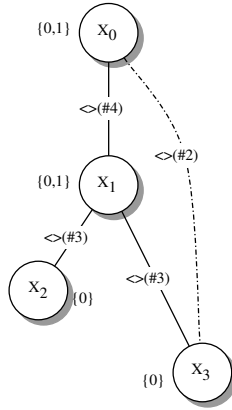
Figure 7: A DCOP with four agents and four inequality constraints. For example, the fact that the cost associated with not satisfying the constraint $x_0 \neq x_1$ is 4 is denoted by the notation (#4).

several previous distributed CSP techniques (Bessiere et al., 2005; Silaghi et al., 2001b). A similar effect is achieved by computing min_resolution(j) with incrementally increasing j and keeping new nogoods only if they have higher cost than previous ones with lower targets.

3.8 Example

Next we detail and contrast the executions of ADOPT-Yos, ADOPT-Aos, and ADOPT-aos illustrating the different types of inferences involved in them. The main description follows the run of ADOPT-Aos while describing differences with ADOPT-Yos and ADOPT-aos when they occur. Take the problem in Figure 7, a trace of which is shown in Figure 8. Identical messages sent simultaneously to several agents are grouped by displaying the list of recipients on the right hand side of the arrow. In our implementation, we decide to maintain a single reference for each agent's secret constraints. In our next description, the notation which refers to the constraints of the agent A_i in a SRC is J_i. In the messages of Figure 8, 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 (i.e., J_i is part of the justification of the valued nogood).

Initialization.   The agents start selecting values for their variables and announce them to interested lower priority agents. There are no constraints between x_3 and x_2. Similarly, there is no constraint between x_0 and x_2; therefore, the first exchanged messages are **ok?** messages sent by A_0 to both successors A_1 and A_3 and which propose the assignment $x_0=0$. Concurrently, A_1 sends **ok?** messages to A_2 and A_3 proposing $x_1=0$. These are messages 1 and 2 in Figure 8. The messages in Figure 8 are grouped by their cycle in the simulator based on rounds (i.e., assuming constant communication latency and no cost for local computations). The simulator with asynchronous cycles can yield different traces function of the random latencies.

Handling data structures for ok? messages.   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, such as: $\{A_0\}$, $\{A_0, A_1\}$, $\{A_0, A_1, A_2\}$. A_1 receives the assignment of x_0 and infers a valued nogood based on its constraint ($x_0 \neq x_1$). It is stored as cost assessment in its structure l, before being integrated in h. $h[1] = l[1] = [\{J_1\}, 4, \langle x_0, 0 \rangle]$. $l[1]$ (and $h[1]$) have cost 0 while $l[0]$ and $h[0]$ have cost 4. Therefore A_1 switches the value of x_1 to 1 and announces it to A_2 and A_3 via message 3. A_1 cannot compute any valued nogood to send to A_0.

1. A_0	**ok?**$\langle x_0, 0 \rangle$	A_1, A_3
2. A_1	**ok?**$\langle x_1, 0 \rangle$	A_2, A_3
3. A_1	**ok?**$\langle x_1, 1 \rangle$	A_2, A_3
4. A_2	**nogood**$[|F, F, T, F|, 3, \langle x_1, 0 \rangle]$	A_1
5. A_3	**nogood**$[|F, F, F, T|, 2, \langle x_0, 0 \rangle]$	A_0, A_2
6. A_3	**nogood**$[|F, F, F, T|, 5, \langle x_0, 0 \rangle \langle x_1, 0 \rangle]$	A_1, A_2
7. A_0	**ok?**$\langle x_0, 1 \rangle$	A_1, A_3
8. A_2	**nogood**$[|F, F, F, T|, 2, \langle x_0, 0 \rangle]$	A_0
9. A_2	**nogood**$[|F, F, T, T|, 5, \langle x_0, 0 \rangle \langle x_1, 0 \rangle]$	A_1
10. A_2	**add-link**$\langle x_0, 0 \rangle$	A_0
11. A_2	**nogood**$[|F, F, T, T|, 8, \langle x_0, 0 \rangle \langle x_1, 0 \rangle]$	A_1
12. A_3	**nogood**$[|F, F, F, T|, 2, \langle x_0, 0 \rangle]$	A_2
13. A_0	**ok?**$\langle x_0, 1 \rangle$	A_2
14. A_1	**ok?**$\langle x_1, 0 \rangle$	A_2, A_3
15. A_2	**nogood**$[|F, F, F, T|, 2, \langle x_0, 0 \rangle]$	A_0, A_1
16. A_2	**nogood**$[|F, F, T, T|, 5, \langle x_0, 0 \rangle \langle x_1, 0 \rangle]$	A_1
17. A_2	**nogood**$[|F, F, T, F|, 3, \langle x_1, 0 \rangle]$	A_1
18. A_3	**nogood**$[|F, F, F, T|, 3, \langle x_1, 0 \rangle]$	A_1, A_2
19. A_1	**nogood**$[|F, T, F, T|, 3, \langle x_0, 1 \rangle]$	A_0
20. A_1	**nogood**$[|F, T, T, T|, 4, \langle x_0, 1 \rangle]$	A_0
21. A_1	**ok?**$\langle x_1, 1 \rangle$	A_2, A_3
22. A_2	**nogood**$[|F, F, T, T|, 6, \langle x_1, 0 \rangle]$	A_1
23. A_0	**ok?**$\langle x_0, 0 \rangle$	A_1
24. A_0	**ok?**$\langle x_0, 0 \rangle$**threshold** $[|F, F, F, T|, 2, \langle x_0, 0 \rangle]$	A_2, A_3
25. A_3	**nogood**$[|F, F, F, T|, 2, \langle x_0, 0 \rangle]$	A_0, A_2
26. A_2	**nogood**$[|F, F, F, T|, 2, \langle x_0, 0 \rangle]$	A_0, A_1
27. A_1	**nogood**$[|F, T, F, T|, 2, \langle x_0, 0 \rangle]$	A_0

Figure 8: Trace of ADOPT-Aos on the problem in Figure 7. Horizontal lines separate groups of messages with the same logic clock (i.e., messages that are part of the same round in a simulator based on rounds).

After the agent A_2 gets message 2, it computes in $l[0]$ a valued nogood with cost 3 (conflict with $x_1 \neq x_2$). This valued nogood is copied in $h[0]$ and $lastSent[1]$ before being sent to A_1 via message 4. No nogood can be computed for A_0.

Remark 11 (ADOPT-Aos vs ADOPT-Yos) *In ADOPT-Yos this message would also include the current list of known $ancestors$ which here contains only A_1.*

When A_3 gets message 1, it tries to separately infer nogoods for the prefixes of the set of agents: $\{A_0\}$, $\{A_0, A_1\}$, and $\{A_0, A_1, A_2\}$. For the set $\{A_0\}$ it detects a conflict with its constraint $x_3 \neq x_0$ from which it infers a valued nogood stored as cost assessment in $l[0]$, copied to $h[0]$ and $lastSent[0]$ before being sent to A_0 via message 5. For the set $\{A_0, A_1\}$, the computed nogood is identical with the one for A_0 and its target does not coincide with A_1, the last agent of the corresponding set. Therefore ADOPT-Aos sends no message to A_1. Message 5 is also sent to A_2 according to the rule that an agent always attempts to send nogoods to its predecessor, to ensure optimality. Its nogood is stored by A_3 in $lastSent[2]$.

Remark 12 (ADOPT-Aos vs ADOPT-aos) *Note that message 5 is not sent to A_1, as would be the case with the version ADOPT-aos.*

Remark 13 (ADOPT-Aos vs ADOPT-Yos) *With ADOPT-Yos, message 5 would not be sent to A_2, since the current parent of A_3 would be A_0.*

After receiving the assignment in message 2, A_3 detects a new conflict with its constraint $x_1 \neq x_3$. From its two constraints A_3 infers a new valued nogood, stored in its $l[0]$ and $h[0]$, and sent to A_1 and A_2 via message 6. Note that a nogood is not sent to A_0 as the nogood to be sent is identical to the last nogood sent to that destination (as recorded in $lastSent[0]$).

Remark 14 (ADOPT-Aos vs ADOPT-Yos) *With ADOPT-Yos, A_1 would become the parent of A_3 at this stage due to the non-zero cost of the constraint between x_3 and x_1. A_3's known ancestors would become A_0, A_1, and this list would be sent with all nogood messages.*

Handling data structures for nogoods. As a result of getting the nogood in message 5 from A_3, the agent A_0 stores that nogood in $lr[3]$, copies it to $ca[0][3]$ (which was empty), and copies it further in $h[0]$. Since now the cost of $h[0]$ is 2, A_0 decides to switch to its next value, 1. This assignment is announced via message 7.

After receiving message 5, A_2 registers that nogood in its $lr[3]$, $ca[0][3]$ and $h[0]$. Computing a nogood for A_0, the nogood of message 5 is stored in $lastSent[0]$ and sent to A_0 via message 8. Agent A_2 also computes a nogood for destination A_1, where it can also use the local constraint with x_1 which yields for $l[0]$ a nogood with cost 3. Combining $l[0]$ with $ca[0][3]$ by sum-inference, A_2 infers a nogood, which it stores in $h[0]$ and $lastSent[0]$ before sending it to A_1 via message 9. A_2 detects a new variable in the nogood in message 6, and sends an **add-link** message to A_0 asking to be notified of changes to the assignment $x_0 = 0$. The nogood in message 6 replaces the one stored in $lr[3]$. Since the new nogood cannot be combined by sum-inference with the old nogood in $ca[0][3]$ but has a higher cost, it also replaces that cost assessment and leads to the computation by sum-inference of message 11 to be sent to A_1.

In the following we skip the details of changes to data structures that are similar to steps that have already been presented. When the new assignment of x_1 in message 3 is received at agent A_3, the old nogoods based on x_1 are discarded from its $l[0]$. To send a nogood to A_0, a new $l[0]$ is computed based solely on the constraint $x_0 \neq x_3$. Nogoods computed for the other prefixes of agents do not differ from this one since the constraint with x_1 is satisfied. This nogood with cost 2 is sent via message 12 to the agent A_2. Note that the nogood does not need to be sent to A_0 because it is not different from the one just sent earlier (via message 5) and recorded in $lastSent[0]$. After getting message 7, A_1 deletes its nogoods in $l[0]$ and $ca[0][3]$, infers a new valued nogood in $l[1]$ with cost 4, and switches to the value 0 (announced via message 14).

Use of lr data structure. Let us assume that A_2 receives message 12 before message 3, which is possible and allows us to illustrate better the usage of the lr structure. On receiving message 12, agent A_2 stores it in $lr[0]$. However, A_2 does not propagate it further to $ca[0][3]$ since the current cost assessment had a higher cost and cannot be combined by sum-inference with the new one (sharing the reference to the constraints of A_3). When A_2 receives message 3, it deletes its $ca[0][3]$ and $l[0]$, which are based on the older value of x_1, and uses $lr[3]$. After copying $lr[3]$ through its $ca[0][3]$ and $h[0]$ data structures where all other nogoods were empty, it passes it further to A_0 and to A_1 via message 15 (storing it at $lastSent[0]$ and $lastSent[1]$). Since A_0's value for x_0 is different from the one in the **add-link** message 10, A_0 answers to A_2 with the message 13.

Now A_2 receives message 14 and computes a new local nogood $l[0]$ with cost 3 that is combined by sum-inference with the nogood received in message 12 to generate the nogood in message 16. No change appears in the nogood computed specially for the target A_0. However, after A_2 also receives message 13 it discards the nogood received via message 12 (which was based on an outdated assignment) and infers its $h[0]$ solely based on $l[0]$. The result is sent to A_1 with message 17. After receiving the two assignments in messages 13 and 14 (in this order) the agent A_3 infers from its constraint $x_3 \neq x_1$ a valued nogood sent to A_1 and A_2 via message 18.

Min-resolution. Now our example encounters the first nontrivial min-resolution. When agent A_1 receives message 18, it stores that nogood in $lr[3]$ and $ca[0][3]$. No other nogood is stored in ca at this point (the nogood received with message 15 in $ca[0][1]$ has already been invalidated by the new assignment in message 7). The only other nogood held by A_1 at this moment is the one in $l[1] = [\{J_1\}, 4, \langle x_0, 1 \rangle]$, which is due to its constraint with x_0. $l[1]$ is copied in $h[1]$ while $ca[0][3]$ is copied in $h[0]$. The two are combined via min-resolution to generate the nogood in message 19 (also stored in $lastSent[0]$).

$$min\_resolution([\{J_3\}, 3, \langle x_1, 0 \rangle], [\{J_1\}, 4, \langle x_0, 1 \rangle]) \rightarrow [\{J_1, J_3\}, 3, \langle x_0, 1 \rangle \langle x_1, 0 \rangle]$$

Message 16 is discarded at its destination because its assignment for x_0 is no longer valid. On the arrival of message 17 (which is concurrent with messages 16 and 18) its nogood is stored in $lr[2]$ and $ca[0][2]$. Now,

when computing the updated nogood to be sent to A_0, $h[0]$ is computed by sum-inference on $ca[0][2]$ and $ca[0][3]$ obtaining $[\{J_2, J_3\}, 6, \langle x_1, 0 \rangle]$.

$$sum\_inference([\{J_2\}, 3, \langle x_1, 0 \rangle], [\{J_3\}, 3, \langle x_1, 0 \rangle]) \rightarrow [\{J_2, J_3\}, 6, \langle x_1, 0 \rangle]$$

The obtained valued nogood has a higher cost than the one for $h[1]$, causing the agent to switch the assignment of x_1 to 1 (announced via message 21). When min-resolution is applied on the two nogoods in $h[0]$ and $h[1]$, the obtained nogood is sent to A_0 via message 20.

$$min\_resolution([\{J_2, J_3\}, 6, \langle x_1, 0 \rangle], [\{J_1\}, 4, \langle x_0, 1 \rangle])$$
$$\rightarrow [\{J_1, J_2, J_3\}, 3, \langle x_0, 1 \rangle \langle x_1, 0 \rangle]$$

Convergence. Agent A_2 also receives message 18, storing the nogood in $lr[3]$ and in $ca[0][3]$. Its constraint $x_2 \neq x_1$ generates a nogood with cost 3 in $l[0]$, which combined by sum-inference with the nogood in ca, leads to a nogood with total cost 6, visible in message 22.

Agent A_0 receives message 19 and registers the nogood in $lr[1]$, $ca[1][1]$, and $h[1]$. The cost assessment obtained in $h[1]$ has a cost higher than the one in $h[0]$, determining the switch of the assignment of x_0 to 0 (announced via messages 23 and 24). Message 24 also transports a threshold nogood obtained from $ca[0][2]$ and $ca[0][3]$ (received via messages 15 and 5). The agent A_3 evaluates its constraint $x_0 \neq x_3$ inferring a valued nogood in $l[0]$, which propagates through its $h[0]$, $lastSent[2]$, $lastSent[0]$ to messages 25. Similarly A_2 propagates this nogood to A_1, which propagates it further through its data structures and eventually delivers it to A_0 via message 27. Messages 25, 26 and 27 basically confirm the already known threshold nogoods. Further research may make it possible to avoid them[14].

We have modeled solved this example with our implementation for ADOPT-Aos and ADOPT-aos with rounds. ADOPT-Aos used one more cycle but 3 less messages than ADOPT-aos.

3.9 Theoretical comparison between ADOPT-ing and ADOPT/ABT

ADOPT-ing vs ADOPT The difference starts with adding SRCs for justification, explicitly bundling cost-related data into valued nogoods such that associated inferences can be performed formally, and enabling the destination of the nogood (cost) messages to include other agents besides the parent. Internal data management is also different:

1. The DFS tree can be dynamically detected (ADOPT-Y__). It can be based only on already used constraints.

2. ADOPT did not have **add-link** messages.

3. In ADOPT (as a result of not using SRCs and not having our rules on the order for combination of nogoods) messages could be sent only to the parent rather than to any ancestor.

4. ADOPT could not use explicit max-inference (because it did not maintain SRCs).

5. ADOPT did not maintain data structures like lr and $lastSent$ to avoid resending the same message several times and easy the network load.

6. ADOPT did not provide guidelines for using any additional storage other than the minimal ones (ADOPT did not specify/have an equivalent of Lemma 6 with rules for using cost information).

7. New assignments arriving first via nogoods can be detected as such in ADOPT-ing (as in (Silaghi & Faltings, 2004)) while in ADOPT they had to be considered old.

---

14. E.g, by a mechanism for storing threshold nogoods in the $lastSent$ of the recipient and in the lr of the sender, resending the $lastSent$ when the threshold nogood does not apply.

ADOPT-ing vs ABT   Unlike ABT:

- An ADOPT-ing agent may send possibly irrelevant messages to a given predecessor (its parent in the current DFS tree). It does this to guarantee optimality given the non-idempotent aggregation operation of DCOPs.

- The **nogood** messages have an associated cost and justification (SRCs). These are used to find the assignments with the least conflicts in case of an unsatisfiable problem.

3.10 Possible Extensions

We addressed ADOPT-ing as an asynchronous version of A*, more exactly a version of iterative deepening A*, where the heuristic is computed by recursively using ADOPT-ing itself, and where the composition of the results of recursive ADOPT-ing is based on backtracking.

A proposed extension to this work consists of composing the recursive asynchronous heuristic estimator by using consistency maintenance. This can be done with the introduction of *valued consistency nogoods*. Details and variations are described in (Silaghi, 2002, 2003b; Silaghi et al., 2004; Gershman, Meisels, & Zivan, 2006, 2007; Sultanik et al., 2006). The control of the space requirements for such extensions may be based on the use of consistency nogoods to simulate the distributed weighted arc consistency in (Silaghi et al., 2004), while the maintenance of this control of space in asynchronous search may be similar to the one for distributed CSPs described in (Silaghi & Faltings, 2004). Another possible extension is by further generalizing the nogoods such that each variable can be assigned a set of values. This type of aggregation was shown in (Silaghi & Faltings, 2004) to improve search, and the extension is detailed in (Silaghi, 2002).

In our implementation we concentrated on minimizing the logic time of the computation, evaluated as the number of rounds on a simulator. The optimization of local processing (which is polynomial in the number of variables) is not at the center of attention at this stage. Local computations can be optimized, for example, by reusing values of structures l and h computed at min-resolution for a given target agent in obtaining values of these structures at the min-resolution for messages sent to lower priority target agents. Further work can determine whether improvements could be made by storing separately the nogoods of h for each target k. The size of messages in ADOPT-Yos could be slightly reduced by appending a given content of the $ancestors$ list only once to each target. ADOPT-Yos is better than ADOPT-Aos in terms of simulated time. Agents in ADOPT-Yos could insert from the beginning all their neighboring predecessors in their ancestors list, obtaining from the first n rounds the DFS tree of ADOPT-Dos, thereby replicating the efficiency of ADOPT-Dos.

Other extensions seem possible by integrating additive branch and bound searches on DFS sub-trees, as proposed by (Chechetka & Sycara, 2006; Yeoh, Koenig, & Felner, 2007). This can be added to ADOPT-ing by maintaining solution-based nogoods as suggested in (Silaghi, 2002). It remains to be seen if the quality of solutions with a certain value can be predicted with the technique in (Petcu & Faltings, 2006b). Further improvements are possible by running ADOPT-ing in parallel for several orderings of the agents (Ringwelski & Hamadi, 2005; Benisch & Sadeh, 2006).

ADOPT-ing can be seen as an extension of ABT. The extension of ABT called ABTR (Silaghi, Sam-Haroud, & Faltings, 2001a; Silaghi, 2006) proposes a way to extend ABT-based algorithms to allow for dynamic ordering of the agents (Armstrong & Durfee, 1997). Work in the area consistent with this approach, but mainly favoring static ordering, appears in (Liu & Sycara, 1995; Chechetka & Sycara, 2005). Finding good heuristics was shown to be a difficult problem (Silaghi et al., 2001b; Zivan & Meisels, 2005) and here one will need to take into account the importance of the existence of a short DFS tree compatible to the current ordering.

4. Experiments

We implemented several versions of ADOPT-ing. Some versions use valued nogoods while other versions use valued global nogoods. Some versions maintain an optional DFS tree precomputed on the constraint

graph. Some versions exploit more opportunities to send optional **nogood** messages[15] than others. In the version ADOPT-pos, valued nogoods are sent only to the parent of the current agent in a maintained DFS tree. In ADOPT-dos, each agent A_i tries to compute a valued nogood after each change, for each of its ancestors A_j in the DFS tree, and sends this nogood to A_j if it is new and has a non-zero cost. ADOPT-aos can be seen as a version of ADOPT-dos where the DFS tree is reduced to the linear list of agents (each having the predecessor as parent). ADOPT-Aos is a version of ADOPT-aos where an optional **nogood** message is sent only if the destination of the message is the same as the target of the nogood in the payload. The same holds for the relation between ADOPT-Dos and ADOPT-dos. The version ADOPT-Yos is a hybrid between ADOPT-Aos and ADOPT-Dos where initially agents can only contact neighboring agents and the DFS tree is dynamically discovered during the search (more similar to ABT). The version of ADOPT-Yos using method (b) in Step 1 of Remark 8 is denoted ADOPT-*Yos.b*. The version of ADOPT-Yos where the computation of h for a destination agent reuses the results computed for its ancestors is denoted ADOPT-*Yos.a.opt*.

ADOPT-pon, ADOPT-don, and ADOPT-aon are variations of ADOPT-pos, ADOPT-dos, and ADOPT-aos where valued global nogoods are used instead of valued nogoods. Note that ADOPT-pon is our implementation of the original ADOPT. For experiments with random message latencies and for outputs not provided by the original implementation of ADOPT (e.g., ENCCCs), we had to provide the results of our implementation (ADOPT-pon). While ADOPT-pon and the original implementation of ADOPT performed similarly in general, ADOPT-pon solved in a few hours the instances for which the original ADOPT implementation was interrupted after some weeks, confirming that some differences in details may exist. Functional differences between ADOPT-pon and the original implementation of ADOPT may lie only in petty details not described in (Modi et al., 2005). To specify that in a certain experiment we used our implementation rather than the original implementation of ADOPT, we will denote it with *ADOPT (-p)*.

We implemented a version of ADOPT (using our implementation of ADOPT-pon with threshold nogoods) that uses a chain of agents like ADOPT-aos, rather than the DFS tree. This version is denoted *ADOPT.chain*. We also experimented with versions of ADOPT-aon, ADOPT-don, ADOPT-aos and ADOPT-Yos where threshold valued nogoods are not used. This helped to isolate and evaluate the importance of threshold valued nogoods in ADOPT-ing.

The algorithms are compared on the same problems that are used to report the performance of ADOPT in (Modi et al., 2005). 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 20% and with density 30%. A smaller set of problems with density 40% is also available. The density of a (binary) constraint problem's graph with n variables is defined by the ratio between the number of binary constraints and $\frac{n(n-1)}{2}$. Results are averaged on the 25 problems with the same parameters.

We believe that the size of problems in this set is sufficiently large, given that the average simulated time (expected time of a real solver) for the instances with 40 agents at density 30% is between 3 hours and 27 hours, (and up to 10 days at 25 agents and density 40%), longer than what users are expected to wait for a solution.

Our simulator allows for defining the latency of each message, and we performed two sets of tests. In the first set of experiments, we followed a common practice of using the same latency for each message (to be referred as *synchronous cycles*). In a second set of experiments[16], we also performed experiments with random message latencies (to be referred as *asynchronous cycles*). The random latencies were generated in the range of common values for Internet communications via optical fiber between Israel and the United States which is between 150ms and 250ms (Neystadt & Har'El, 1997). To reproduce our results for the second set of tests, one has to seed the standard C 'random()' function with the value 10000 and generate

---

15. Messages to predecessors other than the previous agent (or parent agent for versions with DFS trees).
16. At the request of an anonymous reviewer.

each latency as carried out in (Neystadt & Har'El, 1997):

$$latency = 150 + \frac{random() * 100.0}{LONG\_MAX}(msec).$$

FIFO channels are ensured in the second set of tests by setting the delivery time of each message to the maximum between the value obtained using the latency yielded by the aforementioned computation and the delivery time of the last message sent on that particular communication channel. Messages with the same value for the delivery time are handed to the destination agent in a FIFO manner through a queue.

In graphs, an algorithm $ADOPT\text{-}\mathcal{DON}$ is typically shortened to $\mathcal{DON}$. When confusions are possible, the name of the algorithm is followed by the notation *async* to denote asynchronous cycles and *rounds* to denote synchronous cycles.
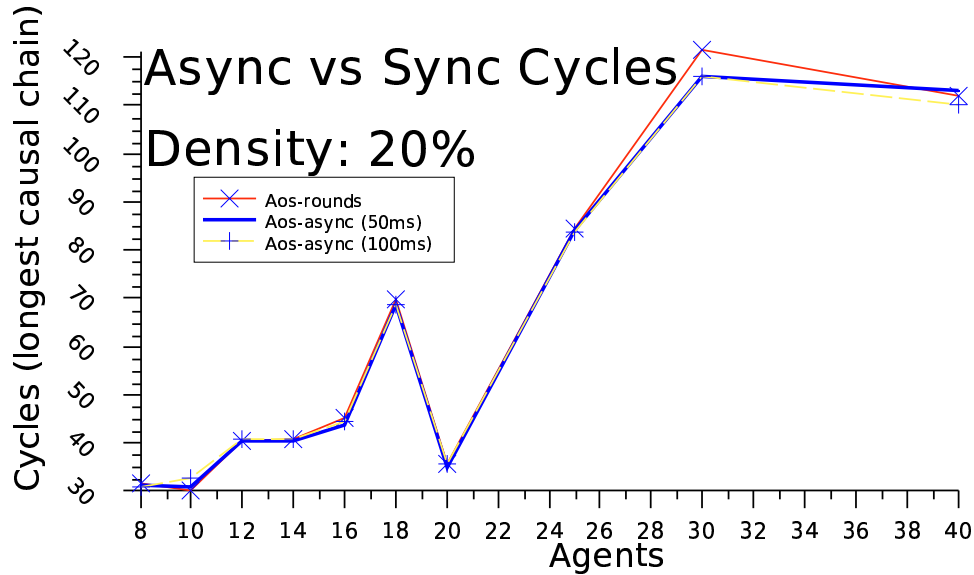
Figure 9: Asynchronous vs. synchronous cycles for problems with density 20%.

We compare the number of synchronous cycles with the number of asynchronous cycles obtained on our two simulators of ADOPT-ing. For this experiment we also evaluated ADOPT-Aos using a smaller distribution of values for the message latency (150ms to 200ms) to see its influence on results. The results show an impact on the different versions of ADOPT-ing, ranging from 1% to 5%. A similar impact may be explained by the common nature of the different versions. Several cases are shown in Figures 9 and 10. For ADOPT-aos the version based on synchronous cycles is 3% faster than the version with 100ms of random variation in latency. The performance of the implementation of ADOPT-Aos based on synchronous cycles is between the performance of its implementation with 50ms random variation of message latencies and the performance of its implementation with 100ms random variation of message latencies. It differs from them by less than 1%.

Particular care has to be taken in the evaluation of distributed algorithms because one needs to take into account two tightly related factors of a very different nature: local computations (constraint checks) and message latencies. The classical solution to this problem is to compute several measures at different possible ratios between the latency of a message and the time associated with a constraint check, yielding the behavior of the techniques in different scenarios. Here we do this analysis for the ratios: 10^6:1, 10^5:1, 10^4:1, 10^3:1, 10^2:1, 10:1, 1:10. The actual ratio for a given application scenario depends on the speed of the used CPUs, but we want to get a hint about the order of magnitude of this ratio, *the operating point (OP)*, in the targeted application scenario of remote Internet communications. We therefore compute the average amount of time
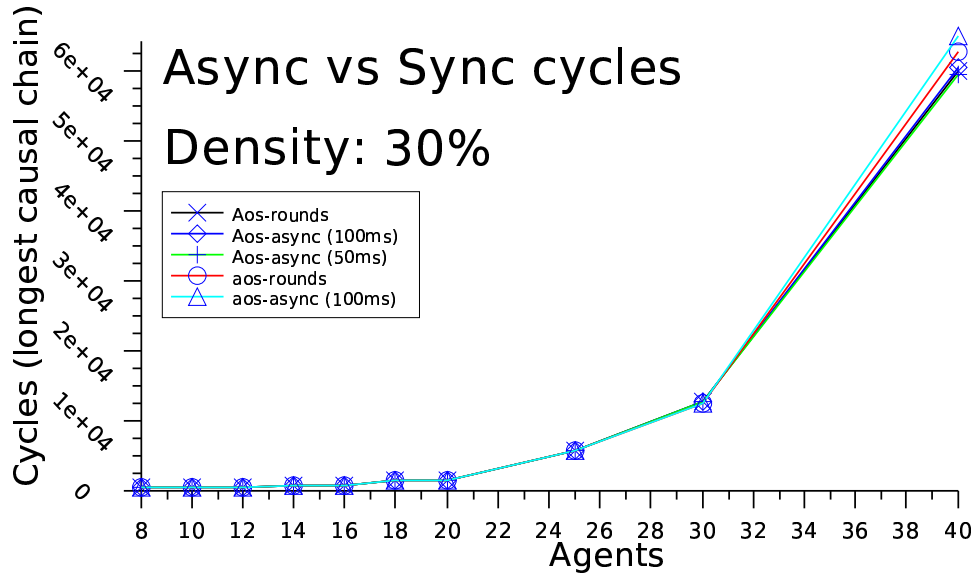
Figure 10: Asynchronous vs. synchronous cycles for problems with density 30%.

Agents	checks/latency				
	ADOPT (-p)	Aos	aos	dos	Yos
8	62518	77417	78076	59314	65032
10	50104	67731	69560	48067	56087
12	46666	61784	61899	43366	50637
14	42088	55634	55209	38715	44616
16	38042	50425	47544	34281	39949
18	29673	41209	36854	26142	32062
20	25104	35455	31078	21064	27012
25	17437	24809	21998	14510	19225
30	12519	17911	14521	10138	13986
40	7041	10654	8461	5899	8555

Table 1: The operating point (checks per message latency) on problems at density 30%.

spent by our simulator for a constraint check. This is computed by dividing the total running time of the simulator by the total number of constraint checks during the process. The result for different problem sizes is shown in Figure 1. Our operation point is shown to vary between 5000:1 and 80000:1, at average message latency 200ms.[17]

Figure 11 shows equivalent non-concurrent constraint checks (ENCCCs) (Chechetka & Sycara, 2006; Silaghi & Faltings, 2004; Meisels, Kaplansky, Razgon, & Zivan, 2002; Silaghi, Sam-Haroud, & Faltings, 2000; Yokoo et al., 1992) for the problems at size 40 and density 30%. There the overall cost is studied for the case where the latency of messages decreases to lower ratios with respect to the cost for constraint checks (predicting behavior for local area networks and MIMD parallel computer systems with efficient message passing). With equivalent non-concurrent constraint checks, the intersection of the graph with the vertical axis yields the number of non-concurrent constraint checks (NCCCs).

17. Earlier experiments with a slower CPU yielded a ratio around 1000:1.
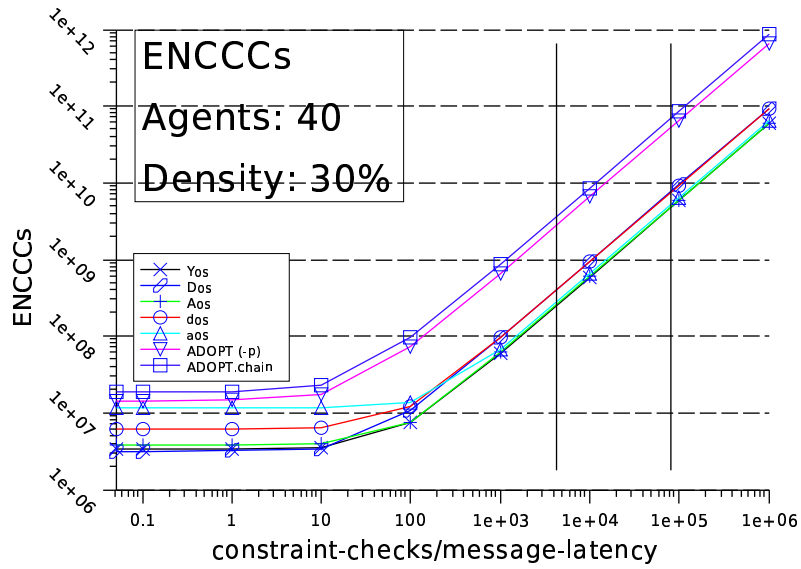
Figure 11: Equivalent non-concurrent constraint checks (ENCCCs). The two vertical segments delimit the operation point area, between 5000 and 80000 checks per message-latency.
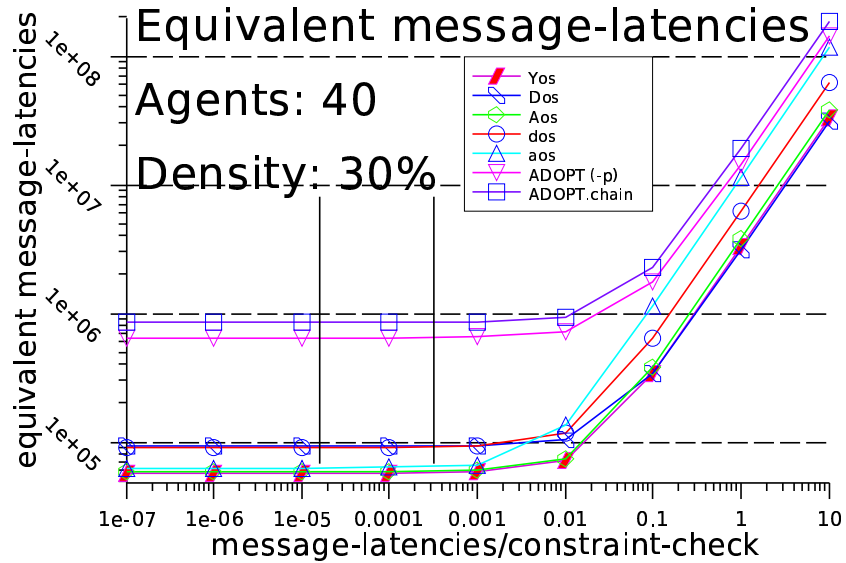

Figure 12: Equivalent latencies. The two vertical segments delimit the operation point area, between 5000 and 80000 checks per message-latency.

We obtain what we consider a slightly better visibility of the behavior with another kind of graph, measuring equivalent latencies rather than equivalent non-concurrent constraint checks. This is obtained by dividing each point in the ENCCCs graph by the number of checks/latency in that point. Such equivalent latencies are shown in Figure 12. We note that the closest analyzed latency/check ratio to the ones found experimentally is 10^4:1. Once the OP is fixed, the two metrics yield the same graph, differing just by a scaling factor
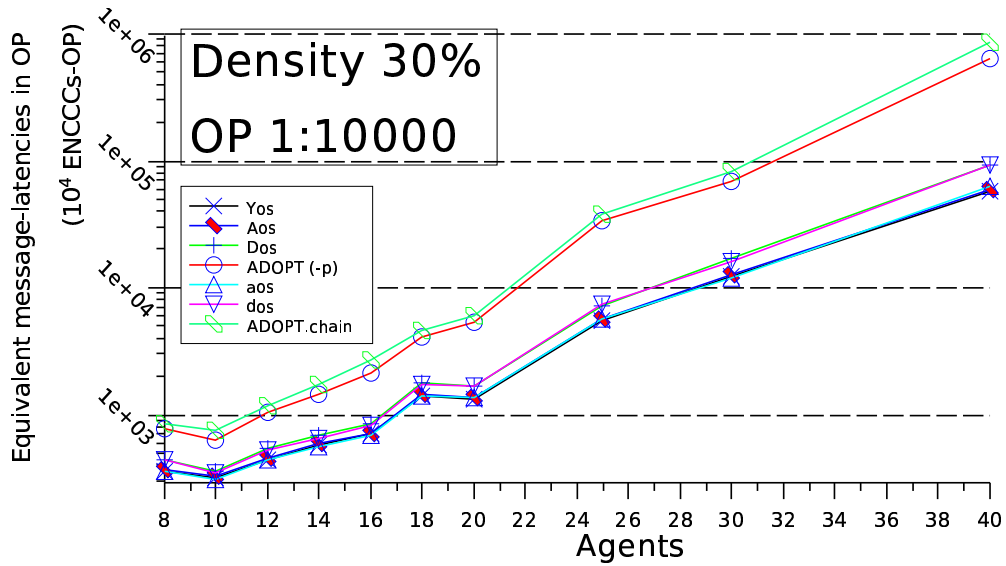
Figure 13: Equivalent latencies and ENCCCs in the operation point 10^4:1 checks per message-latency.

(Figure 13). Figure 12 shows that the behavior in this OP is practically identical to the one obtained from the number of asynchronous cycles (equivalent to a ratio $1:\infty$ given by the intersection of the graph in Figure 12 with the vertical axis). Therefore in the following we look closer at the number of cycles.
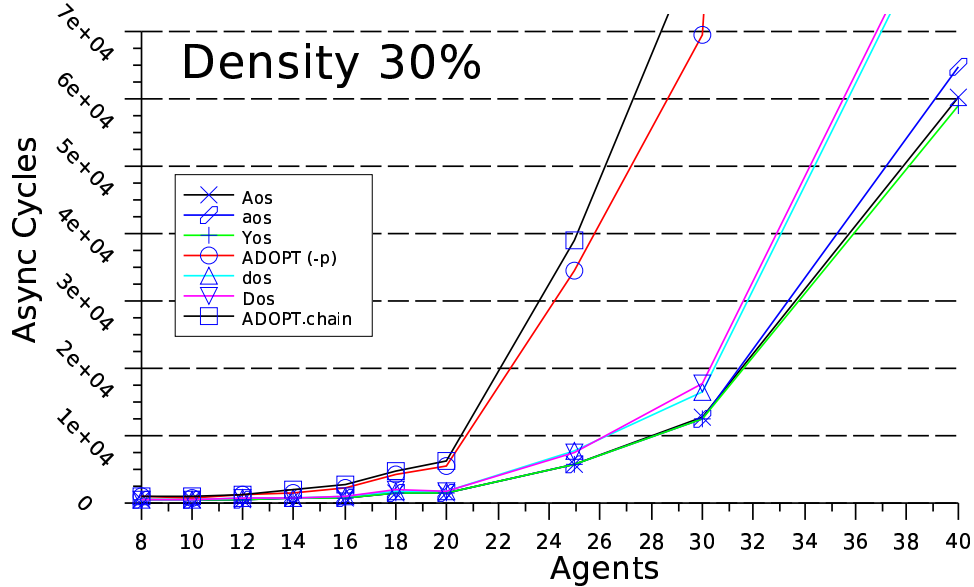
Figure 14: Asynchronous cycles for problems with density 30%.

The length of the longest causal (sequential) chain of messages of each solver (the number of asynchronous cycles), averaged on problems with density 30%, is given in Figure 14. Results for problems with density 20% are given in Figure 15. Results for density 40% are shown in Figure 16. We can note that version ADOPT-Yos of ADOPT-ing brought an improvement of approximately 10 times on problems with
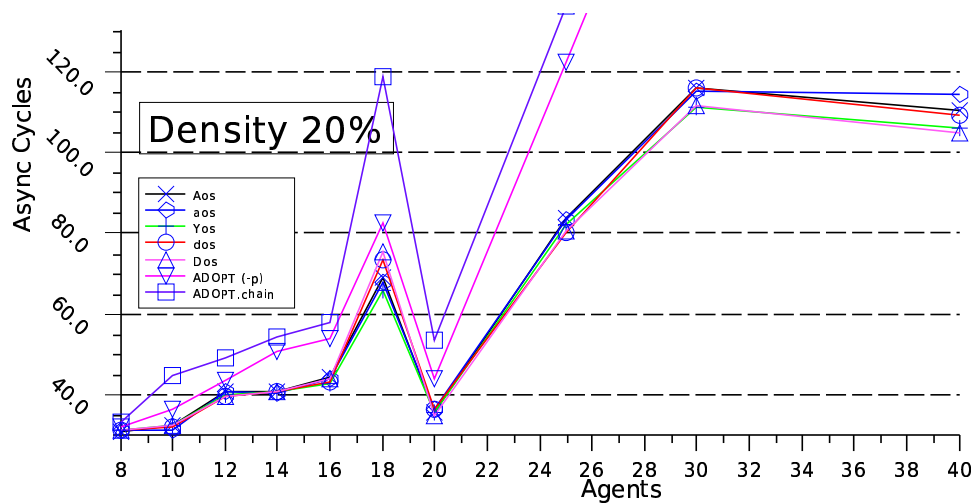
Figure 15: Asynchronous cycles for problems with density 20%.
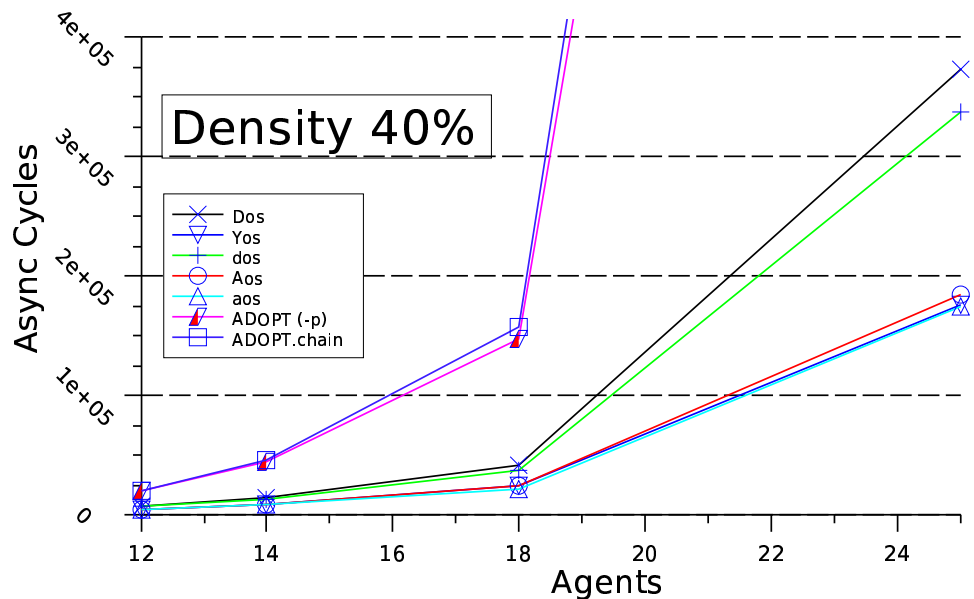
Figure 16: Asynchronous cycles for problems with density 40%.

40 agents and density 30%, and of approximately 12 times on problems with 25 agents and density 40%. The improvement at density 20% is 2 times when compared to ADOPT (-p).[18] Therefore, sending nogoods only to the parent node is significantly worse (in number of cycles), than sending nogoods to several ancestors. With respect to the number of cycles, the use of SRCs with nogood contexts practically replaces the need to maintain the DFS tree since ADOPT-aos and ADOPT-Aos are comparable in efficiency to ADOPT-dos and ADOPT-Dos. New versions of ADOPT-ing are up to 14 times faster than ADOPT.chain, proving that

---

18. At density 20%, with synchronous rounds, the original implementation of ADOPT performs 3.5 times worse than ADOPT (-p), i.e., 7 times worse than ADOPT-Yos. This may be explained by some inefficient detail in the original implementation of ADOPT, since the deviation from ADOPT (-p) does not appear at other densities.
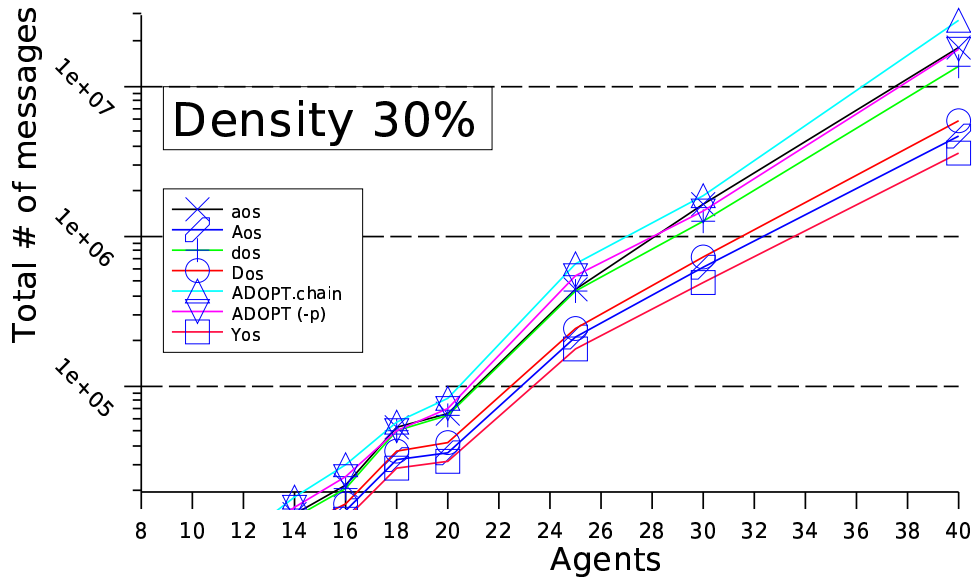
Figure 17: Total number of messages at density 30% (log scale).

Agents	16	18	20	25	30	40
ADOPT-aos	690.24	1420.76	1392.48	5687.28	12254.68	64518.6
no threshold	843.2	1787.16	1801.88	7608.44	17507.16	101956.4
ADOPT-Yos.b	701.72	1438	1345.56	5540.84	12394	59114.36
no threshold	872.76	1781.96	1708.72	7391.28	17531.36	92745.44

Table 2: Impact of threshold valued nogoods on the longest causal chain of messages (asynchronous cycles) for versions of ADOPT-ing, averaged on problems with density 30%.

ADOPT-ing is not a simple application of ADOPT to a chain of agents, but that justified valued nogoods literally succeed in dynamically discovering the DFS tree.

Versions using DFS trees require fewer parallel/total messages, being more network friendly, as seen in Figure 17. Figure 17 shows that refraining from sending too many optional **nogood** messages, as done in ADOPT-Aos, ADOPT-Yos and ADOPT-Dos, is 4 times better at density 30% than ADOPT (-p) in terms of total number of messages, while (as shown by previous graphs) maintaining the efficiency in cycles comparable to ADOPT-aos and ADOPT-dos. At density 40% ADOPT-Yos is 6 times better than ADOPT (-p) in terms of total number of messages. ADOPT-Yos is the most efficient algorithm in terms of total number of messages, being 30% better at density 30% than the second best algorithm, ADOPT-Aos. At density 40% it is 12% better than ADOPT-Aos.

We do not show run-time comparisons with the original implementation of ADOPT since our versions of ADOPT are implemented in C++, while the original ADOPT is in Java (which obviously leads to all our versions being an irrelevant order of magnitude faster). However, we provide run-time comparisons with our implementation of ADOPT, ADOPT (-p). A comparison between the time required by versions of ADOPT-ing on a simulator is shown in Figure 18 for asynchronous cycles. It reveals the computational load of the agents which, as expected, is related to the total number of exchanged messages.

A separate set of experiments was run for isolating and evaluating the contribution of threshold valued nogoods. Table 2 shows that the use of threshold nogoods almost halves the computation time. Another
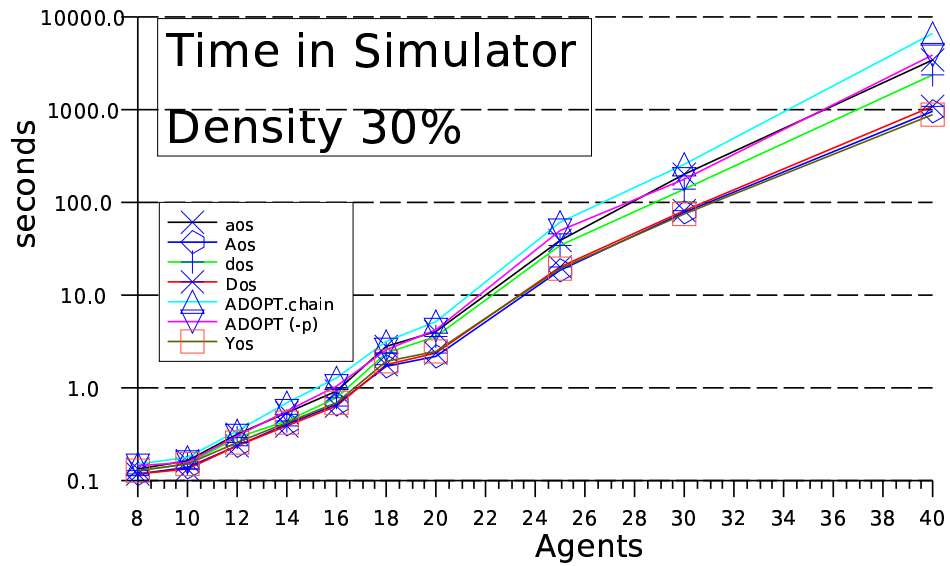
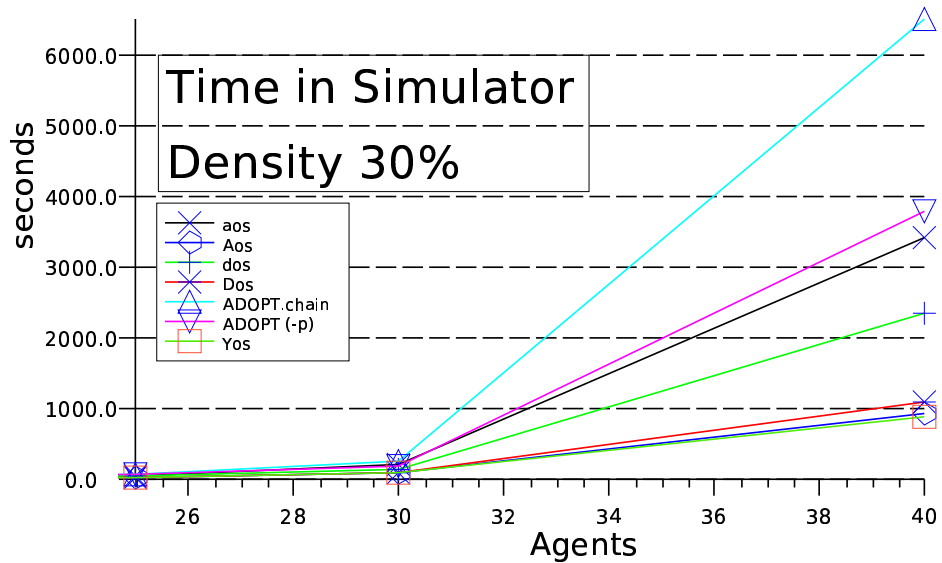Figure 18: Actual time in seconds using our simulator as solver of centralized WCSPs (log scale).

Figure 19: Actual time in seconds using our simulator as solver of centralized WCSPs.

Agents	16	18	20	25	30	40
DFS compatible	708.8	1429.48	1357.07	5579.56	$12.4*10^3$	$60*10^3$
random order	4807.44	$15.6*10^3$	$33*10^3$	$219*10^3$	$708*10^3$	—

Table 3: Impact of choice of order according to a DFS tree on the longest causal chain of messages (asynchronous cycles) for ADOPT-Yos, averaged on problems with density 30%.

experiment, whose results are shown in Table 3, is meant to evaluate the impact of the guarantees that the ordering on agents is compatible with a short DFS tree. We evaluate this by comparing ADOPT-Yos with an ordering that is compatible with the DFS tree built by ADOPT, versus a random ordering. At 30 agents it was found to be 60 times more efficient to ensure that a DFS tree exists rather than to use 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.
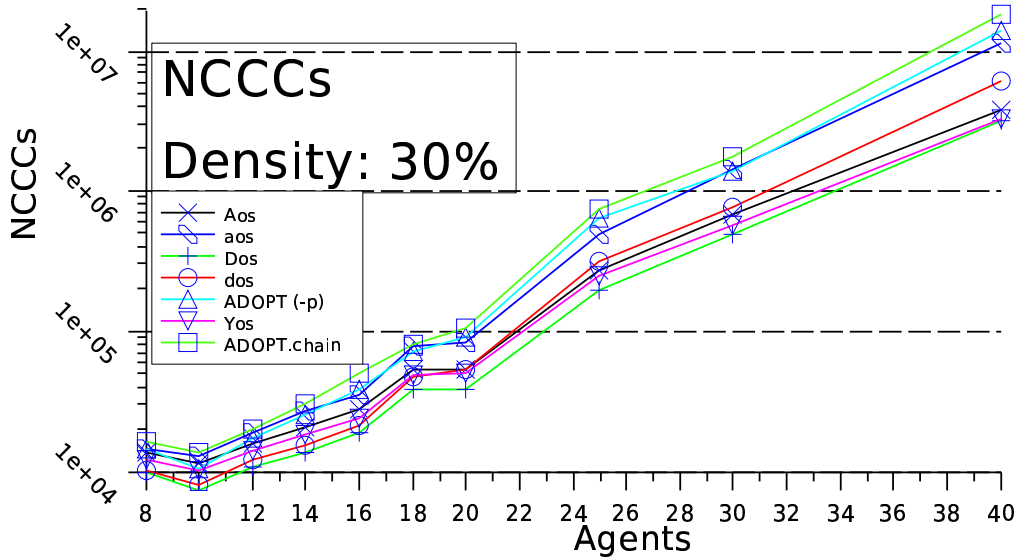
Figure 20: Asynchronous non-concurrent constraint checks (NCCCs).

The number of non-concurrent constraint checks when messages are considered instantaneous (NC-CCs (Meisels et al., 2002)) is relevant for showing the expected performance of the algorithms on MIMD parallel computer systems. We show NCCCs for our algorithms in Figure 20, (they can also be extracted as the intersection points of ENCCCs curves with the coordinate axis in Figure 11). The simulated time of the computations, where the random latencies of the messages are accumulated along the longest causal chain, is shown in Figure 21. The time taken for the local computation handling/generating each message (Figure 18) is hundreds of times smaller than the latency of the associated messages, falling close to the numerical precision of this accounting (Figure 22). The smallest ratio (for ADOPT-aos) is around 80.

We compute the maximum and minimum number of asynchronous cycles of the simulations for each problem size, and the obtained graph is shown in Figure 23.

We have implemented versions that do not use threshold nogoods for ADOPT-aon and ADOPT-don (variants of ADOPT-ing based on *valued global nogoods* instead of the (Dago & Verfaillie, 1996)'s valued nogoods). Those versions were compared with the corresponding versions of ADOPT-aos and ADOPT-dos, evaluated using synchronous cycles, and the results on problems with density 30% are reported in Figure 24. The versions using valued global nogoods perform clearly worse (30%) than the corresponding versions with (Dago & Verfaillie, 1996)'s valued nogoods (ADOPT-aos and ADOPT-dos). Since this suggests that valued global nogoods are not a promising alternative to (Dago & Verfaillie, 1996)'s valued nogoods, we did not evaluate their use in combination with other techniques explored here. Figure 24 also shows the behavior of a version of ADOPT-pon (implementation of ADOPT), without threshold nogoods. It took more than two weeks for the original ADOPT implementation to solve one of the problems for 20 agents and density 30%, and one of the problems for 25 agents and density 30% (at which moment the solver was interrupted). Therefore, it was evaluated using only the remaining 24 problems at those problem sizes. SRCs bring im-
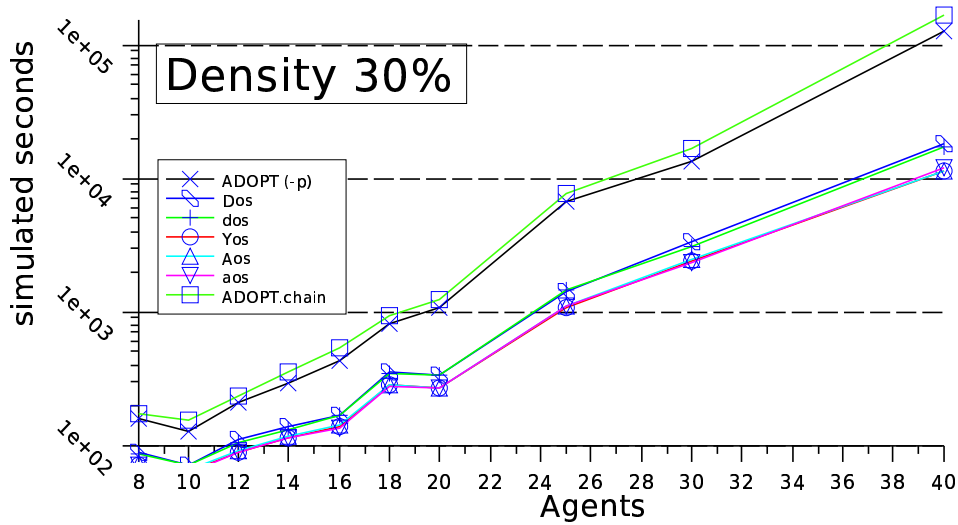
Figure 21: Simulated time in seconds, each latency being drawn randomly between 150ms and 250ms (for problems with density 30%).
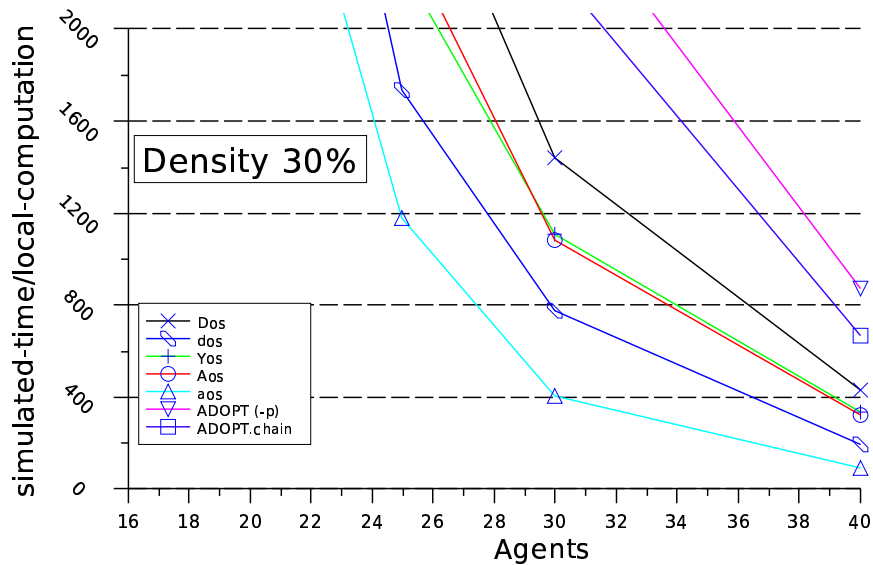

Figure 22: Ratio between total expected time where each latency is drawn randomly between 150ms and 250ms, and the local time of an agent (for problems with density 30%).

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

We tried to figure out the importance of using method (a) rather than method (b) in Step 1 of Remark 8 (comparing obtained versions ADOPT-Yos and ADOPT-Yos.b), and we found the two alternatives to be equally good (ADOPT-Yos being less than 1% better than ADOPT-Yos.b). We also evaluated the effects of optimizations in local computations, by computing the nogoods for an agent A_k based on the nogoods
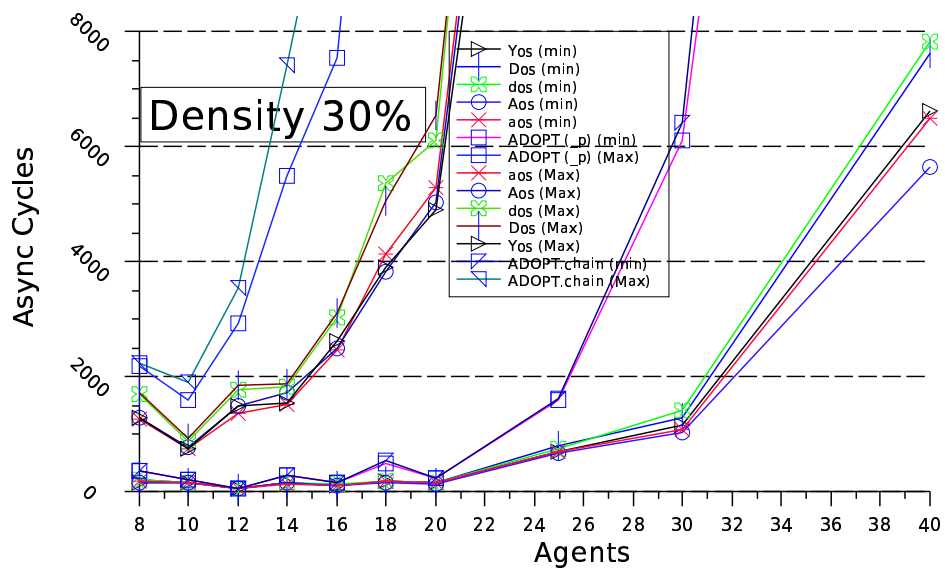
Figure 23: The minimum and the maximum number of asynchronous cycles for each problem size (at density 30%).
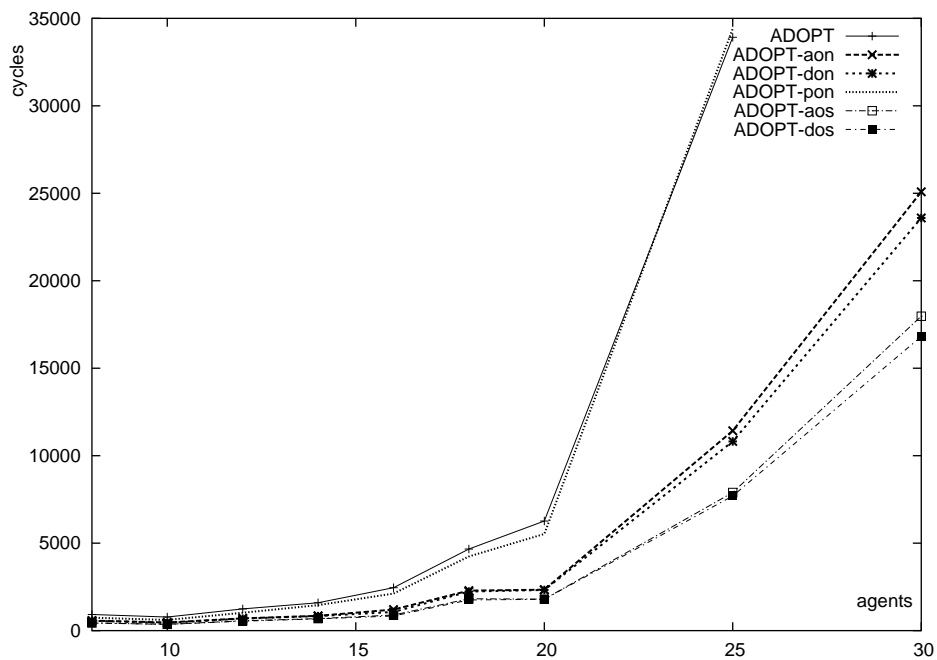

Figure 24: Longest causal chain of messages (synchronous cycles) for versions using valued global nogoods (and without threshold nogoods), at density 30%.
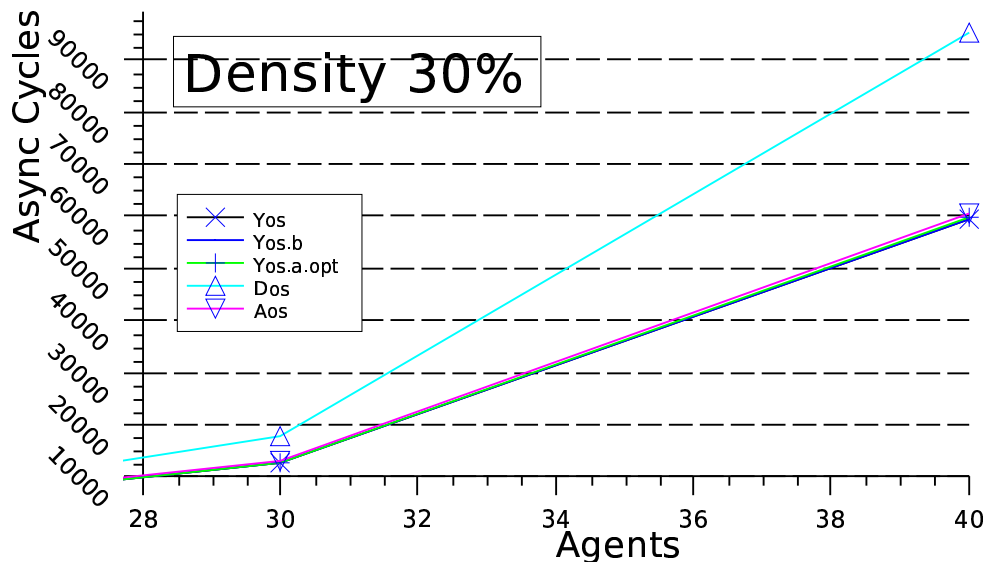
Figure 25: Local computations have little effect, but ADOPT-Yos is clearly better than ADOPT-Aos, competing with ADOPT-Dos. The optimized version of ADOPT-Yos is in average approximately 1% better than ADOPT-Dos (up to 15% better on some problem instances).

computed for higher priority agents rather than computing them from scratch (ADOPT-Yos.a.optim). The same figure shows the effect on asynchronous cycles to be minor (approximately 1% worse than ADOPT-Yos). The effect on constraint checks is similarly minor (4%) and is not depicted here.

Figure 24 clearly shows that the highest improvement in number of cycles is brought by sending valued nogoods to other ancestors besides the parent. The next factor for improvement with difficult problems (density .3) is the use of SRCs. The use of the structures of the DFS tree makes slight improvements in number of cycles (when nogoods reach all ancestors). 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 significant 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).

Experimental comparison with DPOP is redundant since its performance can be easily predicted. DPOP is a good choice if the induced width γ of the graph of the problem is smaller than $\log_d T/n$ and smaller than $\log_d S$, where T is the available time, n the number of variables, d the domain size, and S the available computer memory.

5. Conclusions

With the ADOPT distributed constraint optimization algorithm, an agent can communicate feedback only to a predefined predecessor, its parent in the DFS tree. The extension proposed here enables agents to send feedback to any relevant agent (fulfilling a research direction suggested in the original publication of ADOPT), bringing significant speed-up, and embodying a version of ADOPT on which one can apply the results related to the main algorithm for distributed constraint satisfaction, ABT.

ADOPT-ing can dynamically discover a DFS tree based only on the constraints that had been proved relevant by the search up to that moment. It uses (Dago & Verfaillie, 1996)'s valued nogoods tagging contexts with costs and with sets of references to culprit constraints. The generalized algorithm is denoted ADOPT-

ing. Tagging costs with sets of references to culprit constraints (SRCs) allows detection and exploitation of dynamically created independence between sub-problems. Such independence can be caused by assignments. Experimentation shows that it is important for an agent to infer and send in parallel several valued nogoods to different higher priority agents. It also shows 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). Precomputed DFS trees can still be used in conjunction with the valued nogood paradigm for optimization, thereby providing some additional improvements. ADOPT-ing versions detecting and/or exploiting DFS trees that we tested so far are also slightly better (in number of cycles and total messages) than the ones without DFS trees.

We isolated and evaluated the contribution of using threshold valued nogoods in ADOPT-ing. 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.

The use of SRCs to dynamically detect and exploit independence and the generalized communication of valued nogoods to several ancestors bring elegance and flexibility to the description and implementation of ADOPT in ADOPT-ing. They also produced experimental improvements of an order of magnitude.

Acknowledgments

References

Ali, S., Koenig, S., & Tambe, M. (2005). Preprocessing techniques for accelerating the DCOP algorithm ADOPT. In *AAMAS*.

Armstrong, A., & Durfee, E. F. (1997). Dynamic prioritization of complex agents in distributed constraint satisfaction problems. In *Proceedings of 15th IJCAI*.

Benisch, M., & Sadeh, N. (2006). Examining dcsp coordination tradeoffs. In *AAMAS*.

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

Bistarelli, S., Fargier, H., Montanari, U., Rossi, F., Schiex, T., & Verfaillie, G. (1996). Semiring-based CSPs and Valued CSPs: Basic Properties and Comparison. In Jampel, M., Freuder, E., & Maher, M. (Eds.), *Over-Constrained Systems (Selected papers from the Workshop on Over-Constrained Systems at CP'95, reprints and background papers)*, Vol. 1106, pp. 111–150.

Bistarelli, S., Montanari, U., & Rossi, F. (1995). Constraint solving over semirings. In *Proceedings IJCAI*, pp. 624–630, Montreal.

Bistarelli, S., Montanari, U., Rossi, F., Schiex, T., Verfaillie, G., & Fargier, H. (1999). Semiring-based CSPs and valued CSPs: Frameworks, properties, and comparison. *Constraints*, *4*(3), 199–240.

Chechetka, A., & Sycara, K. (2005). A decentralized variable ordering method for distributed constraint optimization. In *AAMAS*.

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

Collin, Z., Dechter, R., & Katz, S. (2000). Self-stabilizing distributed constraint satisfaction. *Chicago Journal of Theoretical Computer Science*.

Dago, P. (1997). Backtrack dynamique valué.. In *JFPLC*, pp. 133–148.

Dago, P., & Verfaillie, G. (1996). Nogood recording for valued constraint satisfaction problems.. In *ICTAI*, pp. 132–139.

Davin, J., & Modi, P. J. (2005). Impact of problem centralization in distributed cops. In *DCR*.

Dechter, R. (1990). Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *AI'90*.

Dechter, R. (2003). *Constraint Processing*. Morgan Kaufman.

Franzin, M., Rossi, F., E.C., F., & Wallace, R. (2004). Multi-agent meeting scheduling with preferences: efficiency, privacy loss, and solution quality. *Computational Intelligence*, *20*(2).

Freuder, E. C., & Wallace, R. J. (1992). Partial constraint satisfaction. *Artificial Intelligence*, *58*(1-3), 21–70.

Gershman, A., Meisels, A., & Zivan, R. (2006). Asynchronous forward-bounding for distributed constraints optimization. In *ECAI*.

Gershman, A., Meisels, A., & Zivan, R. (2007). Asynchronous forward-bounding with backjumping. In *IJCAI DCR Workshop*.

Ginsberg, M. L. (1993). Dynamic backtracking. *Journal of AI Research*, *1*.

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

Hamadi, Y., & Bessière, C. (1998). Backtracking in distributed constraint networks. In *ECAI'98*, pp. 219–223.

Hirayama, K., & Yokoo, M. (1997). Distributed partial constraint satisfaction problem. In *Proceedings of the Conference on Constraint Processing (CP-97),LNCS 1330*, pp. 222–236.

Jagota, A., & Dechter, R. (1997). Simple distributed algorithms for the cycle cutset problem. In *SAC '97: Proceedings of the 1997 ACM symposium on Applied computing*, pp. 366–373, New York, NY, USA. ACM Press.

Larrosa, J. (2002). Node and arc consistency in weighted csp. In *AAAI-2002*, Edmonton.

Liu, J., & Sycara, K. P. (1995). Exploiting problem structure for distributed constraint optimization. In *ICMAS*.

Maheswaran, R., Tambe, M., Bowring, E., Pearce, J., & Varakantham, P. (2004). Taking DCOP to the real world: Efficient complete solutions for distributed event scheduling. In *AAMAS*.

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

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

Meisels, A., Kaplansky, E., Razgon, I., & Zivan, R. (2002). Comparing performance of distributed constraints processing algorithms. In *AAMAS02 DCR Workshop*, pp. 86–93.

Meseguer, P., & Jiménez, M. (2000). Distributed forward checking. In *CP'2000 Distributed Constraint Satisfaction Workshop*.

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

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

Modi, P. J., Tambe, M., Shen, W.-M., & Yokoo, M. (2002). A general-purpose asynchronous algorithm for distributed constraint optimization. In *Distributed Constraint Reasoning, Proc. of the AAMAS'02 Workshop*, Bologna. AAMAS.

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

Petcu, A., & Faltings, B. (2005a). Approximations in distributed optimization. In *Principles and Practice of Constraint Programming CP 2005*.

Petcu, A., & Faltings, B. (2005b). A scalable method for multiagent constraint optimization. In *IJCAI*.

Petcu, A., & Faltings, B. (2006a). Distributed generator maintenance scheduling. In *Proceedings of the First International ICSC Symposium on ARTIFICIAL INTELLIGENCE IN ENERGY SYSTEMS AND POWER: AIESP'06*, Madeira, Portugal.

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

Ringwelski, G., & Hamadi, Y. (2005). Multi-directional distributed search with aggregation. In *IJCAI-DCR*.

Schiex, T., Fargier, H., & Verfaillie, G. (1995). Valued constraint satisfaction problems: hard and easy problems.. In *Procs. IJCAI'95*, pp. 631–637.

Silaghi, M.-C. (2002). *Asynchronously Solving Distributed Problems with Privacy Requirements*. PhD Thesis 2601, (EPFL). `http://www.cs.fit.edu/~msilaghi/teza`.

Silaghi, M.-C. (2003a). Asynchronous PFC-MRDAC±Adopt —consistency-maintenance in ADOPT—. In *IJCAI-DCR*.

Silaghi, M.-C. (2003b). Howto: Asynchronous PFC-MRDAC –optimization in distributed constraint problems +/-ADOPT–. In *IAT*, Halifax.

Silaghi, M.-C. (2006). Framework for modeling reordering heuristics for asynchronous backtracking. In *IAT*.

Silaghi, M.-C., & Faltings, B. (2002). A comparison of DisCSP algorithms with respect to privacy. In *AAMAS-DCR*.

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

Silaghi, M.-C., Landwehr, J., & Larrosa, J. B. (2004). Vol. 112 of *Frontiers in Artificial Intelligence and Applications*, chap. Asynchronous Branch & Bound and A* for DisWCSPs with heuristic function based on Consistency-Maintenance. IOS Press.

Silaghi, M.-C., & Mitra, D. (2004). Distributed constraint satisfaction and optimization with privacy enforcement. In *3rd IC on Intelligent Agent Technology*, pp. 531–535.

Silaghi, M.-C., Sam-Haroud, D., & Faltings, B. (2000). Asynchronous search with aggregations. In *Proc. of AAAI2000*, pp. 917–922, Austin.

Silaghi, M.-C., Sam-Haroud, D., & Faltings, B. (2001a). ABT with asynchronous reordering. In *IAT*.

Silaghi, M.-C., Sam-Haroud, D., & Faltings, B. (2001b). Hybridizing ABT and AWC into a polynomial space, complete protocol with reordering. Tech. rep. #01/364, EPFL.

Silaghi, M.-C., Sam-Haroud, D., & Faltings, B. (2001c). Consistency maintenance for ABT. In *Proc. of CP'2001*, pp. 271–285, Paphos,Cyprus.

Silaghi, M.-C., Sam-Haroud, D., & Faltings, B. (2000). Maintaining hierarchical distributed consistency. In *Workshop on Distributed CSPs*, Singapore. 6th International Conference on CP 2000.

Stallman, R. M., & Sussman, G. J. (1977). Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, *9*, 135–193.

Sultanik, E., Modi, P. J., & Regli, W. (2006). Constraint propagation for domain bounding in distributed task scheduling. In *CP*.

Wallace, R., & Silaghi, M.-C. (2004). Using privacy loss to guide decisions in distributed CSP search. In *FLAIRS'04*.

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

Yeoh, W., Koenig, S., & Felner, A. (2007). Idb-adopt : A depth first search dcop algorithm. In *IJCAI DCR Workshop*.

Yokoo, M. (1993). Constraint relaxation in distributed constraint satisfaction problem. In *ICDCS'93*, pp. 56–63.

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

Yokoo, M., Durfee, E. H., Ishida, T., & Kuwabara, K. (1998). The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE TKDE*, *10*(5), 673–685.

Yokoo, M., & Hirayama, K. (1998). Distributed constraint satisfaction algorithm for complex local problems. In *Proceedings of 3rd ICMAS'98*, pp. 372–379.

Yokoo, M., Suzuki, K., & Hirayama, K. (2002). Secure distributed constraint satisfaction: Reaching agreement without revealing private information. In *CP*.

Zhang, W., & Wittenburg, L. (2002). Distributed breakout revisited. In *Proc. of AAAI*, Edmonton.

Zivan, R., & Meisels, A. (2005). Dynamic ordering for asynchronous backtracking on discsps. In *CP*, pp. 161–172.