The Fifth International Workshop on

Distributed Constraint Reasoning

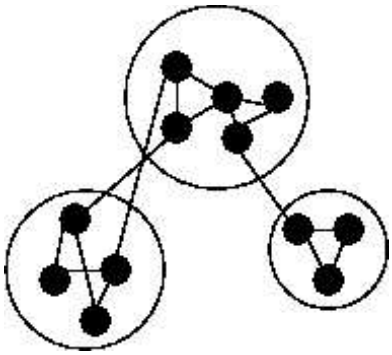
(DCR04)

Toronto, Canada

September 27, 2004

held in conjunction with

Tenth International Conference on Principles and Practice of Constraint Programming
(CP 2004)



Foreword

A common assumption behind many existing constraint programming techniques is that all information about problem variables and constraints is available locally. Distributed Constraint Reasoning (DCR) provides a framework for problem solving in which information and control about the problem is distributed among autonomous agents. This distributed model promises to more closely match the assumptions underlying an increasingly diverse range of real world multiagent problems.

This DCR workshop series addresses modeling, solutions and applications of Distributed Constraint Reasoning, including both Distributed Constraint Satisfaction and Optimization Problems. The goal of the DCR workshop series is to bring together researchers from the many different areas that are relevant to distributed constraint reasoning so that commonalities and relationships can be discovered and understanding improved. DCR is an inter-disciplinary research area involving the Constraint Programming, Multiagent Systems and AI communities. As such, this workshop has historically rotated its location between the three major conferences in each of these areas: CP (2000), IJCAI (2001, 2003) and AAMAS (2002). Building upon these previous successful workshops, we continue in 2004 with the Fifth International DCR workshop held in conjunction with CP 2004 in Toronto, Canada.

We trust that these proceedings will provide the reader with a glimpse of the cutting edge research currently going on in DCR. Looking into the future, we hope this workshop will contribute to the continuing growth of this exciting research area.

Pragnesh Jay Modi (Workshop Chair)

Program Committee

List of Papers

- *DCOP Games for Multi-agent Coordination (*J. Pearce, R. Maheswaran, M. Tambe
- *A Distributed, Complete Method for Multi-agent Constraint Optimization* (A. Petcu, B. Faltings)
- *Preprocessing Techniques for Distributed Constraint Optimization* (S. Ali, S. Koenig, M. Tambe)
- *Dynamic Distributed Backjumping* (V.Ngyuyen, D. Sam-Haroud, B. Faltings)
- *Incremental Constraint Propagation for Interleaved Distributed Backtracking* (G. Ringwelski)
- *Synchronous, Asynchronous and Hybrid algorithms for DisCSP* (I. Brito, P. Meseguer)
- Short Paper: *E-Privacy Requirements for Distributed E-Services* (S. Ghernaouti-Helie, M. Sfaxi)
- *The DDAC4 Algorithm for Arc-Consistency Enforcement in Dynamic and Distributed CSP* (G. Ringwelski)
- *Using Additional Information in DisCSPs Search* (A. Meisels, O. Lavee)
- *Multiagent Meeting Scheduling with Rescheduling* (J. Modi, M. Veloso)
- *On the Evaluation of DisCSP Algorithms* (I. Brito, F. Herrero, and P. Meseguer)
- *Message Delay and DisCSP Search Algorithms* (R. Zivan, A. Meisels)

DCOP Games for Multi-agent Coordination

Jonathan P. Pearce, Rajiv T. Maheswaran and Milind Tambe

University of Southern California, Los Angeles, CA 90089, USA
{jppearce, maheswar, tambe}@usc.edu

Abstract. Many challenges in multi-agent coordination can be modeled as distributed constraint optimization problems (DCOPs) but complete algorithms do not scale well nor respond effectively to dynamic or anytime environments. We introduce a transformation of DCOPs into graphical games that allows us to devise and analyze algorithms based on local utility and prove the monotonicity property of a class of such algorithms. The game-theoretic framework also enables us to characterize new equilibrium sets corresponding to a given degree of agent coordination. A key result in this paper is the discovery of a novel mapping between finite games and coding theory from which we can determine *a priori* bounds on the number of equilibria in these sets, which is useful in choosing the appropriate level of coordination given the communication cost of an algorithm.

1 Introduction

A distributed constraint optimization problem (DCOP) [9, 11] is a useful formalism in settings where distributed agents, each with control of some variables, attempt to optimize a global objective function characterized as the aggregation of distributed constraint utility functions. DCOPs can be applied for coordination in multi-agent domains, including sensor nets, distributed spacecraft, disaster rescue simulations, and software personal assistant agents. For example, sensor agents may need to choose appropriate scanning regions to optimize targets tracked over the entire network, or personal assistant agents may need to schedule multiple meetings in order to maximize the value of their users' time. As the scale of these domains become large, current complete algorithms incur immense computation costs. A large-scale network of personal assistant agents would require global optimization over hundreds of agents and thousands of variables, which is currently very expensive. Though heuristics that significantly speed up convergence have been developed [8], the complexity is still prohibitive in large-scale domains. On the other hand, if we let each agent or variable react on the basis of its local knowledge of neighbors and constraint utilities, we create a system that scales up very easily and is far more robust to dynamic environments.

Recognizing the importance of local search algorithms, researchers initially introduced DBA[12] and DSA[1] for Distributed CSPs, which were later extended to DCOPs with weighted constraints [13]. While detailed experimental analyses of these algorithms on DCOPs is available[13], we still lack theoretical tools that allow us to understand the evolution and performance of such algorithms on arbitrary DCOPs. To provide such tools, this paper decomposes a DCOP into an equivalent graphical *DCOP game*, which differs from graphical games with general reward functions [4, 10]. DCOP

games not only allow us to analyze existing local search algorithms, they also suggest an evolution to *k-coordinated* algorithms, where a collection of k agents coordinate their actions in a single negotiation round, which leads to new notions of equilibria. For example, a 2-coordinated algorithm would be an algorithm in which at most two agents could coordinate their actions, and a 2-coordinated equilibrium would be a situation in which no 2-coordinated algorithm could improve the quality of the assignment of values to variables. A key contribution of this paper is the application of a mapping between finite games and coding theory to determine *a priori* bounds on cardinality of equilibria sets of k-coordinated algorithms. Such bounds could be used to help determine an appropriate level of coordination for agents to use to reach an assignment of variables to values, in situations where the cost of coordination between multiple agents must be weighed against the quality of the solution reached.

2 DCOP Games, *k*-Coordinated Equilibria Sets and Bounds

We begin with a formal representation of a distributed constraint optimization problem and an exposition to our notational structure. Let $V \equiv \{v_i\}_{i=1}^{N}$ denote a set of variables, each of which can take a value $v_i = x_i \in X_i$, $i \in \mathcal{N} \equiv \{1, \ldots N\}$. Here, X_i will be a domain of finite cardinality $\forall i \in \mathcal{N}$. Interpreting each variable as a node in a graph, let the symmetric matrix E characterize a set of edges between variables/nodes such that $E_{ij} = E_{ji} = 1$ if an edge exists between v_i and v_j and $E_{ij} = E_{ji} = 0$, otherwise ($E_{ii} = 0\ \forall i$). For each pair (i, j) such that $E_{ij} = 1$, let $U_{ij}(x_i, x_j) = U_{ji}(x_j, x_i)$ represent a reward obtained when $v_i = x_i$ and $v_j = x_j$. We can interpret this as a utility generated on the edge between v_i and v_j, contingent simultaneously on the values of both variables and hence referred to as a *constraint*. The global or team utility $\overline{U}(x)$ is the sum of the rewards on all the edges when the variables choose values according to the assignment $x \in X \equiv X_1 \times \cdots \times X_N$. Thus, the goal is to choose an assignment, $x^* \in X$, of values to variables such that

$$x^* \in \arg\max_{x \in X} \overline{U}(x) = \arg\max_{x \in X} \sum_{i,j:E_{ij}=1} U_{ij}(x_i, x_j)$$

where x_i is the i-th variable's value under an assignment vector $x \in X$. This constraint *optimization* problem completely characterized by (X, E, U), where U is the collection of constraint utility functions, becomes *distributed* in nature when control of the variables is partitioned among a set of autonomous agents. For the rest of this paper, we make the simplifying assumption that there are N agents, each in control of a single variable.

We present a decomposition of the DCOP into a game as follows. Let v_j be called a *neighbor* of v_i if $E_{ij} = 1$ and let $\mathcal{N}_i \equiv \{j : j \in \mathcal{N}, E_{ij} = 1\}$ be the indices of all neighbors of the i-th variable. Let us define $x_{-i} \equiv [x_{j_1} \cdots x_{j_{K_i}}]$, hereby referred to as a *context*, be a tuple which captures the values assigned to the $K_i \equiv |\mathcal{N}_i|$ neighboring variables of the i-th variable, i.e. $v_{j_k} = x_{j_k}$ where $\cup_{k=1}^{K_i} j_k = \mathcal{N}_i$.

In a DCOP game, for an assignment x, we define a utility function $u_T(x)$ for a team of agents, $T \subseteq \mathcal{N}$ to be the sum of the utilities on all constraint links for which at least

one vertex represents an agent in the team, i.e.

$$u_T(x) = \sum_{i \in T} \sum_{j: E_{ij}=1} U_{ij}(x_i, x_j) - \sum_{i \in T} \sum_{j \in T, j>i, E_{ij}=1} U_{ij}(x_i, x_j).$$

The utility for a single agent ($T = \{i\}$) is

$$u_i(x) \equiv \sum_{j \in \mathcal{N}_i} U_{ij}(x_i, x_j)$$

Thus, in a DCOP game, team utilities are not the sums of individual utilities. We now have a *DCOP game* defined by (X, E, u_T) where u_T is a collection of the utility functions for all teams.

In current local algorithms, agents change values based on anticipated payoffs of only their own utilities. Since DCOPs are inherently cooperative, it is natural for agents to coordinate in order to improve global solution quality. DCOP games provide a framework to analyze, categorize and evaluate such multi-agent coordination. Let us define a *k-concurrent deviation* from an assignment x to be an assignment $\tilde{x}$ where exactly k of the N variables (agents) have values different from x, i.e. $d(x, \tilde{x}) \equiv |\{i : x_i \neq \tilde{x}_i\}| = k$. We now introduce the notion of a *k-coordinated equilibrium*, defined to be an assignment x^* such that if $\hat{k} \leq k$,any $\hat{k}$-concurrent deviation $\tilde{x}$ from x^*, i.e. $d(x^*, \tilde{x}) \leq \hat{k}$, cannot improve the team utility for the set of agents which deviated, $D(x^*, \tilde{x}) \equiv \{i : x_i^* \neq \tilde{x}_i\} \subseteq \mathcal{N}$. A 1-coordinated equilibrium is identical to a Nash equilibrium as $|D(x^*, \tilde{x})| = d(x^*, \tilde{x}) = 1$ is a unilateral deviation and the team utility u_T reduces to the utility u_i for a single agent. Let $X_{kE} \subseteq X$ be the subset of the assignment space which captures all k-coordinated equilibrium assignments:

$$X_{kE} \equiv \{x \in X : \tilde{x} \in X, 1 \leq d(x, \tilde{x}) \leq k \Rightarrow u_{D(x,\tilde{x})}(x) \geq u_{D(x,\tilde{x})}(\tilde{x})\}.$$

Proposition 1. *If x^* optimizes a DCOP characterized by (X, E, U), then $x^* \in X_{kE}$ $\forall$ $k \in \mathcal{N}$.*

Proof. Let us assume that x^* optimizes the DCOP (X, E, U) and $x^* \notin X_{kE}$ for some $k \in \mathcal{N}$. Then, there exists some $\tilde{x} \in X$ such that $u_{D(x^*,\tilde{x})}(x^*) < u_{D(x^*,\tilde{x})}(\tilde{x})$. By adding

$$\sum_{i \notin D(x^*,\tilde{x})} \sum_{j \notin D(x^*,\tilde{x}), j>i} U_{ij}(x_i^*, x_j^*) = \sum_{i \notin D(x^*,\tilde{x})} \sum_{j \notin D(x^*,\tilde{x}), j>i} U_{ij}(\tilde{x}_i, \tilde{x}_j)$$

to both sides, we can show $\overline{U}(x^*) < \overline{U}(\tilde{x})$, which is a contradiction. ■

Simply put, the proposition states that the optimal solution to the DCOP is a k-coordinated equilibrium for all k up to the number of variables in the system. In our DCOP framework, we are optimizing over a finite set. Thus, we are guaranteed to have an assignment that yields a maximum. By the previous proposition, this assignment is an element of X_{kE} $\forall k \in \mathcal{N}$, including $k = 1$. Thus, we are guaranteed the existence of a pure-strategy Nash equilibrium. This claim cannot be made for any arbitrary graphical game [4, 10]. Furthermore, from the definition above we see that for $k = 1, \ldots, N-1$, we have $X_{(k+1)E} \subseteq X_{kE}$ because if $x \in X_{(k+1)E}$, we have

$$d(x, \tilde{x}) \leq k + 1 \Rightarrow u_{D(x,\tilde{x})}(x) \geq u_{D(x,\tilde{x})}(\tilde{x})$$

which implies $d(x, \tilde{x}) \leq k \Rightarrow u_{D(x,\tilde{x})}(x) \geq u_{D(x,\tilde{x})}(\tilde{x})$ and thus, $x \in X_{kE}$. Thus, as k increases, the sets of k-coordinated equilibria can be pictured as a series of smaller and smaller concentric circles, culminating in a single point, representing the k-coordinated equilibrium for $k = N$, which is also the optimal solution to the DCOP.

In our notation X_{1E} characterizes the set of all Nash equilibria (no unilateral deviations) and X_{NE} characterizes the set of assignments that maximize global utility (no N-agent deviations).

We exploit the sets X_{kE} in the design of a new class of DCOP local algorithms, and analysis of their equilibrium points. In particular, for a given algorithm α, let Z_α denote the set of assignments at which the algorithm will remain stationary, i.e. the terminal states. An algorithm α is *k-coordinated* if $Z_\alpha \subseteq X_{kE}$ and $Z_\alpha \not\subseteq X_{(k+1)E}$ for $k < N$ or $Z_\alpha \subseteq X_{NE}$ for $k = N$.

Example 1. **Meeting Scheduling.** Consider two agents trying to schedule a meeting at either 7:00 AM or 1:00 PM with the constraint utility as follows: $U(7, 7) = 1, U(7, 1) = U(1, 7) = -100, U(1, 1) = 10$. If the agents started at $(7, 7)$, any 1-coordinated algorithm would not be able to reach the global optimum, while 2-coordinated algorithms would.

Section 3 illustrates that existing local DCOP algorithms are special cases of such k-coordinated algorithms with $k = 1$, and $k \geq 2$ may improve solution quality but at a higher communication cost.

Choosing an appropriate level of k-coordination given the higher communication cost is thus a critical question, similar to the choice of neighborhood size in large-neighborhood search in centralized constraint satisfaction. We assume that k-coordinated algorithms are capable of searching any neighborhood of size k completely, although the price for this completeness must be paid in the increasing number of messages required to ensure a k-equilibrium for increasing k.

To begin answering this question, we provide *a priori* bounds on the number of equilibria in sets X_{kE}, e.g. a significant reduction in number of equilibria may justify a jump from k-coordination to $(k + 1)$ coordination.

We first consider games, where each player (agent) can choose among q strategies (values), i.e. $|X_i| = q, \ \forall i \in \mathcal{N}$. We assume that the payoff structure is such that the optimal k-concurrent response to any context of cardinality $N - k$ is unique. Otherwise, any bound can be violated in the case where all assignments yield identical utilities and every assignment is an optimal equilibrium point. Furthermore, we assume that agents have the ability to communicate with all other agents to facilitate all k-concurrent deviations (although such communication may be indirect, requiring message relay).

To find upper bounds for the number of k-coordinated equilibria in such games, we discovered a correspondence from games to coding theory [6, 5]. A fundamental problem in the theory of error-correcting codes is the determination of appropriate codewords to use in a code. The code designer must balance the need for brevity, expressiveness, and error-correctability of the code, determined, respectively, by the length, maximum number, and distinctiveness of the allowed codewords. A common measure of the distinctiveness of two codewords is the Hamming distance, which is defined as the number of places at which the codewords differ.

For our purposes, an assignment is analogous to a codeword of length N from an alphabet of cardinality q (Each variable in the DCOP maps to a place in a codeword,

and each member of the domain of the variables maps to a member of the alphabet from which the codewords are created). An assignment $\tilde{x}$ which is a k-concurrent deviation from an assignment x, can also be interpreted as two codewords with a Hamming distance of k, where $d(x, \tilde{x}) \equiv |\{i : x_i \neq \tilde{x}_i\}| = k$ as stated earlier. If x_1 is a k-coordinated equilibrium and $\tilde{x}_1$ is a k-concurrent deviation from x_1, $\tilde{x}_1$ cannot be a k-coordinated equilibrium point because $u_{D(x,\tilde{x})}(x) > u_{D(x,\tilde{x})}(\tilde{x})$ since there is a unique optimal response to the context $\{x_i : i \in \mathcal{N} \setminus D(x, \tilde{x})$. Thus, if x_2 is a different k-coordinated equilibrium, then x_2 cannot be reachable from x_1 via a k-concurrent deviation (and vice-versa). In the language of coding theory, x_1 and x_2 must be separated by a Hamming distance greater than k. The problem of finding the maximum possible number of k-coordinated equilibria can then be reduced to finding the maximum number of codewords in a codespace of size q^N such that the the minimum distance among any two codewords is $d = k + 1$.

In coding theory literature, a q-ary (n, M, d) code refers to a collection of length n words constructed over an alphabet A of cardinality q where M codewords are chosen such that the minimum Hamming distance between any two codewords is at least d. Let $A_q(n, d) \equiv \max\{M : \exists \text{ an } (n, M, d) \text{ code over alphabet } A\}$. Three well-known bounds for $A_q(n, d)$ are the Hamming bound:

$$A_q(n, d) \leq q^n / \left(\sum_{i=0}^{\lfloor (d-1)/2 \rfloor} \binom{n}{i} (q-1)^i \right)$$

the Singleton bound:

$$A_q(n, d) \leq q^{n-d+1}$$

and the Plotkin bound:

$$A_q(n, d) \leq \left\lfloor \frac{d}{d - rn} \right\rfloor$$

Note that the Plotkin bound is only valid when $rn < d$, where $r = 1 - q^{-1}$, and $A_q^S(n, d) = q^{n-d+1}$ [5]. For the special case of binary ($q = 2$) codes, we can use the relation

$$A_q(n, 2r - 1) = A_q(n_1, 2r)$$

[6] to obtain tighter bounds for even distances using the Hamming bounds for odd distances. Thus, the number of k-coordinated equilibria for a given n, q and $d = k + 1$ can be bounded by the tightest of the bounds mentioned above.

For non-binary codes, we note that the Hamming bound is identical for d and $d + 1$ when d is odd. The Hamming bound is derived by using a sphere packing argument that states that the number of words q^n must be greater than the number of codewords $A_q(n, d)$ times the size of a sphere centered around each codeword. A sphere $S_A(u, r)$ with center u and radius r is the set $\{v \in A^n : d(u, v) \leq r\}$. It can be shown that $S_A(u, r)$ in A^n contains exactly $\sum_{i=0}^{r} \binom{n}{i} (q-1)^i$ words. If d is odd, the tightest packing then occurs with spheres of radius $(d-1)/2$ and each word can be uniquely assigned to the sphere of a codeword closest to it. If d is even, it is possible for a word to be equidistant from two codewords and it is unclear how to assign this word to a sphere. The Hamming bound

addresses this issue by simply using the bound obtained with the smaller distance $d-1$, which leads to smaller spheres and hence a larger bound than necessary. In essence, this ignores the contribution of a word that lies on the "boundary" to the volume of a sphere. We show one can appropriately partition these boundary assignments to achieve tighter bounds.

Proposition 2. *For even d,*

$$A_q(n,d) \leq \min\left\{ \frac{q^n - \binom{n}{d/2}(q-1)^{d/2}}{\sum_{i=0}^{\lfloor (d-1)/2 \rfloor}\binom{n}{i}(q-1)^i}, \frac{q^n}{\sum_{i=0}^{\lfloor (d-1)/2 \rfloor}\binom{n}{i}(q-1)^i + \binom{n}{d/2}(q-1)^{d/2}(\frac{1}{n})} \right\}.$$

Proof. It is clear that any word that has Hamming distance $\lfloor (d-1)/2 \rfloor$ or less from a codeword belongs in the sphere of that codeword, because belonging to more than one sphere under those conditions would violate the distance requirement of the code. Given an even distance, each codeword will see $\binom{n}{d/2}(q-1)^{d/2}$ words that are $d/2$ away from it. It cannot claim all those words as other codewords may be seeing the same words. We do know however that each of the words on the boundary can be seen by at most n codewords as a word of length n can be on the boundary of at most n spheres. Furthermore, each word on a boundary can be seen by at most $A_q(n,d)$ codewords, i.e. the number of codewords in the space. Thus, each codeword can safely incorporate $1/\min\{n, A_q(n,d)\}$ of each boundary word into its sphere. Aggregating over all the words on the boundary, we can increase the volume of the sphere by $\binom{n}{d/2}(q-1)^{d/2}/\min\{n, A_q(n,d)\}$. Using the sphere packing argument, if $A_q(n,d) \leq n$, we have

$$q^n \geq A_q(n,d)\left[\sum_{i=0}^{\lfloor (d-1)/2 \rfloor}\binom{n}{i}(q-1)^i + \frac{\binom{n}{d/2}(q-1)^{d/2}}{A_q(n,d)}\right]$$

$$\Rightarrow A_q(n,d) \leq \frac{q^n - \binom{n}{d/2}(q-1)^{d/2}}{\sum_{i=0}^{\lfloor (d-1)/2 \rfloor}\binom{n}{i}(q-1)^i} \equiv G_1,$$

and if $A_q(n,d) \geq n$, we have

$$q^n \geq A_q(n,d)\left[\sum_{i=0}^{\lfloor (d-1)/2 \rfloor}\binom{n}{i}(q-1)^i + \frac{\binom{n}{d/2}(q-1)^{d/2}}{n}\right]$$

$$\Rightarrow A_q(n,d) \leq \frac{q^n}{\sum_{i=0}^{\lfloor (d-1)/2 \rfloor}\binom{n}{i}(q-1)^i + \binom{n}{d/2}(q-1)^{d/2}(\frac{1}{n})} \equiv G_2.$$

Now, we have $A_q(n,d) \leq n \Rightarrow A_q(n,d) \leq G_1$ and $A_q(n,d) \geq n \Rightarrow A_q(n,d) \leq G_2$. We can show that $G_1 \odot n \Leftrightarrow G_2 \odot n,\ \forall \odot \in \{<, >, =\}$. Furthermore, $G_1 \odot n, G_2 \odot n \Leftrightarrow G_1 \odot G_2$. Thus, when $G_1 < n, G_2 < n$, we have both that G_2 is invalid and G_1 is the tighter bound and when $G_1 > n, G_2 > n$, G_1 is invalid and G_2 is the tighter bound. We can then express the bound as

$$A_q(n,d) \leq \min\{G_1, G_2\}.\blacksquare$$

We refer to this as the *modified Hamming bound*. The new bound appears to dominate other bounds for sufficiently large n, for even d and $q > 2$. In Figure 1, we illustrate the usefulness of our new bound.

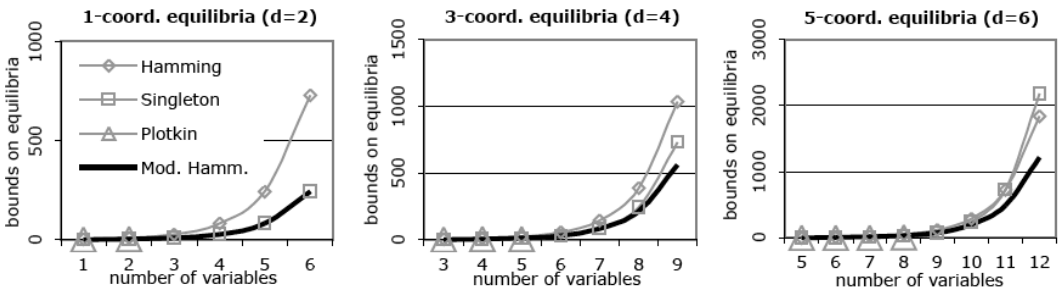


Fig. 1. Modified Hamming Bound

3 DCOP Algorithms: Analysis and Design

The *DCOP game* perspective also aids in the analysis of existing local-utility based algorithms and design of key new algorithms. Among existing DCOP algorithms, the first is the MGM (Maximum Gain Message) algorithm which is a modification of DBA (Distributed Breakout Algorithm) [12] focused solely on gain message passing. DBA cannot be directly applied because there is no global knowledge of solution quality which is necessary to detect local minima. The second is DSA (Distributed Stochastic Algorithm) [1], which is a homogeneous stationary randomized algorithm.

These algorithms work as follows: For synchronous running, let us define a *round* as the duration between a change in assignment for a particular algorithm. A single round could involve multiple broadcasts of *messages*. Every time a messaging phase occurs in a round, we will count that as one *cycle* and cycles will be our performance metric for speed, as is common in DCOP literature. Let $x^{(n)} \in X$ denote the assignments at the beginning of the n-th round. We assume that every algorithm will broadcast its current value to all its neighbors at the beginning of the round taking up one cycle. Once agents are aware of their current contexts, they will go through a process as determined by the specific algorithm to decide which of them will be able to modify their value. For MGM, each agent broadcasts a gain message to all its neighbors that represents the maximum change in its local utility if it is allowed to act under the current context. An agent is then allowed to act if its gain message is larger than all the gain messages it receives from all its neighbors (ties can be broken through variable ordering or other methods). For DSA, each agent generates a random number from a uniform distribution on $[0, 1]$ and acts if that number is less than some threshold p (the agent will only change value if there is a local utility gain). We note that MGM has a cost of two cycles per round while DSA has a cost of only one cycle per round.

Given the game-theoretic perspective introduced earlier, we recognize that MGM and DSA are in effect k-coordinated algorithms, where $k = 1$. In particular, these algorithms allow only unilateral actions by single agents in a given context. One method to improve the solution quality is for agents to coordinate actions with their neighbors, thus giving rise to k-coordinated algorithm classes. We define two such classes as MGM-k and SCA-k (Stochastic Coordination Algorithm), which facilitate monotonic and randomized evolution, respectively. DSA is in the SCA family of algorithms,

namely SCA-1. In these k-coordinated algorithms, teams of up to k agents can coordinate value updates in order to maximize $u_T(x)$ where T is the set of agents in the team.

Instantiating this concept in SCA-2, we allow agents to make offers to neighboring agents to perform a joint change of value, such that the sum of the utilities of the two agents will increase. They become committed partners if the offer receiver determines that team utility yields a greater gain than its unilateral move. To determine the roles of offerer or receiver, each agent generates a random number from a uniform distribution on $[0, 1]$ and becomes an offerer if that number is less than some threshold q, and a receiver otherwise.

Let $M^{(n)} \subseteq \mathcal{N}$ denote the set of agents allowed to modify the values in the n-th round. In SCA-2, $M^{(n)}$ includes all members of committed teams and uncommitted agents who update with probability p. In MGM-2, additional rounds of message exchanges ensures that if $i \in M^{(n)}$, then i belongs to a team (possibly a team of one) whose gain is larger than the gains of the teams of all its neighbors.

MGM, DSA, and MGM-2 are presented in full in the appendix.

Through our game-theoretic framework, we are able to prove the following monotonicity property of MGM-k, where teams of up to k agents can be formed.

Proposition 3. *When applying MGM, the global utility $\overline{U}(x^{(n)})$ is strictly increasing with respect to the round (n) until $x^{(n)} \in X_{NE}$.*

Proof. We assume $M^{(n)} \neq \emptyset$, otherwise we would be at a Nash equilibrium. When utilizing MGM, if $i \in M^{(n)}$ and $E_{ij} = 1$, then $j \notin M^{(n)}$. If the i-th variable is allowed to modify its value in a particular round, then its gain is higher than all its neighbors gains. Consequently, all its neighbors would have received a gain message higher than their own and thus, would not modify their values in that round. Because there exists at least one neighbor for every variable, the set of agents who cannot modify their values is not empty: ${M^{(n)}}^C \neq \emptyset$. We have $x_i^{(n+1)} \neq x_i^{(n)}\ \forall i \in M^{(n)}$ and $x_i^{(n+1)} = x_i^{(n)}\ \forall i \notin M^{(n)}$. Also, $u_i(x_i^{(n+1)}; x_{-i}^{(n)}) > u_i(x_i^{(n)}; x_{-i}^{(n)})\ \forall i \in M^{(n)}$, otherwise the i-th player's gain message would have been zero. Looking at the global utility, we have

$$
\begin{aligned}
&\overline{U}\left(x^{(n+1)}\right) \\
&= \sum_{i,j:E_{ij}=1} U_{ij}\left(x_i^{(n+1)}, x_j^{(n+1)}\right) \\
&= \sum_{\substack{i,j:i\in M^{(n)},\\ j\in M^{(n)},E_{ij}=1}} U_{ij}\left(x_i^{(n+1)}, x_j^{(n+1)}\right) + \sum_{\substack{i,j:i\in M^{(n)},\\ j\notin M^{(n)},E_{ij}=1}} U_{ij}\left(x_i^{(n+1)}, x_j^{(n+1)}\right) \\
&\quad + \sum_{\substack{i,j:i\notin M^{(n)},\\ j\in M^{(n)},E_{ij}=1}} U_{ij}\left(x_i^{(n+1)}, x_j^{(n+1)}\right) + \sum_{\substack{i,j:i\notin M^{(n)},\\ j\notin M^{(n)},E_{ij}=1}} U_{ij}\left(x_i^{(n+1)}, x_j^{(n+1)}\right) \\
&= \sum_{\substack{i,j:i\in M^{(n)},\\ j\notin M^{(n)},E_{ij}=1}} U_{ij}\left(x_i^{(n+1)}, x_j^{(n)}\right) + \sum_{\substack{i,j:i\notin M^{(n)},\\ j\in M^{(n)},E_{ij}=1}} U_{ij}\left(x_i^{(n)}, x_j^{(n+1)}\right) + \sum_{\substack{i,j:i\notin M^{(n)},\\ j\notin M^{(n)},E_{ij}=1}} U_{ij}\left(x_i^{(n)}, x_j^{(n)}\right)
\end{aligned}
$$

$$= \sum_{i \in M^{(n)}} u_i\left(x_i^{(n+1)}; x_{-i}^{(n)}\right) + \sum_{j \in M^{(n)}} u_j\left(x_j^{(n+1)}; x_{-j}^{(n)}\right) + \sum_{\substack{i,j: i \notin M^{(n)}, \\ j \notin M^{(n)}, E_{ij}=1}} U_{ij}\left(x_i^{(n)}, x_j^{(n)}\right)$$

$$> \sum_{i \in M^{(n)}} u_i\left(x_i^{(n)}; x_{-i}^{(n)}\right) + \sum_{j \in M^{(n)}} u_j\left(x_j^{(n)}; x_{-j}^{(n)}\right) + \sum_{\substack{i,j: i \notin M^{(n)}, \\ j \notin M^{(n)}, E_{ij}=1}} U_{ij}\left(x_i^{(n)}, x_j^{(n)}\right)$$

$$= \sum_{\substack{i,j: i \in M^{(n)}, \\ j \notin M^{(n)}, E_{ij}=1}} U_{ij}\left(x_i^{(n)}, x_j^{(n)}\right) + \sum_{\substack{i,j: i \notin M^{(n)}, \\ j \in M^{(n)}, E_{ij}=1}} U_{ij}\left(x_i^{(n)}, x_j^{(n)}\right) + \sum_{\substack{i,j: i \notin M^{(n)}, \\ j \notin M^{(n)}, E_{ij}=1}} U_{ij}\left(x_i^{(n)}, x_j^{(n)}\right)$$

$$= \overline{U}\left(x^{(n)}\right).$$

The second equality is due to a partition of the summation indexes. The third equality utilizes the properties that there are no neighbors in $M^{(n)}$ and that the values for variables corresponding to indexes not in $M^{(n)}$ in the $(n + 1)$-th round are identical to the values in the n-th round. The strict inequality occurs because agents in $M^{(n)}$ must be making local utility gains. The remaining equalities are true by definition. Thus, MGM yields monotonically increasing global utility until equilibrium. ■

Furthermore, it is clear that an equilibrium will be reached because this algorithm can be mapped to a discrete Hopfield model in which agents act as neurons which ”fire” by choosing a value. It has been shown that such networks always reach local equilibrium [3].

But why is monotonicity important? In anytime domains where communication may be halted arbitrarily and existing strategies must be executed, randomized algorithms risk being terminated at highly undesirable assignments. Given a starting condition with a minimum acceptable global utility, monotonic algorithms guarantee lower bounds on performance in anytime environments.

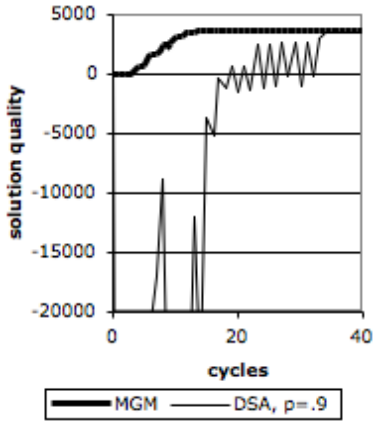


Fig. 2. MGM and DSA for a High-Stakes Scenario

Consider the example in Figure 2 which displays a sample trajectory for both MGM and DSA with identical starting conditions for a high-stakes scenario with 40 variables with three values each. Here, if two neighboring agents take the same value, a penalty of -1000 is incurred. If they take different values, they obtain a reward ranging from 10 to 100. To allow for a "safe" starting point for such a dangerous scenario, if two neighboring agents choose zero as their values, neither a reward nor a penalty is obtained. The figure is cropped to highlight the oscillation that occurs with DSA. In domains such as independent path planning of trajectories for UAVs or rovers, in environments where communication channels are unstable, bad assignments could lead to crashes whose costs preclude the use of methods without guarantees of monotonicity.

In addition, monotonicity provides insight as to why coordination might lead to better solution quality. If $k_2 > k_1$, we know that for all assignments x where $x \in X_{k_1 E}$, $x \notin X_{k_2 E}$, there exists an assignment $\tilde{x} \in X_{k_2 E}$ reachable from x such that $\overline{U}(\tilde{x}) > \overline{U}(x)$. This can be seen simply by running MGM-k_2 with initial assignment x.

Example 2. **The Traffic Light Game.** Consider two variables, both of which can take on the values *red* or *green*, with a constraint that takes on utilities as follows:

$$U(red, red) = 0, U(red, green) = U(green, red) = 1, U(green, green) = -1000.$$

Turning this DCOP into a game would require the agent for each variable to take the utility of the single constraint as its local utility. If (red, red) is the initial condition, each agent would choose to alter its value to *green* if given the opportunity to move. If both agents are allowed to alter their value in the same round, we would end up in the adverse state $(green, green)$. When using DSA, there is always a positive probability for any time horizon that $(green, green)$ will be the resulting assignment.

4 Experiments

We considered two domains. The first was a standard graph-coloring scenario, in which a cost of one is incurred if two neighboring agents choose the same color, and no cost is incurred otherwise. Real-world problems involving sensor networks, in which it may be undesirable for neighboring sensors to be observing the same location, are commonly mapped to this type of graph-coloring scenario. The second was a fully randomized DCOP, in which every combination of values on a constraint between two neighboring agents was assigned a random reward chosen uniformly from the set $\{1, \ldots, 10\}$.

In both domains, we used ten randomly generated graphs with 40 variables with three values each, and 120 constraints. We ran: MGM, DSA with $p \in \{.1, .3, .5, .7, .9\}$, MGM-2 with $q \in \{.1, .3, .5, .7, .9\}$ and SCA-2 with all combinations of the above values of p and q (where q is the probability of being an offerer and p is the probability of an uncommited agent acting). Each graph shows an evolution of global solution quality averaged over 100 runs (with random start-states) each for ten examples with selected values of p and q.

We used communication cycles as the metric for our experiments, as is common in the DCOP literature, since it is assumed that communication is the speed bottleneck. However, we note that, as we move from 1-coordinated to 2-coordinated algorithms, the

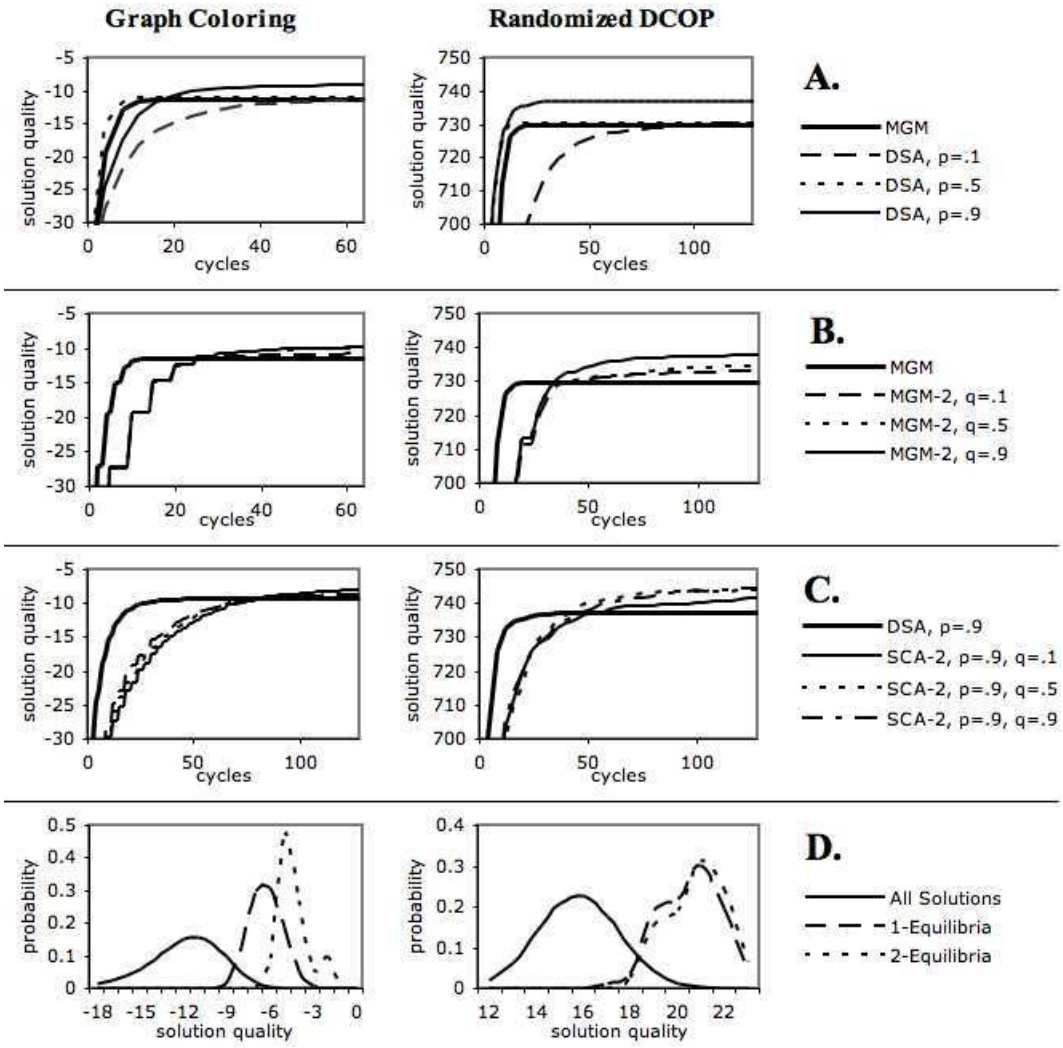


Fig. 3. Experimental results

computational cost each agent i must incur can increase by a factor of as much as $\sum_j |X_j|$ as the agent can now consider the combination of its and all its neighbors' moves. However, in the 2-coordinated algorithms we present, each agent randomly picks a single neighbor j to coordinate with, and so its computation is increased by a factor of only $|X_j|$. Although each run was 256 cycles, the graphs display a cropped view to show the important phenomena.

Figure 3A shows a comparison between MGM and DSA for several values of p. For graph coloring, MGM is dominated, first by DSA with $p = 0.5$, and then by DSA with $p = 0.9$. For the randomized DCOP, MGM is completely dominated by DSA with $p = 0.9$. MGM does better in the high-stakes scenario as all DSA algorithms have a negative solution quality (not shown in the graph) for the first few cycles. This happens because at the beginning of a run, almost every agent will want to move. As the value of p increases, more agents act simultaneously, and thus, many pairs of neighbors are choosing the same value, causing large penalties. Thus, these results show that the nature of the constraint utility function makes a fundamental difference in which algorithm dominates. Results from the high-stakes scenario contrast with [13] and show that DSA is not necessarily the algorithm of choice compared with DBA across all domains.

Figure 3B shows a comparison between MGM and MGM-2, for several values of q. In all domains, MGM-2 eventually reaches a higher solution quality after about thirty cycles, despite the algorithms' initial slowness. The stair-like shape of the MGM-2 curves is due to the fact that agents are changing values only once out of every five cycles, due to the cycles used in communication. Of the three values of q shown in the graphs, MGM-2 rises fastest when $q = 0.5$, but eventually reaches its highest average solution quality when $q = 0.9$, for each of the three domains. We note that, in the high-stakes domain, the solution quality is positive at every cycle, due to the monotonic property of both MGM and MGM-2. Thus, these experiments clearly verify the monotonicity of MGM and MGM-2, and also show that MGM-2 reaches a higher solution quality as expected.

Figure 3C shows a comparison between DSA and SCA-2, for $p = 0.9$ and several values of q. DSA starts out faster, but SCA-2 eventually overtakes it. The result of the effect of q on SCA-2 appears inconclusive. Although SCA-2 with $q = 0.9$ does not achieve a solution quality above zero for the first 65 cycles, it eventually achieves a solution quality comparable to SCA with lower values of q.

Figure 3D shows a probability mass function (PMF) of solution quality for three sets of assignments: the set of all assignments in the DCOP (X), the set of 1-coordinated (Nash) equilibria (X_{1E}), and the set of 2-coordinated equilibria (X_{2E}). Here we considered smaller scenarios with twelve variables, 36 constraints, and three values per variable in order to investigate tractably explorable domains. In both domains, the solution quality of the set of 2-coordinated equilibria (the set of equilibria to which MGM-2 and SCA-2 must converge) is, on average, higher than the set of 1-coordinated equilibria, potentially explaining the higher solution quality of the experimental runs. Even though a higher level of coordination yields better solution quality, the relationship between magnitude of improvement and the difference in solution qualities of the equilibrium sets is not obvious. Trajectories may not be uniformly distributed over the equilibrium sets. Investigating these effects is a ripe area for further investigation.

5 Related Work and Summary

Research in general graphical games has focused on centralized algorithms for finding mixed-strategy Nash equilibria [4, 10]. DCOP games not only guarantee pure-strategy Nash equilibria but also introduce *k*-coordination and hence *k*-coordinated equilibria. In [2], coordination was achieved by forming coalitions represented by a *manager* who made the assignment decisions for all variables within the coalition. These methods require high-volume communication to transfer utility function information and the abdication of authority which is often infeasible or undesired in many distributed decision-making environments. Furthermore, the cost of forming a coalition discourages rapid commitment and detachment from teams. MGM-*k* and SCA-*k* allow for coordination while maintaining the underlying distributed decision-making process and allowing dynamic teaming in each round.

A fundamental novelty of our approach is our analysis of distributed *k*-coordination algorithms as well as *k*-coordinated equilibria. The key contributions of this paper include: (i) an introduction of *DCOP games* for analysis of DCOP algorithms, (ii) development of *k*-coordinated DCOP algorithms, (iii) identification of a mapping between finite games and coding theory leading to *a priori* bounds on cardinality of equilibria sets of *k*-coordinated algorithms, (iv) improvement on the tightness of current bounds, (v) proof of monotonicity of the MGM-*k* class of algorithms and (vi) an investigation of the equilibria sets of algorithms of differing degrees of coordination.

We provided key experimental results, verifying our conclusions about monotonicity and equilibria bounds. This paper is a significant extension of the authors' previous work in DCOP games [7], in which *k*-coordinated algorithms and equilibria were introduced. Our results comparing 1-coordinated and 2-coordinated algorithms illustrate the need to develop efficient *k*-coordination algorithms for higher *k* in the future.

6 Acknowledgment

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA), through the Department of the Interior, NBC, Acquisition Services Division, under Contract No. NBCHD030010.

References

1. S. Fitzpatrick and L. Meertens. Distributed coordination through anarchic optimization. In V. Lesser, C. L. Ortiz Jr., and M. Tambe, editors, *Distributed Sensor Networks: A Multiagent Perspective*, pages 257–295. Kluwer Academic Publishers, 2003.
2. K. Hirayama and J. Toyoda. Forming coalitions for breaking deadlocks. In *Proc. ICMAS*, pages 155–162, 1995.
3. J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79:2554–8, 1982.
4. M. Kearns, M. Littman, and S. Singh. Graphical models for game theory. In *Proc. UAI*, pages 253–260, 2001.
5. S. Ling and C. Xing. *Coding theory: A first course*. Cambridge University Press, 2004.

6. F. J. MacWilliams and N. J. A. Sloane. *The theory of error-correcting codes*. North-Holland, 1977.
7. R. T. Maheswaran, J. P. Pearce, and M. Tambe. Distributed algorithms for DCOP: A graphical-game-based approach. In *PDCS 2004*, San Francisco, CA, September 2004.
8. R. T. Maheswaran, M. Tambe, E. Bowring, J. P. Pearce, and P. Varakantham. Taking DCOP to the real world: efficient complete solutions for distributed multi-event scheduling. In *AAMAS 2004*, New York, NY, July 2004.
9. P. J. Modi, W. Shen, M. Tambe, and M. Yokoo. An asynchronous complete method for distributed constraint optimization. In *Proceedings of the Second International Conference on Autonomous Agents and Multi-Agent Systems*, Sydney, Australia 2003.
10. D. Vickrey and D. Koller. Multi-agent algorithms for solving graphical games. In *Proc. AAAI*, pages 345–351, 2002.
11. M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, 1998.
12. M. Yokoo and K. Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction and optimization problems. In *Proc. ICMAS*, pages 401–408, 1996.
13. W. Zhang, Z. Xing, G. Wang, and L. Wittenburg. An analysis and application of distributed constraint satisfaction and optimization algorithms in sensor networks. In *AAMAS 2003*, pages 185–192, Melbourne, Australia, July 2003.

Appendix A: Algorithms

Algorithm 1 MGM (allNeighbors, currentValue)

```
1: SendValueMessage(allNeighbors, currentValue)
2: currentContext = GetValueMessages(allNeighbors)
3: [gain,newValue] = BestUnilateralGain(currentContext)
4: SendGainMessage(allNeighbors,gain)
5: neighborGains = ReceiveGainMessages(allNeighbors)
6: if gain > max(neighborGains) then
7:     currentValue = newValue
8: end if
```

Algorithm 2 DSA (allNeighbors, currentValue)

```
1: SendValueMessage(allNeighbors, currentValue)
2: currentContext = GetValueMessages(allNeighbors)
3: [gain,newValue] = BestUnilateralGain(currentContext)
4: if  Random(0,1) < threshold then
5:     currentValue = newValue
6: end if
```

Algorithm 3 MGM-2 (allNeighbors, currentValue)

```
SendValueMessage(allNeighbors, currentValue)
currentContext = GetValueMessages(allNeighbors); committed = no
if  Random(0,1) < offererThreshold then
    committed = yes; partner = RandomNeighbor(allNeighbors)
    SendOfferMessage(partner,allCoordinatedMoves(partner))
end if
[gain,newValue] = BestUnilateralGain(currentContext)
offers = ReceiveOffers(allNeighbors); offerReplySet = ∪ {offers.neighbor}
if committed = no then
    bestOffer = FindBestOffer(offers)
    if bestOffer.gain > gain then
        offerReplySet = offerReplySet \{ bestOffer.neighbor}
        committed = yes; partner = bestOffer.neighbor
        newValue = bestOffer.myNewValue; gain = bestOffer.gain
        SendOfferReplyMessage(partner, commit, bestOffer.partnerNewValue, gain)
    end if
    for all neighbor ∈ offerReplySet do
        SendOfferReplyMessage(neighbor, noCommit)
    end for
end if
if committed = yes then
    reply = ReceiveOfferReplyMessage(partner)
    if reply = commit then
        newValue = reply.myNewValue; gain = reply.gain
    else
        committed = no
    end if
end if
SendGainMessage(allNeighbors,gain)
neighborGains = ReceiveGainMessages(allNeighbors); changeValue=no
if committed=yes then
    if gain > max(neighborGains) then
        SendConfirmMessage(partner, go)
    else
        SendConfirmMessege(partner, noGo)
    end if
    confirmed = ReceiveConfirmMessage(partner)
    if confirmed=yes then
        changeValue=yes
    end if
else
    if gain > max(neighborGains) then
        changeValue=yes
    end if
end if
if changeValue=yes then
    currentValue = newValue
end if
```

A Distributed, Complete Method for Multi-Agent Constraint Optimization

Adrian Petcu[1] and Boi Faltings[1]

Ecole Politechnique Federale de Lausanne (EPFL)
CH-1015 Lausanne, Switzerland
{adrian.petcu, boi.faltings}@epfl.ch
http://liawww.epfl.ch/

Abstract. We present in this paper a new complete method for distributed constraint optimization. This is a utility-propagation method, inspired by the sum-product algorithm [6]. The original algorithm requires fixed message sizes, linear memory, and is time-linear in the size of the problem. However, it is correct only for tree-shaped constraint networks. In this paper, we show how to extend the algorithm to arbitrary topologies using cycle cutsets, while preserving the linear message size and memory requirements. We present some preliminary experimental results on randomly generated problems. The algorithm is formulated for optimization problems, but can be easily applied to satisfaction problems as well.

1 Introduction

Distributed Constraint Satisfaction (DisCSP) was first studied by Yokoo [10] and has recently attracted increasing interest. In distributed constraint satisfaction, variables and constraints are distributed so that each variable and constraint is owned by an agent. Systematic search algorithms for solving DisCSP are generally derived from depth-first search algorithms based on some form of backtracking [9, 11, 12, 7, 3]. Recently, the paradigm of asynchronous distributed search has been extended to constraint optimization by integrating a bound propagation mechanism (ADOPT - [8]).

Backtracking algorithms are very popular in centralized systems because they require very little memory. In a distributed implementation, however, they may not be the best basis since in backtrack search, control shifts rapidly between different variables. Thus, every state change in a distributed backtrack algorithm requires at least one message. Furthermore, in the worst case even in a parallel algorithm there will be exponentially many state changes [5], thus resulting in exponentially many messages.

This leads us to believe that other search paradigms, in particular those based on dynamic programming, may be more appropriate for DisCSP. For example, an algorithm that incrementally computes the set of all partial solutions for all previous variables according to a certain order would only use a linear number of messages. However, the messages could grow exponentially in size, and the algorithm would not have any parallelism.

Recently, the sum-product algorithm [6] has become popular for certain constraint satisfaction problems, for example decoding. It is an acceptable compromise as it combines a dynamic-programming style exploration of a search space with a fixed message

size, and can easily be implemented in a distributed fashion. However, it is correct only for tree-shaped constraint networks. In this paper, we show how to extend the algorithm to arbitrary topologies using cycle cutsets, and report on initial experiments with randomly generated problems. The algorithm is formulated for optimization problems, but can be easily applied to the satisfaction problem by having relations with utility either 0 or 1.

2 Definitions & notation

Definition 1. *A discrete* multiagent constraint optimization problem *(MCOP) is a tuple* $< \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{R} >$ *such that:*

- $\mathcal{A} = \{A_1, ..., A_n\}$ *is the set of agents interested in the problem/solution;*
- $\mathcal{X} = \{X_1, ..., X_m\}$ *is the set of variables/solving agents;*
- $\mathcal{D} = \{d_1, ..., d_m\}$ *is a set of domains of the variables, each given as a finite set of possible values.*
- $\mathcal{R} = \{r_1, ..., r_p\}$ *is a set of relations, where a relation* r_i *is a function* $d_{i1} \times .. \times d_{ik} \rightarrow \Re^+$ *which is expressed by an agent* A_i*, and denotes how much utility that agent assigns to each possible combination of values of the involved variables.*

We chose to model the problem in this way (with two separate sets of agents) having in mind a social-choice-like problem, where a set of agents (A_i) are the "citizens" interested in choosing an outcome denoted by the assignment of values to variables (X_j) that are controlled by some public authorities.

In this paper we deal with unary and binary relations, being well-known that higher arity relations can also be expressed in these terms with little modifications. In a MCOP, any value combination is allowed; the goal is to find an assignment $\mathcal{X}^*$ for the variables X_i that maximizes the sum of utilities of all the agents $\mathcal{A}$.

A tree-structured problem is a tree network in which we can have several links (constraints) belonging to different agents between two adjacent nodes. Furthermore, unary constraints on each variable are also allowed.

For a node X_k, we define:

- $R^i(X_k)$: constraints of arity i on X_k (where i is 1 or 2)
- $Ngh(X_k)$: the neighbors of X_k
- R_k: the set of constraints belonging to agent X_k
- $R_k(X_j)$: constraints between X_k and its neighbor X_j

3 Distributed constraint optimization for tree-structured networks

For tree-structured networks (see an example in Figure 1), it is possible to devise polynomial-time complete optimization methods (e.g. the sum-product algorithm [6])

In this problem setting there is a set $\mathcal{X}$ of agents (each agent X_i is responsible for a variable), and a set $\mathcal{A}$ of agents that are interested in the assignments that are made for the variables $\mathcal{X}$. All the agents A_i declare their relations R_i to the agents X_i concerned

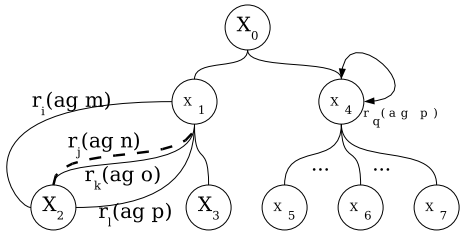


Fig. 1. *Problem example where the underlying constraint graph is a tree.*

in those relations (each relation is declared only to the 2 agents X_j and X_k involved - assuming binary constraints, or to a single agent in the case of unary constraints). We assume that the resulting constraint graph is a tree.

The "normal" agents - A_i participate in this process only by specifying their relations; in the optimization itself, they have a passive role; only the "variable-agents" will play an active role. Therefore, in the following, while explaining the optimization process, by "agent", or "node" we will mean one of the agents X_i.

In this protocol, agents send messages to each other; the leaf nodes initiate the process, and then the other nodes relay the messages according to the following rule:

Definition 2. *The **k-1 rule**: if node X_i has k neighbors, X_i will send out a message to its k^{th} neighbor only after having received the other k-1 messages, and will send out the rest of k-1 messages after having received the message from the k^{th} neighbor.*

Each agent X_i executes Algorithm 1:

- In the beginning, examine its own relations. All the other agents that are connected through relations with the current node will be its neighbors. During the algorithm an agent communicates only with its neighbors.
- Each agent determines whether it is a leaf in the constraint tree or not (if it has a single neighbor, even if they share multiple constraints) If X_i is a leaf node, then send the *UTIL* message to its only neighbor.
- Wait for incoming messages and respond to them.
  The messages passed in this system are in fact utility vectors; a neighbor X_j of node X_i would send X_i a vector of all the optimal utilities that can be achieved for the subtree rooted at X_i that contains X_j, for each of X_i's possible values (thus, the size of each message is $|dom(X_i)|$
  The agents send messages to their neighbors following the *k-1 rule*. Upon receiving $k-1$ messages from the neighbors, since all of the respective subtrees are disjoint, by summing them up, X_i computes how much utility each of its values gives for the whole set of $k-1$ subtrees. This, together with the relation(s) between X_i and the last neighbor, enable X_i to compute exactly how much utility can be achieved by the entire subtree rooted at the last neighbor and containing X_i, for each of this neighbor's values. Thus, X_i can send to its last (k^{th}) neighbor its *UTIL* message.
  Eventually, the last neighbor would also send its message back to X_i, and at this point X_i would be able to pick the optimal value for itself (as the value that max-

imizes the sum of the utilities of all subtrees rooted at itself, and of any unary constraints on itself, if any).

At this point, the algorithm is finished for X_i.

Proposition 1. *Algorithm 1 is sound and complete.*

PROOF.

Correctness: since there are no cycles in the problem, it means that all messages that a node X_i receives from its neighbors come from disjoint parts of the constraint problem. They represent exact evaluations of the utility that can be obtained by the subtrees rooted at the sender nodes, for each possible value that X_i can take (can be inferred by induction from the leaves inside the tree) By summing all messages up, X_i has accurate upper bounds on the amount of utility obtained from the whole problem, for each of its values; it is therefore easy to pick the one that gives the maximum utility.

Liveness: again, since there are no cycles in the problem, and all the leaves initiate the message propagation, it is guaranteed that each node will eventually receive *k-1* messages (with k=the number of neighbors) and therefore it will be able to send its k^{th} message. Therefore, it will also receive the final message from the last neighbor, leading to the conclusion of the algorithm for this node. □

Proposition 2. *Algorithm 1 is linear in the number of variables - there are exactly* $2 \times (n-1)$ *messages propagated through the system (where n is the number of agents in the system)*

PROOF. In a tree there are exactly $n-1$ edges between the n nodes of the tree (if less than $n-1$, then we have a set of disconnected problems which we can treat separately, if more, the problem is not a tree anymore). Along each edge, there are exactly 2 messages going through (one from each of the nodes connected through the edge) □

Observations In this algorithm, the agents do not assume any knowledge of the problem structure, and do not have parent-child relationships. All they need to know is whether they are leaf nodes or not (a leaf node has only 1 neighbor), and a way to distinguish between neighbors (ids).

The execution of Algorithm 1 proceeds in an asynchronous fashion from the leaves, traversing the tree and going to other leaves. This means that certain subtrees of the problem proceed faster than others, and it's not always the case that a "child" node is the first to send a *UTIL* message to its "parent" (like it would happen in a centralized setting); it can also happen the other way around (consider the example from Figure 1: it could happen that nodes X_2,X_3 and X_1 finish their processing faster, and X_1 delivers the *UTIL* message toX_0; then, contrary to the centralized setting, X_0 would send its message to X_4 *before* X_4 manages to send its message to X_0). In a sense, the "root" of this tree is dynamically determined, as the single node that happens to receive messages from all its neighbors before being able to send out any message.

Algorithm 1: *DTREE - Distributed optimization procedure for tree-structured networks.*

DTREE: distributed tree-optimization($\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{R}$)
We have a set of agents $X_i \in \mathcal{X}$ that each controls its variable, and a set of agents $A_i \in \mathcal{A}$ that are interested in the assignments of the variables X_i
All agents A_i declare their relations R_i to the subset of agents concerned about those constraints. We assume that the resulting constraint graph is a tree.
Each agent X_i executes:

Initialization(X_i, R_i)
$R_i \leftarrow$ the set of relations binding X_i
$Ngh(X_i) \leftarrow$ the neighbors of X_i (based on R_i)
for all $X_k \in Ngh(X_i)$ **do**
  send $Dom(X_i)$ to X_k
  receive and record $Dom(X_k)$
if $|Ngh(X_i)| == 1$ (i.e. X_i is a leaf node) **then**
  let X_k be the single element in $Ngh(X_i)$
  let $utils_{X_i}(X_k) \leftarrow$ Compute_utils(X_k)
  Send_message($X_k, utils_{X_i}(X_k)$)
$msg\_cnt \leftarrow 0$
activate Message_handler()
return

Message_handler(X_k,$utils_{X_k}(X_i)$)
store X_k, $utils_{X_k}(X_i)$
msg_cnt ++
if $msg\_cnt = |Ngh(X_i)| - 1$ **then**
  let X_j be the only neighbor that did not send $utils_{X_j}(X_i)$ yet
  let $utils_{X_i}(X_j) \leftarrow$ Compute_utils(X_j)
  Send_message($X_j, utils_{X_i}(X_j)$)
else
  if $msg\_cnt = |Ngh(X_i)|$ **then**
    for all $X_l \in \{Ngh(X_i) \setminus X_j\}$ **do**
      let $utils_{X_i}(X_l) \leftarrow$ Compute_utils(X_l)
      Send_message($X_l, utils_{X_i}(X_l)$)

$$v_i^* \leftarrow argmax_{v_i} \left(\sum_{X_l \in Ngh(X_i)} utils_{X_l}(X_i = v_i) + \sum_{r_i \in R^1(X_i)} r_i(v_i) \right)$$

    $X_i \leftarrow v_i^*$
    FINISH_ALGORITHM
return

Compute_utils(X_j)
for all $v_j \in Dom(X_j)$ **do**
  for all $v_i \in Dom(X_i)$ **do**

$$Util_{X_j}(v_i, v_j) \leftarrow \sum_{r_i \in R^1(X_i)} r_i(v_i) + \sum_{r_i \in R_i(X_j)} r_i(v_i, v_j) + \sum_{X_l \in \{Ngh(X_i) \setminus X_j\}} utils_{X_l}(X_i = v_i)$$

  $v_i^*(v_j) \leftarrow argmax_{v_i}(Util_{X_j}(v_i, v_j))$
return a vector $utils_{X_i}(X_j)$ of all $\{Util_{X_j}(v_i^*(v_j), v_j) | v_j \in Dom(X_j)\}$

Send_message($X_j, utils_{X_i}(X_j)$)
send the utils vector to agent X_j
return

4 Distributed constraint optimization for general networks

The scenario is similar to the one for tree networks, except that we can now drop the assumption that the constraint network is a tree. We will show in the following how the previous algorithm must be modified to accommodate this change.

First, let us consider what would happen if we would directly apply the *DTREE* algorithm to a graph. The fact that the constraint network has cycles breaks the *liveness* argument from Proposition 1 and leads to a deadlock in the execution of the algorithm: messages would still circulate through all the *TREE* parts of the problem, hanging from nodes involved in cycles; however, in a cycle there are no leaf nodes to initiate the message propagation, so the nodes involved in it wait for incoming messages indefinitely.

Based on this observation, we can devise a very simple cycle detection mechanism: whenever some nodes reach a (reasonably chosen) timeout while waiting for (some of) their neighbors to send messages, that means that those nodes are involved in a cycle with the neighbors that did not yet send their messages.

4.1 Cycle cutset

It has been pointed out in the literature [2, 4, 1] that breaking a problem with cycles into cycle-free parts can greatly improve the search performance for centralized, crisp CSPs. In the following, we will try to use this idea to find optimal solutions for *optimization* problems, in a *distributed* fashion.

The basic idea of such a technique would be to identify the nodes involved in cycles, select a subset of these nodes that will act as *cycle cuts*, apply an algorithm similar to *DTREE* to the now cycle-free parts of the problem, and in the end, put together the partial results in a coherent fashion. The rest of this section explains how this can be done.

4.2 Definitions

Node labeling In our model, the nodes of the constraint graph are labeled in one of the following ways:

1. *TREE* (nodes that have at most one path from themselves to at most one *CycleCut* node) - initially only leaf nodes are labeled *TREE*.
2. *Cycle* (nodes that are "between" several CC nodes - there is more than one path from themselves to other CC nodes) - initially all but the leaf nodes are *Cycle*. As a *Cycle* node receives $k-1$ (where k is the number of its neighbors) context-free messages, it turns into a *TREE* node, and sends to the k^{th} neighbor a context-free message.
3. *CycleCut* - *CC* (nodes that are cycle cuts) - initially no node is *CC*; after timeout and negotiation, some become *CC*

Definition 3.

– *disconnected subtree*: a maximal set of interconnected *Cycle* nodes, that connect to the rest of the problem only through *CC* or *TREE* nodes (e.g. $X_i - X_{11} - X_{13} - X_j - X_k$ in Figure 2)

– *cyclic subgraph*: a maximal set of *CC* nodes connected pairwise through at least 2 different *CC* nodes, or through a *disconnected subtree*, together with the *Cycle* nodes from the disconnected subtrees connecting them (for example, X_i and all the lower-right box in Figure 2; a counter-example are Subgraph3 and Subgraph2 in 3, which are disjoint, since they are connected only through X_i)
– *context* of a UTIL message: additional information attached to a UTIL message, specifying under which "assumptions" the respective UTIL message is valid (for instance, a context could be ($X_i = v_2/4, X_k = v_4/7$), meaning that the respective UTIL message is valid when X_i takes its second value out of 4 possible values, and X_k takes its 4^{th} value out out 7 possible values). The context can be null (empty), in which case it means that this message is always valid, without any assumptions. Such messages come from the tree parts of the problem. Messages that circulate inside cyclic subgraphs will have non-empty contexts.
– *context union*: the union of one or more contexts is the union of the sets of variables from all the contexts, with their respective assignments. If one or more variable appears in several contexts, then *it has to have the same assignment in all of them*.

4.3 Topological considerations

In order for the *CC* nodes to know how to treat the incoming messages, it is important for them to have some knowledge of the problem structure. This is important, since in a utility-message propagation algorithm, it is possible that multiple messages coming from the same cycle on different paths are actually duplicates, and should be discarded. On the other hand, messages coming from independent subgraphs should always be considered.

For a categorization of the possible neighborhoods an agent X_i might have, please refer to figure 2. Please note that a "*" denotes the possibility of having 0 or more structures of that kind, a "+" denotes at least one, and a "1" denotes exactly one. The hashed nodes are the nodes that are *CC*, and the others are *TREE* or *Cycle* nodes.

The possible neighborhoods of the node X_i can be categorized as follows:

1. *TREE*: this region is a tree rooted at X_i. X_i's neighbor that is the root of the subtree will eventually send a context-free *UTIL* message.
2. *Subgraph_self*: this region is a part of the graph that contains cycles; however, it suffices to remove X_i to break all these cycles. The probes sent by X_i into this region will return *with the same contexts, which only contain X_i as a CC node*. The contexts contain the same set of ids, but not in the same order (depending on the path they took) Node X_i can differentiate between several independent subgraphs of this type by the set of *Cycle* nodes contained in the context.
3. *Subgraph_safe*: this region may contain one or several other CC nodes and several local cycles; however, apart from the link $X_i - X_j$ there is no other path between X_i and this region.
4. *Subgraph_unsafe*: this region may contain one or more other CC nodes and several local cycles; there are multiple paths from X_i and this region (e.g. $X_i - X_{11}$ and $X_i - X_{12}$). What is important to see is that all these paths will eventually connect. This is the general case, and the previous 2 kinds of cycles are special cases of this one; therefore, in the following, we will discuss only about this kind of cycle.

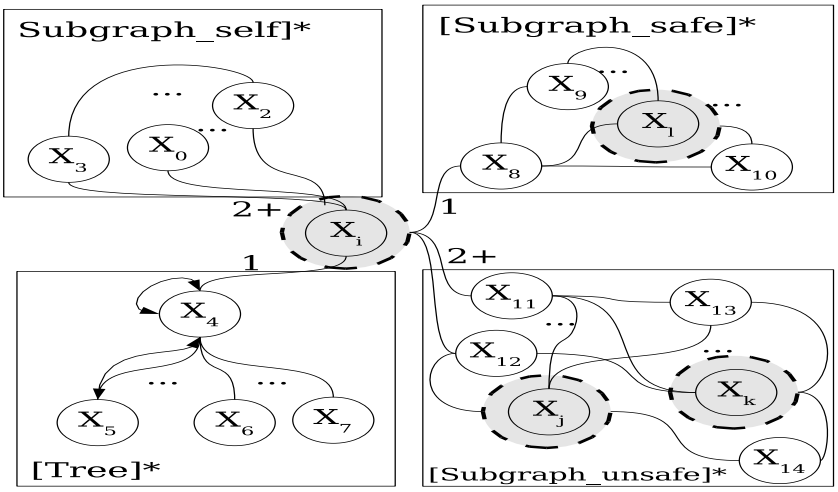


Fig. 2. Categorization of the possible neighborhoods Agent X_i can have, when the underlying constraint problem is a graph.

Topology probing The CC nodes initiate a topology probing process that has as a result the fact that they can categorize their neighboring areas. The probing begins with the *CC* nodes sending out probes to all of their neighbors. Initially the probes have a *context* composed only of the id of the emitting *CC* node. The receiving nodes append their own id to the context of the probes, and then forward them to all their other neighbors. The forwarding stops when reaching *TREE* nodes, or when visiting the same node a second time. For each incoming probe, the *CC* nodes update the largest context that the sending neighbor has sent so far. Upon completion of this procedure (typically after a timeout has been reached), the *CC* nodes sort their neighbors into different sets (cyclic subgraphs) according to their respective largest context; the ones belonging to the same subgraph will necessarily have the same context. They also know their neighborhoods up to the borders of the cyclic subgraphs they are involved in (e.g. in Figure 3, nodes X_i or X_k will know nothing of the Subgraph 1, not even that it exists, since the only contact point between them and any node in Subgraph 1 is the node X_j which will not forward the same probes both ways).

4.4 CyPro - distributed utility probing within a cyclic subgraph

In the most general configuration of a cyclic subgraph, we have a set of *CC* nodes, interconnected through an arbitrary number of disconnected trees (for example, in the lower-right cycle from Figure 2, involving X_i, X_j, and X_k as *CC* nodes, we have 3 disconnected trees: $X_i - X_{12} - X_j - X_k$, $X_i - X_{11} - X_{13} - X_j - X_k$, and $X_j - X_{14} - X_k$). A subgraph like this can be arbitrarily complex. Let us assume for now that there are no links with the outside world (we will relax this condition in section 4.5, and present the complete algorithm)

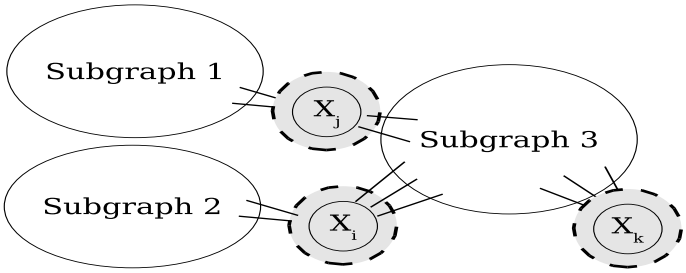


Fig. 3. Problem seen as a meta-tree, composed of cyclic subgraphs connected through *CC* nodes

This algorithm (let us call it *CyPro*) will distributedly generate all the value combinations for all the *CC* nodes involved in this cyclic subgraph, and for each combination, compute the total optimal utility that this assignment yields, provided that the intermediary disconnected trees that lie between the *CC* nodes optimize their values w.r. to this particular assignment of the *CC* nodes. The optimization of the trees is done with a version of *DTREE* extended to support message contexts, therefore the number of messages is linear in the number of arcs of the trees.

During the topology probing phase, each *CC* node received from all its neighbors *TOPO* probes that contained in their context each node in the cyclic subgraph, with the additional *domain size* information for the *CC* nodes involved in this cyclic subgraph. Therefore, each node can easily compute what is the total number of combinations of values required to explore the whole search space: $\prod_{X_i \in CC} |d_i|$. Now, in order to distributedly generate all combinations of values, each node X_i would cycle through all its values for $higher$ times, in each cycle sending out $lower$ probes with the respective value, where

$$higher = max\{1, \prod_{\{X_j \in CC | j > i\}} |d_j|\}, lower = max\{1, \prod_{\{X_j \in CC | j < i\}} |d_j|\}$$

This ensures that all combinations are generated, with the node having the highest id cycling the slowest through its values. *CC* nodes send out their probes to all their neighbors in the subgraph, and wait for replies (they do not forward any messages).

In between the *CC* nodes there are the disconnected trees, composed of *Cycle* nodes that act according to the *k-1 rule*, combining incoming contexts. This ensures that for each value combination that the surrounding *CC* nodes inject in the tree, the results that come out of the tree are optimal with respect to that combination (and contain as context the complete set of *Cycle* nodes from the tree, and the *CC* nodes with their values). Identical results come out from any of the leaves of the tree, so all *CC* nodes connected by that tree have a consistent view of the optimal utility the tree can achieve in that context.

Since the subgraph is arbitrarily complex, it is possible that there is no single node which is connected to all the trees in the subgraph, therefore it is possible that no *CC* node has a global view of the total optimal utility for the current context. In order to overcome this, a ”leader” node is used (it is irrelevant who that leader is, it may be the

node with the highest id in the cycle). Each *CC* node sends the leader a single message that sums up the utilities of the trees that node is involved in, and in which it has the highest id (this ensures that no tree is reported twice). Upon receiving messages from all the *CC* nodes in the subgraph, the leader can sum them up, update its lower bound (thus, it is not needed to store all incoming messages: linear memory requirements), and send back to the other *CC* nodes the result (they can also update their lower bounds, and remember the best local value used in the best context); then a new context is tried, until the last one. At the last context, each *CC* node picks for itself the value that is stored as the best one (from the context that generated the highest utility), and a final round of propagations is initiated, with context-free messages, such that also the *Cycle* nodes within the extended cycle can choose their values.

The algorithm is formally presented in Algorithm 2; informal description:

- If an agent has a single neighbor (even if there are multiple relations to that node), then it labels itself as *TREE*, otherwise as *Cycle*. If X_i is *TREE*, then send the *UTIL* message to its only neighbor.
  The messages passed in this system are the same utility vectors as in *DTREE*, augmented with context information (showing in which context are these vectors valid). If the message is relayed only through *TREE* nodes, then it has an empty context.
- Wait for incoming messages, and respond to them.
- Upon reaching a timeout, X_i realizes it is involved in a cycle, and initiates a negotiation with its neighbors to assume the role of *CycleCut*.
  If the negotiation is successful, X_i becomes *CC*. In the following, the *CC* node will execute two phases: a *topology probing phase*, and a *utility probing phase*.
  Otherwise, negotiation/timeouts repeat until all cycles are broken (detected by the fact that all nodes receive *UTIL* probes/messages).
  If in the end X_i remains a *Cycle* node, then follow the *k-1 rule*.
- *CC* nodes do the *topology probing* (described in section 4.3) and then the *utility probing* (generate all the value combinations of the *CC* nodes involved in the cyclic subgraph and computing the overall optimal utility for each combination)
- *termination*: *TREE* and *Cycle* nodes terminate when the node has received context-free messages from all its neighbors, and *CycleCut* nodes terminate when all the value combinations of their *CC* peers have been explored

Proposition 3. *CyPro is sound and complete.*

PROOF. Follows from the correctness of DTREE (Proposition 1), the fact that all possible value combinations of the cycle cut nodes are tried (a finite number), and that the results of DTREE applied on the disconnected subtrees are combined correctly (only once) by the subgraph leader. □

Overall, for each context, there is a **linear** number of messages generated: $2 \times number\_of\_arcs + 2 \times (k-1)$, where $number\_of\_arcs$ is the number of links (which is less than or equal with the number of relations) in the subgraph, and k is the number of *CC* nodes.

Alternatively, it is possible to cope without any leader, if the *CC* nodes are more "verbose", and send their results to each other ($2 \times number\_of\_arcs + k \times (k-1)$ messages for each context)

Proposition 4. *CyPro has the following complexity:*
$O((dom^k + 1) \times (2 \times number\_of\_arcs + 2 \times (k-1)))$
where dom=domain size, k=size of the cycle cutset and $arcs\_in\_cycle$= the number of arcs in this subgraph.

PROOF. Follows from the discussion above. □

Algorithm 2: *CyPro: distributed utility probing in a cyclic subgraph.*

1: **CyPro(**$Subgraph^k(X_i)$**)**
2: **for all** possible contexts in $Subgraph^k(X_i)$ **do**
3: send out *UTIL* probes with my corresponding value in that context, to all my neighbors
4: wait for incoming *UTIL* probes from all my neighbors in $Subgraph^k(X_i)$
5: duplicates from the same subtree are discarded
6: **if** leader **then**
7: centralize the partial results from all the *CC* peers in $Subgraph^k(X_i)$, and send the total back; update higher bound for my particular value.
8: **else**
9: send the leader the results from the subtrees that I am directly connected to, and in which I am the *CC* node with the highest ID; wait for the total coming from the leader; update higher bound for this particular value of the leader, and remember my own value if bound was improved.
10: At the end, all *CC* nodes know how much utility the whole $Subgraph^k(X_i)$ would get in an optimal assignment for each one of the leader's values, and which one of their values they would pick in that context.

4.5 CyCOpt - distributed cycle-cutset optimization algorithm

We have seen in the previous section that *CyPro* requires fixed message sizes, linear memory, and its message complexity is exponential in the size of the cycle cutset. *CyPro* reduces the complexity from dom^n (equivalent to a standard backtracking) to dom^k (where n=number of nodes in the problem, and k=number of cycle-cut nodes). In the case that the constraint graph is relatively loose, it is likely that $k \ll n$ (a small number of the nodes in the graph are actually cycle-cuts); this would amount to an exponential complexity reduction.

The obvious application of the previous section is to consider the whole problem as an extended cycle, and solve it in the afore mentioned way.

However, in the following, we explore the possibility of further reducing the complexity of the optimization procedure by breaking the problem in *separate* subgraphs, exploring each of them using *CyPro*, and then combining the partial results using a version of *DTREE* that operates at a meta-level, on subgraphs instead of variables. This approach would have the advantage that at a meta-level, the *DTREE* would be linear in the number of subgraphs, and the overall complexity would be the highest complexity of the composing subgraphs.

Some issues need to be considered however, in order to correctly assemble the partial results of *CyPro* applied to the subgraphs:

- *topology*: subgraphs must be independent, connected through at most one *CC* node. That node would play the role of a relay between subgraphs;
- *synchronization*: it is imperative that the *CyPro* be started in a subgraph only after all but one of the externalities (links with other subgraphs through *CC* nodes) have been solved (this is the equivalent *k-1 rule* for the *meta-TREE*);

The first point is already a by-product of the topology-probing phase; it is certain that each *CC* node knows for sure if two subgraphs are independent or not (assuming that there were a link between them in addition to the node itself, a *TOPO* probe is sure to have gone through that link and have returned to the *CC* node, which would have then marked the two subgraphs as the same).

The second one is a little more difficult; in fact it is needed that inside a subgraph there exist a mechanism that allows all the *CC* nodes involved to announce to the other *CC* nodes that they have finished their external *CyPro*s, and now they dispose of accurate and final information about the utility that the rest of the *meta-TREE* can achieve for each of their values. Note that this is completely equivalent to the *k-1* rule for the standard *DTREE*; the difference is that in the standard *DTREE* there was a local decision (each node was receiving all the k-1 messages itself), whereas now we need to implement a *distributed* mechanism that mimics the same functionality.

We solved this problem with a token mechanism: upon solving all of its externalities, a node throws a token in the subgraph; when k-1 (where k is the number of *CC* nodes involved in the subgraph) tokens are received, *CyPro* can be launched. Note that *CC* nodes that are involved in a single subgraph (like X_j and X_k in Figure 2) throw their tokens in from the beginning, since they have no externalities (they are the equivalent of leaf nodes in *DTREE*)

A good strategy is to elect as subgraph leader the last *CC* node that has not yet thrown the token in the subgraph; after *CyPro* is finished in the subgraph, it would be this node that would throw its token in one of its other subgraphs, and start *CyPro* in there, and so on. This synchronization mechanism has the effect that *CyPro*s are starting to cascade, exactly like the *DTREE* propagation that we explained in Section 3.

In the example of Figure 3, node X_k would immediately throw its token in Subgraph 3, X_i in Subgraph 2 and X_j in Subgraph 3. X_k would not start anything in Subgraph 3 because there is a single token in there. *After* having finished *CyPro* in Subgraph 2, X_i would throw its token in Subgraph 3, would see that 2=3-1 tokens exist, and would start *CyPro* in Subgraph 3, etc.

When the last externality of a subgraph is solved, the responsible *CC* node already has complete information for the whole problem (similar to the case in *DTREE* when the last (k^{th}) message is received). It can immediately choose its value, and inform its *CC* peers in all its subgraphs, which in turn will choose theirs, and so on.

The nodes labeled as *TREE* or *Cycle* will execute just as in Section 4.4, sending/relaying messages by the *k-1 rule*. The difference is made by the *CC* nodes that are involved in several subgraphs, which operate in the afore mentioned way.

Proposition 5. *Algorithm 3 is sound and complete.*

PROOF. Follows from Proposition 1, Proposition 3, and the fact that each individual subgraph is explored only when all but one of its externalities are solved (therefore observing the *k-1* rule for the meta-tree). □

Algorithm 3: *CyCOpt: distributed cycle-cutset optimization algorithm.*

CyCOpt($\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{R}$**)**
Each agent X_i executes:

Initialization(X_i, R_i**)**
same as in *D-TREE*
if $|Ngh(X_i)| == 1$ (i.e. X_i is a *TREE* node) **then**
  mark X_i as *TREE*, and send *UTIL* message to the single neighbor
else
  mark X_i as *Cycle*
activate Message_handler()
activate Timeout_handler()
return

Timeout_handler()
if (! received any message from at least $|Ngh(X_i)| - 1$ neighbors) **then**
  $Cycle(X_i) \leftarrow \{X_j \in Ngh(X_i) | X_j$ did not send any message yet $\}$
  negotiate cycle_cut with $\forall X_j \in Cycle(X_i)$; set is_cycle_cutset accordingly
  if is_cycle_cutset **then**
    do TOPOLOGY PROBING
    do MAIN PHASE
  else
    reactivate Timeout_handler()

Message_handler()
if X_i is *TREE* or *Cycle* **then**
  relay messages according to the *k-1 rule*
  terminate upon receipt of k context-free messages

TOPOLOGY DEEP PROBING
send out *TOPO* probes to neighbors in $Cycle(X_i)$ and wait for their return
probes are forwarded by *CC*/*Cycle* nodes, collecting in their context the set of visited nodes
upon completion, X_i can categorize all its neighbors in the sets $TREE(X_i)$ (containing all the *TREE* neighbors) and $Cycle^k(X_i)$ (containing all the neighbors in the **independent** cycle $Cycle^k(X_i)$)

MAIN PHASE (*CC* nodes)
if $|Cycles(X_i)| == 1$ **then**
  send my token in my only cycle
for all $Cycle^k(X_i)$ **do**
  wait for c-1 tokens in each cycle (c=the number of *CC* nodes in $Cycle^k(X_i)$, then perform CyPro in the cycle
  when $|Cycles(X_i)| - 1$ cycles have been explored, send my token in the last cycle, and then perform CyPro in there as well
at this point, X_i has complete information from all $Cycle^k(X_i)$, and can choose its optimal value
inform the *CC* peers from all $Cycle^k(X_i)$ about the value chosen
perform a last optimization step in each $Cycle^k(X_i)$ with the chosen value *and context-free UTIL probes*, such that all *Cycle* nodes can also choose their values and terminate.
terminate

Proposition 6. *Algorithm 3 has the following complexity:*

$O((dom^k + 1) \times (2 \times number\_of\_arcs + 2 \times (k - 1)))$

where dom=domain size, k=size of the cycle cutset for the largest subgraph, number_of_arcs = the number of arcs in the largest subgraph

PROOF. As explained above, the problem is broken up in disjoint subgraphs, which are connected through *CC* nodes. Between subgraphs, there is no explicit communication (except for the fact that the node that connects them will depose its token at some point in one of them, when all the rest are done). The difficult problems lie within the subgraphs, and the largest subgraph is the one that gives the overall complexity. Within a subgraph, the message complexity is given by the formula for *CyPro*, so the overall complexity is given by the largest complexity of all subgraphs. When the leader has finally finished as well, another round of $arcs\_in\_cycle \times 2$ messages is required, but this is a one-time, linear number of messages. □

5 Experimental evaluation

We have done some preliminary evaluation of the algorithms on randomly generated optimization problems (weighted graph coloring) with increasing number of variables. We recorded the number of exchanged messages and present the resulting curve in Figure 4. As expected, the number of messages increases with the problem size, which in turn influences the size of the cycle cutset. However, the direct correlation is with the cycle cutset, and not with the problem size, leading us to believe that this method is a good candidate for solving large but sparse problems, where the cycle cutset has manageable sizes.

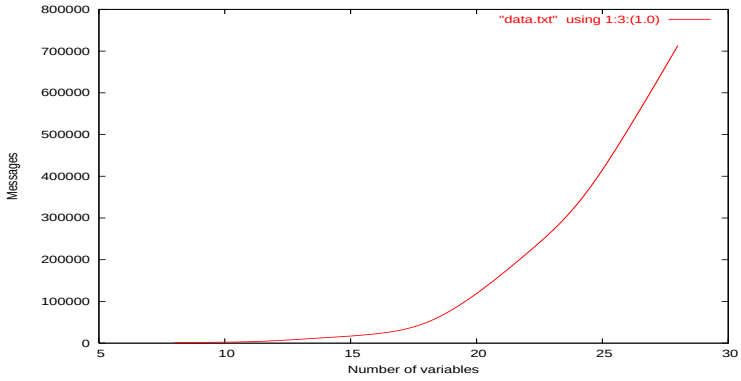


Fig. 4. *Number of messages exchanged while solving problems of increasing size.*

6 Conclusions and future work

We presented in this paper a new complete method for distributed constraint optimization. This method is a utility-propagation method that extends the sum-product algorithm to work on arbitrary topologies using cycle cutsets. It requires fixed message sizes, linear memory, and its message complexity is exponential in the size of the cycle cutset for the largest subgraph in the problem. This method reduces the complexity from dom^n (equivalent to a standard backtracking) to dom^k (*CyPro*) or even $dom^{k'}$ (*CyCOpt*), where n=number of nodes in the problem, k=total number of cycle-cut nodes, and k'=number of cycle-cut nodes in the largest subgraph. For relatively loose problems, it is likely that the inequality $n \gg k \gg k'$ holds, thus our method is likely to produce important complexity reductions.

The algorithm is formulated for optimization problems, but can be easily applied to the satisfaction problem as well.

As future work we consider experimenting with different strategies of selecting the cycle-cut nodes, developing more efficient methods for computation within cyclic subgraphs, and more informed topology probing techniques.

References

1. F. Becker and D. Geiger. Optimization of pearl's method of conditioning and greedy-like approximation algorithms for the vertex feedback set problem. *AI Journal*, 1996.
2. Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
3. Y. Hamadi, C. Bessière, and J. Quinqueton. Backtracking in distributed constraint networks. In *ECAI-98*, pages 219–223, 1998.
4. Arun Jagota and Rina Dechter. Simple distributed algorithms for the cycle cutset problem. In *Proceedings of the 1997 ACM symposium on Applied computing*, pages Pages: 366 – 373, San Jose, California, United States, 1997. ACM, ACM Press New York, NY, USA.
5. Simon Kasif. On the parallel complexity of some constraint satisfaction problems. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-86*, pages 349–353, Philadelphia, PA, 1986.
6. Frank R. Kschischang, Brendan Frey, and Hans Andrea Loeliger. Factor graphs and the sum-product algorithm. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 2001.
7. Amnon Meisels and Roie Zivan. Asynchronous forward-checking on DisCSPs. In *Proceedings of the Distributed Constraint Reasoning Workshop, IJCAI 2003, Acapulco, Mexico*, 2003.
8. P. Modi, W. Shen, M. Tambe, and M. Yokoo. An asynchronous complete method for distributed constraint optimization, 2003.
9. Marius-Calin Silaghi, Djamila Sam-Haroud, and Boi Faltings. Asynchronous search with aggregations. In *AAAI/IAAI*, pages 917–922, Austin, Texas, 2000.
10. Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *International Conference on Distributed Computing Systems*, pages 614–621, 1992.
11. Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem - formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, 1998.
12. Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.

Preprocessing Techniques for Accelerating the DCOP Algorithm ADOPT

Syed Muhammad Ali*, Sven Koenig, and Milind Tambe

USC, CS Department, 941 W 37th Street, Los Angeles, CA 90089-0781, USA
{syedmuha,skoenig,tambe}@usc.edu

Abstract. Distributed Constraint Optimization (DCOP) has emerged as a key technique for distributed reasoning, particularly given the recent progress on complete DCOP algorithms that provide optimal solutions. Yet, their application faces significant hurdles in many multiagent domains due to their inefficiency. Preprocessing techniques have been successfully used to speed up algorithms for centralized constraint satisfaction problems. This paper introduces a framework of very different preprocessing techniques that are based on dynamic programming and speed up ADOPT, an asynchronous complete and optimal DCOP algorithm. We investigate when preprocessing is useful and which factors influence the resulting speedups in two DCOP domains, namely graph coloring and distributed sensor networks. Our experimental results demonstrate that our preprocessing techniques are fast and can speed up ADOPT by more than one order of magnitude.

1 Introduction

Distributed constraint optimization (DCOP) [1, 2] has emerged as a key technique for distributed reasoning in multiagent domains, given its ability to optimize over a set of distributed constraints. For example, DCOP is useful for meeting scheduling in large organizations, where privacy needs make centralized constraint optimization difficult [3]. DCOP is also useful for allocating sensor nodes to targets in sensor networks [4, 1, 5], where the limited communication and computation power of individual sensor nodes makes centralized constraint optimization difficult. Finally, DCOP is useful for coordinating teams of unmanned air vehicles [6], where the need for rapid local responses makes centralized constraint optimization difficult.

Unfortunately, the application of DCOP algorithms faces significant hurdles in many multiagent domains due to their inefficiency. Solving DCOPs optimally is known to be NP-hard, yet one often needs to find optimal DCOP solutions quickly. In this context, researchers have recently developed ADOPT, an asynchronous complete and optimal DCOP algorithm that significantly outperforms competing complete and optimal DCOP algorithms that do not allow partial or complete centralization of value assignments [1].

* This research was partly supported by a subcontract from NASA's Jet Propulsion Laboratory (JPL) and an NSF award under contract IIS-0350584. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations or the U.S. government.

In this paper, we introduce a framework of preprocessing techniques that make ADOPT even more efficient. We focus on ADOPT since it provides an efficient baseline and has been used to solve DCOPs in domains where one needs to find optimal DCOP solutions quickly, namely sensor networks [5] and meeting scheduling for teams of personal assistant agents [3].

Preprocessing techniques have been studied before in the context of CSPs. For example, arc-consistency, path-consistency and general k-consistency algorithms can speed up CSP algorithms dramatically [7]. The key idea behind these preprocessing techniques is to reduce the search space, for example, by eliminating possible values for nodes. Recent work has applied similar preprocessing techniques to both distributed CSPs [8, 9] and centralized COPs [10, 11]. However, preprocessing techniques have not yet been investigated in the context of DCOPs, which is not surprising since efficient complete and optimal DCOP algorithms have been developed only recently. In this paper, we close this gap. Our preprocessing techniques, however, are motivated by heuristic search algorithms rather than preprocessing techniques for CSPs and thus are very different from preprocessing techniques for CSPs. ADOPT is an uninformed search method and our preprocessing techniques speed it up by supplying it with heuristic values that focus its search. Our framework consists of a preprocessing phase followed by the main phase which just runs ADOPT. The preprocessing phase solves a relaxed version of the DCOP to calculate the heuristic values, using either ADOPT itself or specialized preprocessing techniques. We show how one can systematically construct preprocessing techniques of polynomial runtime, some of which are more computation or communication intensive than others and thus tend to calculate more informed heuristic values, thus trading off effort in the preprocessing phase and main phase. We investigate when preprocessing decreases the total effort and which factors influence the resulting speedups in two DCOP domains, namely graph coloring and distributed sensor networks. Our experimental results are very encouraging. For example, our new versions of ADOPT can solve a distributed sensor network problem with 40 nodes about 37 times faster than ADOPT, even with our most pessimistic way of counting cycles, and about 92 times faster if we allow for larger messages.

2 Distributed Constraint Optimization

A DCOP consists of a set of nodes (= agents) N. $D(n)$ denotes the set of possible values of node $n \in N$. $c(d(n), d(n'))$ denotes the cost of a soft binary constraint between nodes $n \in N$ and $n' \in N$ if node n is assigned value $d(n) \in D(n)$ and node n' is assigned value $d(n') \in D(n')$. The objective is to assign a value to every node so that the sum of the costs of the constraints is minimal.

Figure 1 shows an example DCOP with three nodes (A, B and C). All nodes can be assigned either the value x or the value y. There are constraints between A and B, B and C, and A and C. The DCOP has two cost-minimal solutions, namely (A=x, B=y, C=x) and (A=x, B=y, C=y).

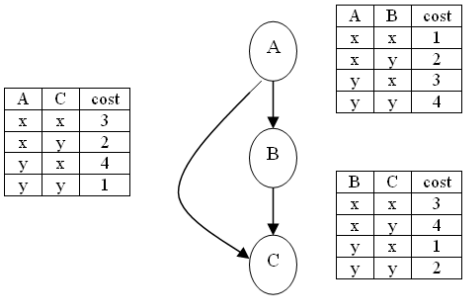


Fig. 1. Example DCOP

3 Distributed Constraint Optimization with ADOPT

ADOPT is an asynchronous complete and optimal DCOP algorithm that significantly outperforms competing complete and optimal DCOP algorithms that do not allow partial or complete centralization or value assignments [1, 12]. It was the first optimal DCOP algorithm that used only localized asynchronous communication and polynomial space for each node. Communication is local in the sense that a node does not send messages to every other node. Rather, ADOPT constructs a constraint tree, which is a tree of nodes with the property that any two nodes that are involved in some constraint are in an ancestor-successor (but not necessarily parent-child) relationship in the tree. For instance, the DCOP in Figure 1 is organized as a tree where A is the root, B is the child of A, and C is the child of B. In this case, the constraint tree is a chain since every node has at most one child. ADOPT searches the constraint tree in a way that resembles uninformed and memory-bounded versions of A*, except that it does so in a distributed way where every node sends messages only to its parent or successors in the constraint tree: Each node asynchronously executes a processing loop in which it waits for incoming messages, processes them and sends outgoing messages. VALUE messages are sent from a node to its children in the constraint tree, informing them of the values of their ancestors. The children then record these values in a "current context." In response to VALUE messages, nodes send COST messages to their parents to provide them with feedback about the costs of the best complete assignment of values to nodes that is consistent with the current context of the node. To this end, a node adds the exact costs of all constraints that involve nodes with known values (= its ancestors) and a lower bound cost estimate of the smallest sum of the costs of all constraints in the subtree rooted at the node (received from its children via COST messages) for its current context. Thus, COST messages contain estimates of the cost of the constraints for the best complete assignment of values to nodes that is both consistent with the current context of the node and a lower bound on the actual cost. Nodes initially use zero as cost estimates, and update these cost estimates when they receive COST messages

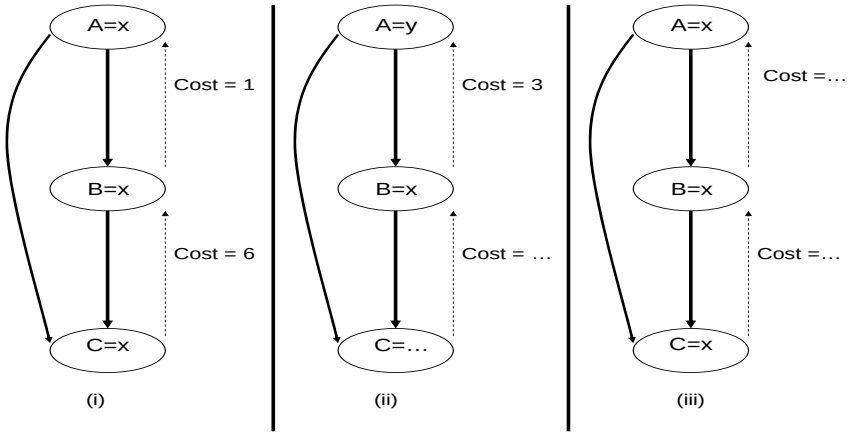


Fig. 2. Snapshots of Possible Execution Trace of ADOPT

from their children. Nodes reset their cost estimates to zero when their current context changes.

Figure 2 illustrates the execution of ADOPT for the DCOP from Figure 1, with an emphasis on aspects that illustrate the benefits of our modifications of ADOPT. The figure shows three snapshots in the progression of a possible execution path of ADOPT. Initially, the cost estimate of choosing value x and the one of choosing value y are zero for every node, and either value can thus be chosen. In Figure 2(i), nodes A, B and C initially each choose value x. Node A now sends VALUE messages to inform its successors B and C about its choice of value x, and node B sends a VALUE message to inform its successor C about its choice of value x, as indicated by the thick arrows. The current context of node B now records that node A has chosen value x, and computes its cost estimate for the best complete assignment of values to nodes that is consistent with node A having chosen value x. This cost estimate is one: If node B chooses value x (value y) then the constraint cost between nodes A and B is one (two, respectively), and the constraint cost between nodes that involve node C is estimated to be zero since node B has not yet received a COST message from node C. Thus, node B sends a COST message to inform node A of an estimated cost of one. The cost estimate of choosing value x is now one for node A while the cost estimate of choosing value y is still zero. In Figure 2(ii), node A now chooses value y (the value with the smallest cost estimate) and sends VALUE messages to inform its successors B and C about its choice of value y. Node B then sends a COST message to inform node A of an estimated cost of three. The cost estimate of choosing value x is now one for node A while the cost estimate of choosing value y is three. Thus, in Figure 2(iii), node A now switches back to value x and thus backtracks in its search space, and the execution of ADOPT continues. ADOPT is described in detail in [12], including some optimizations that are not relevant to this paper and that we did not describe here. Our key point is that node A switched its value from x to y and back to x based on the cost estimates of its values. While such context switching is appropriate to avoid blocking in an asynchronous execution environment, it causes successors to reconstruct their solution, and thus we could potentially improve the performance of ADOPT if we we able to reduce such context switching by supplying it with better cost estimates. For example, if the cost estimate of choosing value y

had been three for node A, then one would have avoided the context switch in Figure 2(ii).

Our new versions of ADOPT are motivated by the need to avoid or reduce such unnecessary context switches. These new versions of ADOPT are identical to ADOPT except that they initialize ADOPT with non-zero cost estimates, called heuristic values. They solve DCOPs optimally if we guarantee that the heuristic values are indeed lower bound cost estimates, which is the case since they use preprocessing techniques that calculate heuristic values by solving a relaxed version of the DCOP (= the DCOP with some constraints deleted) in a preprocessing phase before they run ADOPT in the main phase. The main question of this paper then is whether the total runtime of the new versions of ADOPT is smaller than the one of ADOPT itself. The answer is not obvious since it takes time to compute the heuristic values. It is known that running an uninformed version of A* on a relaxed version of a search problem to obtain heuristic values that are then used to focus the search of an informed version A* on the original version of the search problem cannot result in smaller total runtimes than just using the uninformed version of A* on the original version of the search problem [13]. However, the scheme may potentially work for ADOPT because ADOPT does not resemble A* but memory-bounded versions of A*.

4 Preprocessing Framework

The heuristic values can be calculated by using either ADOPT on a relaxed version of the given DCOP or specialized preprocessing techniques on the given DCOP directly. In the following, we describe three preprocessing techniques (DP0, DP1 and DP2) that trade-off between how long it takes to calculate the heuristic values and how informed they are. We use the following additional notation to describe them formally: $C(n) \in N$ denotes the set of children of node $n \in N$. $A(n)$ denotes the set of those ancestors of node $n \in N$ with which the node has constraints. Finally, the heuristic value $h(d(n))$ is a lower bound cost estimate of the smallest sum of the costs of the constraints between two nodes, at least one of which is a successor of node $n \in N$ in the constraint tree if node n is assigned value $d(n) \in D(n)$.

DP0, DP1 and DP2 are dynamic programming algorithms that assign heuristic values to the nodes, starting at the leaves of the constraint tree and then proceeding from each node to its parent. They set the heuristic values of all leaves to zero, that is, they set $h(d(n)) := 0$ for all $d(n) \in D(n)$ and $n \in N$ with $C(n) = \emptyset$. They calculate the remaining heuristic values $h(d(n))$ for all $d(n) \in D(n)$ and $n \in N$ with $C(n) \neq \emptyset$ as follows, where the minimums in the formulas guarantee that the resulting heuristic values are lower bound cost estimates:

DP0	$h(d(n)) := \sum_{n' \in C(n)} \sum_{n'' \in A(n')} \min_{d(n') \in D(n')} \min_{d(n'') \in D(n'')} c(d(n'), d(n''))$
DP1	$h(d(n)) := \sum_{n' \in C(n)} \min_{d(n') \in D(n')} (h(d(n')) + c(d(n'), d(n)))$
DP2	$h(d(n)) := \sum_{n' \in C(n)} (\min_{d(n') \in D(n')} (h(d(n')) + c(d(n'), d(n)) + \sum_{n'' \in A(n') \setminus \{n\}} \min_{d(n'') \in D(n'')} c(d(n'), d(n''))))$

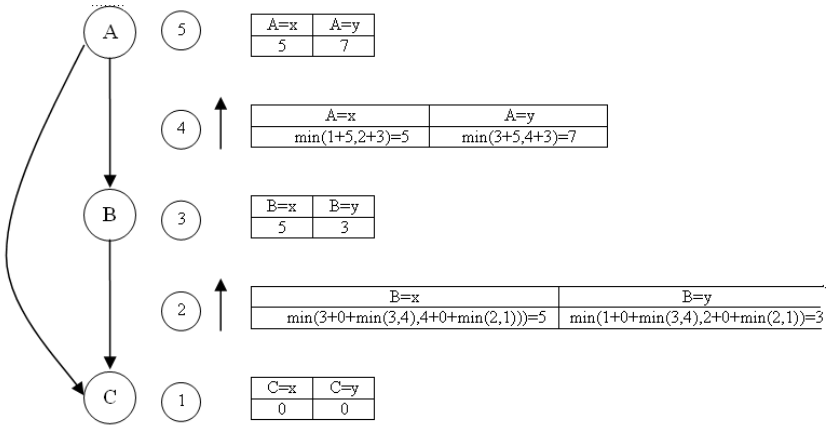


Fig. 3. DP2 Example

It is straightforward to implement DP0, DP1 and DP2 in a decentralized way where nodes send messages to their parents. Basically, the leaves in the constraint tree calculate the heuristic values for each possible value of their parents and then send them in a message to their parents. All other nodes wait until they have received such messages from each of their children, then set the heuristic value of each of their possible values to the sum of the heuristic values reported by their children for this value, and then proceed in the same way as the leaves. For example, Figure 3 describes the operation of DP2 on the DCOP example from Figure 1. In Step 1 of the preprocessing phase, C initializes the heuristic values for its values x and y to 0. In Step 2, C calculates the heuristic values for the values x and y of B. The heuristic value for the value x of B is calculated as follows: If C is assigned the value x then it is cost-minimal to assume that A is assigned the value x. In this case, the cost of the constraint between A and C is 3 and the cost of the constraint between B and C is 3, resulting in an overall cost estimate of 6. On the other hand, if C is assigned the value y then it is cost-minimal to assume that A is assigned the value y as well. In this case, the cost of the constraint between A and C is 1 and the cost of the constraint between B and C is 4, resulting in an overall cost estimate of 5. The heuristic value for the value x of B is the minimum of the two cost estimates and thus 5. Similarly, C calculates the heuristic value for the value y of B. It then sends these heuristic values to B. In Step 3, B updates its heuristic values and, in Step 4, calculates the heuristic values of the values x and y of A. It then sends these heuristic values to A, and finally, in Step 5, A updates its heuristic values, which ends the preprocessing phase. In the main phase, node A initially chooses value x and switches to value y only when the cost estimate of choosing value x exceeds seven (= the initial cost estimate of choosing value y) which avoids the initial context switch in Figure 2(ii).

DP0, DP1 and DP2 can differ in both the heuristic values they calculate and in their computation and communication overhead. Each heuristic value of DP2 is guaranteed to

be at least as large (= at least as informed) as the corresponding heuristic value of either DP0 or DP1. The following table contains the heuristic values for our example, where the last row contains the largest lower-bound cost estimates that satisfy our definition of the heuristic values:

	A=x	A=y	B=x	B=y	C=x	C=y
DP0	1	1	2	2	0	0
DP1	3	5	3	1	0	0
DP2	5	7	5	3	0	0
optimal	6	7	5	3	0	0

We can now examine the overhead of DP0, DP1 and DP2. Unfortunately, it is non-trivial to measure the runtime of the preprocessing techniques since nodes can operate in parallel but are often simulated in different threads on a single-processor machine. We follow other researchers and measure the runtime using cycles, where every node is allowed to process all of its messages in each cycle. However, cycles typically measure only the communication but not the computation overhead. While this is appropriate in those situations where the communication overhead dominates the computation overhead, we also investigate the computation overhead to ensure that it is not excessive.

- **Computation Overhead:** The computation overhead is affected by how many constraint costs a node must access. DP1 needs to access only the costs of the constraints that a node has with its parent while DP0 and DP2 also need to access the costs of the constraints that the node has with its other ancestors.
- **Communication Overhead:** The communication overhead is measured in cycles. DP0 needs only one cycle because it does not propagate heuristic values up the constraint tree while DP1 and DP2 need a number of cycles that equals the depth of the constraint tree (plus one). For example, Steps 1 and 2 constitute one cycle in the DP2 example from Figure 3, Steps 3 and 4 constitute another cycle, and Step 5 constitutes the third and final cycle. Another key difference between DP0 and the other two preprocessing techniques is that DP0 sends only one heuristic value from a node to its parent (because the heuristic values are identical for all possible values of the parent) while DP1 and DP2 send one heuristic value for each possible value of the parent (because they can be different). For example, every node sends two heuristic values to its parent in the DP2 example from Figure 3. As discussed in the section on experimental results, we penalize DP1 and DP2 for their larger messages by increasing their cycle count by a factor that equals the number of heuristic values they send per message (which simulates them only being able to send a single value per message).

Based on these two axes of computation and communication overhead, we identify two key design choices. They provide the rationale for our choices of DP0, DP1 and DP2. In the following, we always list the choice first that results in more informed heuristic values.

- **Property a (= Computation Overhead):** A preprocessing technique can either take all constraints into account (1) or only the constraints between nodes and their

parents (2), in which case the constraints form a tree. (2) corresponds to relaxing the DCOP by deleting all constraints that are between any two nodes that are not in a parent-child relationship in the constraint tree, which is basically exactly what DP1 does. Instead of using DP1 on a given DCOP, one can therefore also use ADOPT itself on the relaxed DCOP to calculate the same heuristic values, which needs more cycles than DP1 but makes the preprocessing easier to implement and might still result in substantial speedups. (We also experimented with other ways of deleting constraints. For example, randomly deleting a given percentage of constraints turned out not to be advantageous.)

– **Property b (= Communication Overhead):** A preprocessing technique can either take the heuristic values of a node into account (1) or ignore them (2) when calculating the heuristic values of the parent. (1) needs a number of cycles that equals the depth of the constraint tree (plus one) to propagate the heuristic values up the constraint tree, while (2) can be computed in only one cycle.

The following table categorizes DP0, DP1 and DP2 according to these two properties:

	Property a	Property b
DP0	(1)	(2)
DP1	(2)	(1)
DP2	(1)	(1)

The following table shows the runtimes of DP0, DP1, and DP2 per cycle as a function of the two properties, where $v = \max_{n \in N} |D(n)|$ is the largest cardinality of the set of possible values of any node, $k = \max_{n \in N} |A(n)|$ is the largest cardinality of the set of those ancestors with which any node has constraints, c denotes the runtime of the preprocessing technique measured in cycles, and m denotes the size of its messages:

	Preprocessing Cost per Cycle		
		low (c=1, m=1)	high (c=tree depth, m=v)
Graph Structure	tree	O(v^2)	DP1 O(v^2)
	full graph	DP0 O(kv^2)	DP2 O(kv^2)

There are v^2 constraint costs for each constraint. Each preprocessing technique might have to process all v^2 constraint costs for each of the at most k ancestors of a node with which it has constraints. If the constraints form a tree (upper row of the table), then the number of ancestors is one (k=1). When the constraints do not form a tree (lower row of the table), each node must examine its input and thus v^2 constraint costs for each of its k ancestors, and thus the kv^2 cost is mandatory for both DP0 and DP2. The cost for DP2 needs further explanation since given a node n, it iterates over all the k ancestors of all the children of n, and would thus appear to require an additional cost of iterating over all such children. However, in a decentralized implementation, each child node only computes the heuristic values relevant to itself and sends the values to the parent node n, which sums the inputs from the children. Thus, each child incurs the cost of kv^2 per cycle. This explains the table. Since the runtimes of DP0, DP1, and DP2 are polynomial per cycle and their number of cycles is polynomial as well, their runtimes

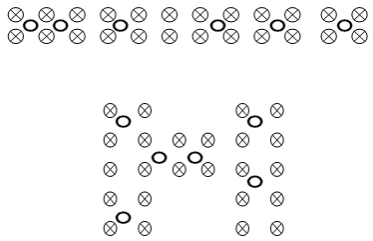


Fig. 4. Sensor Network Topologies: Chain (top) and H Configuration (bottom)

are polynomial. This means that their runtimes are small in the worst case compared to the runtime of the main phase since solving DCOPs is NP-hard.

5 Experimental Results

It is not immediately obvious whether the runtime of the preprocessing techniques is sufficiently overcome by the speedups achieved in the main phase and, if so, which preprocessing technique results in the smallest total runtime, that is, sum of the runtimes of the preprocessing phase and main phase. We conducted experiments in two different DCOP domains to answer these questions:

- **Graph Coloring:** Our first domain is a three-coloring problem with a link density (= number of constraints over the number of nodes) of two. The values of the nodes correspond to the colors, and all constraint costs are drawn with uniform probability from the integers between 1 and 100.
- **Distributed Sensor Network (DSN):** Our second domain is a distributed sensor network problem where 24 sensors, arranged in either a chain or an H configuration, have to track a given number of targets that are randomly positioned between four sensors each [14]. Figure 4 shows examples of the two sensor configurations, where circles with Xs represent sensors and circles without Xs represent targets. Each sensor can track at most one target, which needs to be in its immediate vicinity. Each target is either tracked by exactly three sensors or one incurs a cost that is drawn with uniform probability from the integers between 0 and 100. The mapping from this DSN domain to a DCOP is described in detail in [14, 12]. Basically, one creates the nodes TA1, TB1, TC1, and TD1 if the sensors A, B, C, and D are able to track target 1. Thus, there is one node for each combination of a sensor and one of its possible targets. The possible values of these three nodes are all combinations of three sensors that are able to track the target (ABC, ABD, ACD, BCD), the value IGNORE that represents that no sensor will track the target, and the value ABSENT that represents that the target disappeared and thus no longer needs to get tracked. There are equality constraints between any two nodes with the same target. For example, there is an equility constraint between TA1 and TB1

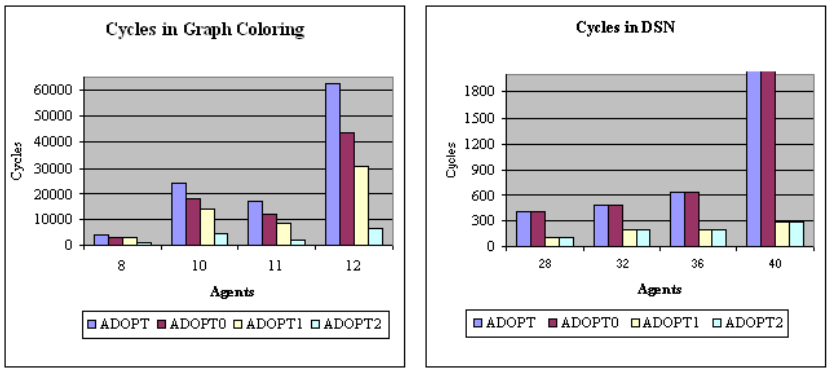


Fig. 5. Cycles in Graph Coloring (left) and DSN (right)

that requires sensors A and B to agree on the set of sensors that track target 1. Similarly, there are mutual exclusion constraints between any two nodes with the same sensor. For example, there is a mutual exclusion constraint that enforces that sensor A cannot track targets 1 and 2 at the same time. The costs are zero if the constraints are satisfied and very high if the constraints are not satisfied, making them hard constraints. If a node is assigned the value IGNORE, then it incurs a cost for ignoring that target.

The following table gives details on the number of nodes and the number of their possible values for the two DCOP domains. We varied the sizes of the domains by varying the number of their nodes. We report averages over 15 problem instances for each domain and size:

Domain	Number of Nodes	Number of Values per Node
Graph Coloring	8, 10, 11, 12	3
DSN	28, 32, 36, 40	6

5.1 Discussion of Cycle Count

In the following, we refer to ADOPT0, ADOPT1 and ADOPT2 as the combination of DP0, DP1 and DP2, respectively, in the preprocessing phase and ADOPT in the main phase. Figure 5 shows the total number of cycles of ADOPT and the three new versions of ADOPT as a function of the number of nodes (= agents). Remember that, whenever we report cycles, we penalize ADOPT1 and ADOPT2 for their larger messages in the preprocessing phase by increasing their cycle count in the preprocessing phase by a factor that equals the number of heuristic values they send per message, namely 3 in graph coloring and 6 in DSN. ADOPT2 outperforms all other versions of ADOPT in graph coloring and its speedups increase with the size of the domain. For example, ADOPT2 speeds up ADOPT by a factor of 9.8 in graph coloring with 12 nodes. ADOPT0 does not

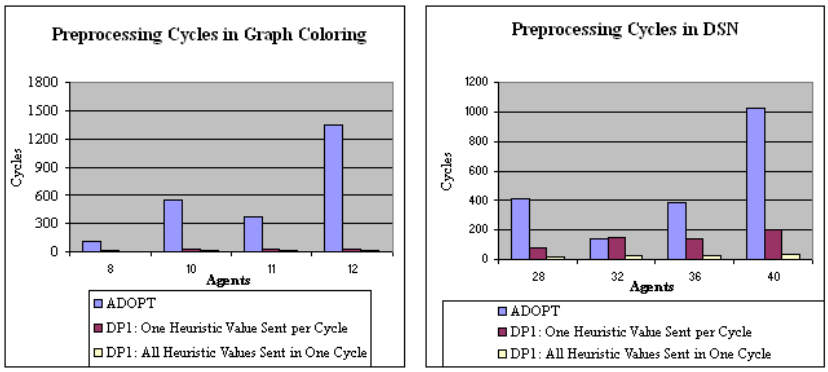


Fig. 6. Preprocessing Cycles in Graph Coloring (left) and DSN (right)

speed up ADOPT in DSN and ADOPT1 and ADOPT2 speed it up by the same amount. The bars for ADOPT and ADOPT0 in DSN with 40 nodes have been shortened in the figure since their number of cycles is 10,694, and ADOPT2 speeds up ADOPT by a factor of 37.6 in this case. It turns out that ADOPT2 even speeds up ADOPT by a factor for 92.5 if we do not penalize it for its larger messages. To summarize, ADOPT2 has the smallest number of total cycles in graph coloring. Both ADOPT1 and ADOPT2 have the smallest number of total cycles in DSN, which means that ADOPT1 should be preferred over ADOPT2 in this domain since the computation overhead of DP1 is smaller than the one of DP2. On the other hand, ADOPT0 is not the method of choice in either domain despite the small computation and communication overhead of DP0 over DP1 and DP2.

Remember that one can use both DP1 on a given DCOP or ADOPT on a relaxed version of the DCOP to calculate the same heuristic values in the preprocessing phase. Thus, the overhead in the main phase will be identical in both cases and one should choose the preprocessing technique that results in the smallest number of cycles in the preprocessing phase. Figure 6 shows that the number of cycles of DP1 in the preprocessing phase is smaller than the one of ADOPT by a factor of 52.5 in graph coloring with 12 nodes and by a factor of 5.1 in DSN with 40 nodes. Its number of cycles in the preprocessing phase would even be smaller than the one of ADOPT by a factor of 157.4 in graph coloring with 12 nodes and by a factor of 30.5 in DSN with 40 nodes if we did not penalize DP1 for its larger messages by increasing its cycle count. To summarize, there is an advantage to using specialized preprocessing techniques in the preprocessing phase rather than the more general ADOPT itself.

5.2 Discussion of Accuracy

To understand better why the speedups depend on the preprocessing technique, remember that the heuristic values computed by the preprocessing techniques are used to seed the cost estimates of ADOPT in the main phase. ADOPT can raise these cost estimates

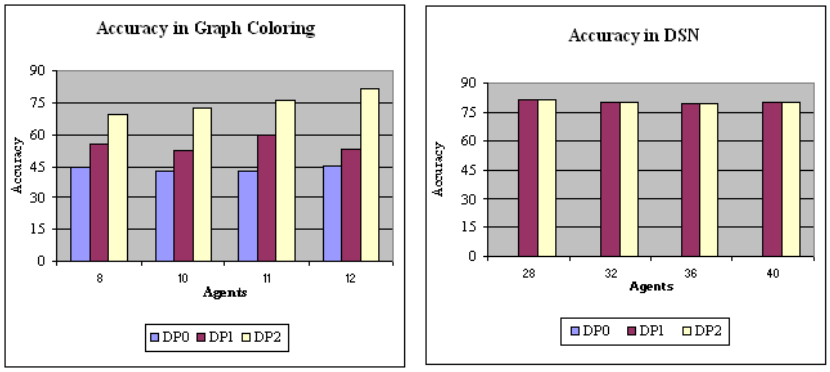


Fig. 7. Accuracy in Graph Coloring (left) and DSN (right)

during its operation. We therefore computed the average ratio of the cost estimates computed by the preprocessing techniques and the cost estimates after the termination of ADOPT. We refer to this ratio as the accuracy. The larger the accuracy, the more informed the heuristic values are. An accuracy of 0 percent means that the heuristic values are no more informed than the initial cost estimates of ADOPT itself. In this case, the preprocessing technique does not speed up ADOPT. On the other hand, an accuracy of 100 percent means that the heuristic values computed by the preprocessing technique were so good that ADOPT was not able to raise them. Figure 7 shows the accuracies of DP0, DP1 and DP2. The accuracy of DP0 is 45.1 percent, the accuracy of DP1 is 53.4 percent, and the accuracy of DP2 is 81.6 percent in graph coloring with 12 nodes. On the other hand, the accuracy of DP0 is zero percent (and hence the bar does not appear in the figure) in DSN with 40 nodes since the heuristic values calculated by DP0 are all zero. This is so since every constraint has at least one constraint cost that is zero. Thus, ADOPT and ADOPT0 are equally fast in this case. The accuracies of DP1 and DP2 are larger than zero percent but, for a similar reason, identical at 80.0 percent. Thus, ADOPT1 and ADOPT2 are equally fast in this case. Figure 7 shows that the number of cycles from Figure 5 are closely correlated with the accuracies of the heuristic values. The more accurate the heuristic values, the more they speed up ADOPT.

To examine this relationship further in DSN with 36 nodes, we first ran ADOPT without preprocessing and obtained the cost estimates after its termination. We then ran ADOPT again but now simulated a preprocessing phase that produces heuristic values that are equal to the product of the corresponding cost estimates after the termination of ADOPT and the same constant factor (smaller than one), which represents the desired accuracy of the heuristic values. Figure 10 (left) shows that the total number of cycles is closely correlated with the factors. Similar to the previous experiment, the larger the factors and thus the more accurate the heuristic values, the more they speed up ADOPT.

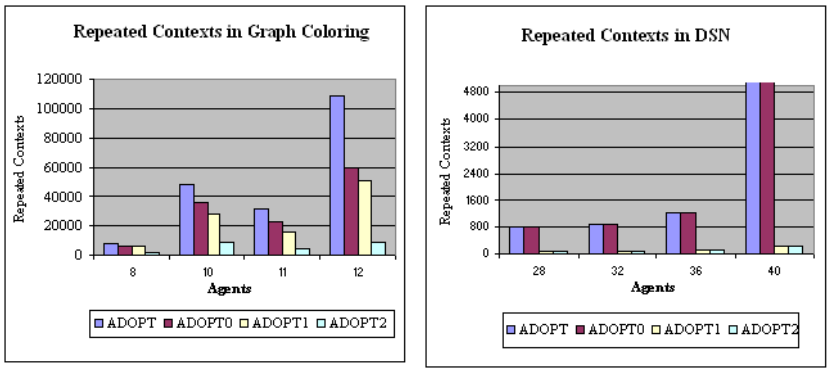


Fig. 8. Repeated Contexts in Graph Coloring (left) and DSN (right)

5.3 Discussion of Repeated and Unique Contexts

There are two reasons why the informedness of the heuristic values can have a large effect on the resulting speedups. We explore both reasons, illustrating that the speedups are a combination of both reasons:

- The first reason why the informedness of the heuristic values can have a large effect on the resulting speedups is that ADOPT, as a memory-bounded DCOP algorithm, has to regenerate partial solutions (in the form of current contexts) when it backtracks to a previously explored part of the search space. More informed heuristic values reduce the amount of backtracking of ADOPT and thus the number of regenerated (= repeated) current contexts, resulting in a smaller number of cycles in the main phase. To verify our hypothesis, we measured the average number of regenerated current contexts at each node. Figure 8 shows that the number of regenerated current contexts is indeed closely correlated with the number of cycles from Figure 5 and the accuracies from Figure 7. The more accurate the heuristic values, the fewer current contexts are repeated in the main phase, and the more the heuristic values speed up ADOPT. The bars for ADOPT and ADOPT0 in DSN with 40 nodes have been shortened in the figure since their number of repeated current contexts is 22,610, and ADOPT2 speeds up ADOPT by a factor of 37.6 in this case.
- The second reason why the informedness of the heuristic values can have a large effect on the resulting speedups is that more informed heuristic values reduce the part of the search space explored by ADOPT and thus the number of unique (= different) current contexts, resulting in a smaller number of cycles in the main phase. To verify our hypothesis, we measured the average number of unique current contexts at each node. Figure 9 shows that the number of unique current contexts, surprisingly, changes very little in graph coloring and can even increase with the accuracy of the heuristic values. The number of unique current contexts decreases with the accuracy of the heuristic values in DSN. The more accurate the heuristic values in this case, the fewer unique current contexts are generated in the main phase, and

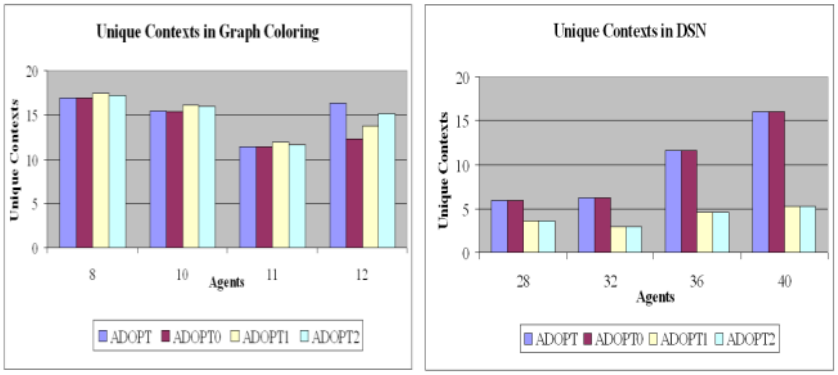


Fig. 9. Unique Contexts in Graph Coloring (left) and DSN (right)

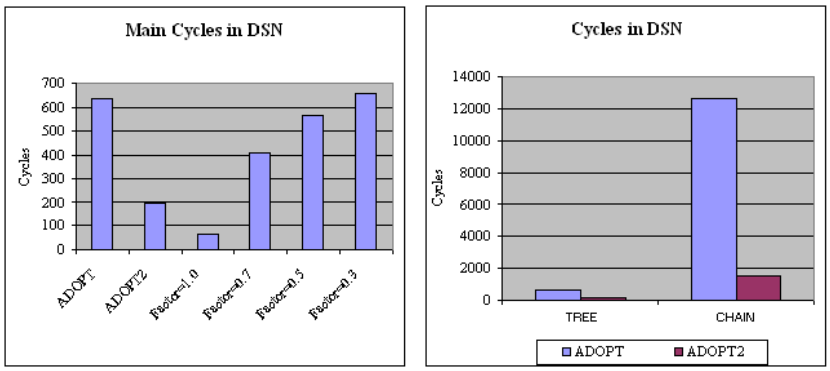


Fig. 10. Impact of Accuracy (left) and Constraint Topology (right) on Cycles in DSN

the more the heuristic values speed up ADOPT. This difference contributes to the speedups tending to be higher in DSN than graph coloring.

5.4 Discussion of Constraint Tree Topologies

We also tested the impact of the topology of the constraint tree on the number of cycles in DSN with 36 nodes. Our initial hypothesis was that the speedups would be substantially larger for chains than for trees. Figure 10 (right) shows, however, that the speedup of ADOPT2 over ADOPT is about the same in either case.

6 Conclusions

In this paper, we developed a framework of preprocessing techniques that speed up ADOPT, an asynchronous complete and optimal DCOP algorithm. Our preprocessing

techniques use dynamic programming to calculate informed lower bound cost estimates for ADOPT. Our empirical results in two DCOP domains, namely graph coloring and distributed sensor networks, demonstrated that our preprocessing techniques are fast and can speed up ADOPT by more than one order of magnitude, at a relatively low preprocessing cost. We showed that the key reason for the speedup is the informedness of the heuristic values, which in turn determines how many partial solutions ADOPT generates and how many of these it revisits. The results also demonstrated that the preprocessing techniques are significantly more efficient than using ADOPT itself in the preprocessing phase. As outlined in [1], it is essential to use lower bound cost estimates in DCOP algorithms. Since our preprocessing techniques focus on computing such lower bound cost estimates, the ideas behind them might also apply to DCOP algorithms other than ADOPT. It is future work to explore their applicability to other DCOP algorithms as well as to develop even more sophisticated preprocessing techniques.

References

1. Modi, P., Shen, W., Tambe, M., Yokoo, M.: An asynchronous complete method for distributed constraint optimization. In: AAMAS. (2003) 161–168
2. Mailler, R., Lesser, V.: Solving distributed constraint optimization problems using cooperative mediation. In: AAMAS. (2004) 438–445
3. Maheswaran, R., Tambe, M., Bowring, E., Pearce, J., Varakantham, P.: Taking DCOP to the real world: Efficient complete solutions for distributed event scheduling. In: AAMAS. (2004) 310–317
4. Lesser, V., Ortiz, C., Tambe, M., eds.: Distributed sensor networks: A multiagent perspective. Kluwer (2003)
5. Scerri, P., Modi, J., Tambe, M., Shen, W.: Are multiagent algorithms relevant for real hardware? A case study of distributed constraint algorithms. In: ACM Symposium on Applied Computing. (2003) 38–44
6. Schurr, N., Okamoto, S., Maheswaran, R., Scerri, P., Tambe, M.: Evolution of a teamwork model. In Sun, R., ed.: Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation. Cambridge University Press (2004) (to appear)
7. Dechter, R., Meiri, I.: Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In: IJCAI. (1989) 271–277
8. Jung, H., Tambe, M.: Performance models for large scale multiagent systems using POMDP building blocks. In: AAMAS. (2003) 297–304
9. Silaghi, M., Sam-Haroud, D., Faltings, B.: Consistency maintenance for ABT. In: CP. (2001) 271–285
10. Bistarelli, S., Gennari, R., Rossi, F.: Constraint propagation for soft constraints: generalization and termination conditions. In: CP. (2000) 83–97
11. Schiex, T.: Arc consistency for soft constraints. In: CP. (2000) 411–424
12. Modi, P., Shen, W., Tambe, M., Yokoo, M.: ADOPT: asynchronous distributed constraint optimization with quality guarantees. Artificial Intelligence (2004) (to appear)
13. Hansson, O., Mayer, A., Valtorta, M.: A new result on the complexity of heuristic estimates for the A* algorithm. Artificial Intelligence **55** (1992) 129–143
14. Modi, P., Jung, H., Tambe, M., Shen, W., Kulkarni, S.: A dynamic distributed constraint satisfaction approach to resource allocation. In: CP. (2001) 685–700

Dynamic Distributed BackJumping

Viet Nguyen[1], Djamila Sam-Haroud[2], and Boi Faltings[2]

[1] Laboratory of Autonomous Systems
[2] Laboratory of Artificial Intelligence
Swiss Federal Institute of Technology
{viet.nguyen,jamila.sam,boi.faltings}@epfl.ch

Abstract. We consider Distributed Constraint Satisfaction Problems (DisCSP) when control of variables and constraints is distributed among a set of agents. This paper presents a distributed version of the centralized BackJumping algorithm, called the *Dynamic Distributed BackJumping - DDBJ* algorithm. The advantage is twofold: *DDBJ* inherits the strength of synchronous algorithms that enables it to easily combine with a powerful dynamic ordering of variables and values, and still it maintains some level of autonomy for the agents. Experimental results show that *DDBJ* outperforms the *DiDB* and *AFC* algorithms by a factor of *one to two* orders of magnitude on hard instances of randomly generated DisCSPs.

Keywords: Search, Constraint Satisfaction, Distributed Systems, Multi-Agent Systems.

1 Introduction

Constraint Satisfaction has been used as a powerful paradigm for general problem solving. It consists of finding values for problem variables in some particular domains subject to constraints that specify possible consistent combinations. Solving a CSP is to find a set of variable assignments that satisfies all the constraints.

A distributed CSP (DisCSP) is a CSP when variables and constraints are distributed among a network of automated agents. Each agent may hold one or more variables which are connected by local constraints, and also connected by inter-constraints to variables of other agents. Many application problems in Multi-Agent Systems (MAS) can be formulated and solved using a DisCSP framework ([1]), such as distributed resource allocation problems, distributed scheduling problems or multi-agent truth maintenance tasks.

In solving DisCSPs, agents exchange messages about the variable assignments and conflicts of constraints. Several distributed search algorithms have been proposed for solving DisCSPs. They can be divided into two main groups: asynchronous and synchronous algorithms. The former are algorithms in which the process of assigning variable values and exchanging messages is performed asynchronously between the agents, whereas in the latter group, agents assign values to variables in a synchronous, sequential way. Each group has different strengths and drawbacks. We discuss some of them in the next section.

2 Related Work

One of the pioneer algorithms is the *Asynchronous BackTracking - ABT* algorithm ([2], [3]). It is a distributed, asynchronous version of a generic backtracking algorithm. Agents communicate by two types of messages: `OK?` messages to distribute the current value, and `Nogood` messages to declare new constraints. The simplicity and computational concurrency are its strengths. *ABT* needs polynomial space for storing nogoods to be complete ([2]). The algorithm requires the assumption that messages are received in the order in which they were sent for completeness, otherwise all nogoods have to be stored and it would suffer from exponential space complexity. One way to work around is to attach a sequence number for each message, so the order of messages can be determined at the receiving end.

A later version of *ABT* which makes use of dynamic ordering of agents, called the *Asynchronous Weak-Commitment Search - AWC*, is given in [3]. This algorithm is shown to be faster than *ABT*, but the main drawback is that it requires exponential space for completeness.

The *Distributed Dynamic Backtracking - DiDB* algorithm is another distributed, asynchronous algorithm which is inspired by its centralized version *Dynamic Backtracking* ([4]), presented in [5], [6]. Briefly, the algorithm transforms the constraint network into a directed acyclic graph and performs dynamic jumps over the set of conflicting agents. Again, this algorithm requires the assumption that messages are received in the order in which they were sent and polynomial space for nogood stores. However, the main weakness is the problem of message duplication. Due to asynchrony, an agent may keep asking values of its parents, and the parents keep sending reply messages. This process propagates down the whole graph, creates many duplicated messages. Experimental results show that the number of messages increases dramatically and soon consumes all resources. Some duplication prevention mechanism can be added, but great attention must be paid for not loosing solutions (A simple way which consists in sending a given message only once does not work!).

Another distributed asynchronous algorithm is given lately in [7], the *Asynchronous Aggregation Search - AAS*. This algorithm works in a similar way as *ABT*, except that consistent values of the partial solution are also included in `OK` messages. This mechanism helps in reducing number of backtracks. For problems with large variable domains, including consistent values produces long messages. Thus, *AAS* is more practical for problems with small variable domains.

A recently proposed algorithm, called the *Asynchronous Forward Checking - AFC* ([8]), belongs to the group of distributed synchronous algorithms. It is a generic backtracking algorithm combined with a look ahead heuristic by means of asynchronous forward checking messages. Agents assign their values for variables sequentially by having one current partial assignment shared among all agents. When a dead end is detected, the algorithm backtracks sequentially following the reverse ordering. A strength of this algorithm is in its algorithmic simplicity and good computational efficiency, inherited from centralized algorithms. It has been shown to provide better performance, in terms of number of messages and constraint checks, than asynchronous algorithms *ABT* and *DiDB* ([8]). The main drawback of *AFC* is that it does not exploit concurrency: at any time, there is only either one `AFC` or one `BT` message that is exchanged between

the agents, results in long running time (running cycles) compared to asynchronous algorithms.

3 Preliminaries

Constraint Satisfaction

Classically, Constraint Satisfaction Problems (CSP) have been defined for problems in centralized architectures. A finite CSP is defined by a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where

- $\mathcal{X} = \{x_1, ..., x_n\}$ is the set of n variables.
- $\mathcal{D} = \{D_1, ..., D_n\}$ is the set of n finite, discrete domains of variables $x_1, ..., x_n$, respectively.
- $\mathcal{C} = \{C_1, ..., C_k\}$ is the set of k constraints on the variables. These constraints give the allowed values that the variables can simultaneously take. $\mathsf{var}(C_i)$ is the set of variables that are constrained by C_i.

A *solution* to a CSP is an assignment of values taken from the domains to all variables such that all the constraints are satisfied. Constraint satisfaction is NP-complete in general, and it is typically solved by a tree-search procedure with backtracking.

Distributed Constraint Satisfaction

A distributed CSP (DisCSP) is a CSP in which the variables and constraints are distributed among a network of automated agents. Formally, a finite DisCSP is defined by a 5-tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \phi)$, where $\mathcal{X}$, $\mathcal{D}$ and $\mathcal{C}$ are the same as in centralized CSP, and

- $\mathcal{A} = \{A_1, ..., A_p\}$ is the set of p agents
- $\phi : \mathcal{X} \rightarrow \mathcal{A}$ is a function that maps variables to agents

Solving a DisCSP is to find an assignment of values to variables by the collective and coordinated action of automated agents. A *solution* to a DisCSP is a compound assignment of values to all variables such that all constraints are satisfied.

In DisCSP, agents communicate with each other by sending messages. We make the following assumptions for the communication model similar to those proposed in [3]:

1. An agent can send messages to other agents iff the agent knows the addresses of the agents.
2. The delay in delivering a message is finite but random; there is no message lost.

The second assumption has been partially relaxed from the original one in [3] that also assumes that messages are received in the order in which they were sent. Some algorithms (*ABT*, *DiDB*) require this assumption to be complete. Furthermore, for simplicity and without loss of generality, we assume that:

1. ϕ is a one-to-one function; it means that each agent holds only one variable; and there are no intra-agent constraints.
2. $\mathcal{C}$ are binary constraints so that $\mathsf{var}(C_i) = 2$, and every constraint is known by both agents involved in the constraint.

By these assumptions, the constraint network is simplified to a constraint graph where agents represent graph nodes and constraints represent graph edges.

4 The Algorithm DDBJ

The *Dynamic Distributed BackJumping - DDBJ*, is a complete, distributed, semi-asynchronous version of a graph-based backjumping algorithm which was previously introduced in centralized CSP ([9]). The algorithm combines the concurrency of an asynchronous dynamic backjumping algorithm and the computational efficiency of the synchronous *AFC* algorithm ([8]), coupled with the heuristics of dynamic value and variable ordering.

The Distributed BackJumping procedure

Agents perform value assignments in two phases:

- *Advancing forward* phase: which occurs when a new assignment tuple is added to the current partial solution.
- *Backjumping (backward)* phase: which occurs when an agent encounters a conflict. The process is "jumped back" to the culprit agent.

An agent is either in a *forward* phase or a *backward* phase. Algorithmically, the *forward* phase is performed sequentially: the assigning agent sends an `OK` to the next agent and `FC` messages to unassigned connected agents (similarly to *AFC* algorithm). If an agent detects a conflict when receiving some `OK`/`FC` message, it performs the *backward* phase asynchronously to backjump to the culprit agent, and also sends `NG` messages to unassigned agents. At any time, there can be several culprit agents detected and thus several backjumps are performed simultaneously. The culprit agents will change their values, hence the current partial solution (CPS), and perform the *forward* phase, without synchronizing with other agents nor waiting for other agents to switch phases. Consequently, at any time, agents are performing the *forward* and *backward* phases simultaneously in parallel without any synchronous control.

An example of algorithm execution is illustrated in Fig.1. At time *t1*, agent `A3` sends one `OK` message to `A4` (solid lines) and `FC` messages to connected agents (dotted lines). At a later time *t2*, `A11` finds a conflict and backjumps to `A3` by a `BT` message (dashed lines) and sends `NG` messages to others (not shown). At the same time, `A3`'s assignment has already propagated down to `A6` and `A7`, and get backjumped at `A6` to `A4` and backtracked at `A7` to `A5`. However, the asynchronous executions at `A6` and `A7` and the consequent ones will soon be overwritten by the new assignment at `A3`. These execution flows are carried out simultaneously.

In *AFC*, backtracking is performed sequentially (or synchronously) from the detecting agent to the culprit. At any time, there is only either one `OK` or one `BT` message being sent. In *DDBJ*, any agent who receives an `OK` or `FC` message can initiate a backjump. Thus, there can be several `OK` and `BT` messages exchanged simultaneously, generating multiple asynchronous execution threads. However, there is only one `OK` message which may potentially lead to a solution (the most updated one or equivalently the one on the highest level of the search tree). The other `OK` messages will continue to propagate and create the assignment chains down the search tree, until only when the `NG` or newer messages arrive. Usually, it takes some cycles to stop these obsolete processes,

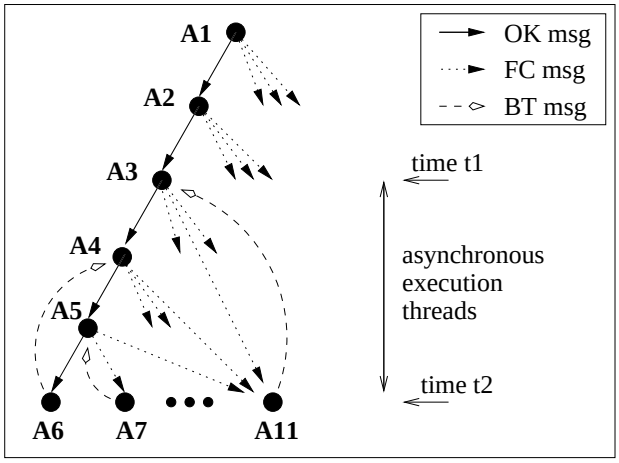


Fig. 1. An example of the *DDBJ* execution

depending on the size of the network, the connectivity density, the message delivering delay, etc.

The *DDBJ* algorithm is executed on every agent. Each maintains current value assignments of other agents in an $AgentView$ ([2]). We also adopt the $AgentView.consistent$ from [5] to represent whether the CPS it holds is consistent. To determine which OK message is the most updated one and to discard obsolete messages, we introduce for each agent a time flag called $TimeStamp$ which is incremented by 1 when the agent changes its value. When sending OK/FC messages, an agent includes its $TimeStamp$ with its assignment. The receiving agent checks the attached $TimeStamp$s and updates its context only if the message is valid. In the example above, by the $TimeStamps$, A4's new assignment (due to A6's backjump) will overwrite executions from A5 (due to A7's backtrack); however the new A3's assignment (due to A11's backjump) will eventually overwrite all executions below it.

The Dynamic Value and Variable Ordering Heuristics

The *DDBJ* algorithm uses dynamic value and variable ordering heuristics. Each agent keeps a potential conflict counter list of its domain values, and a potential conflict counter list of other agents. An agent chooses the value which has the lowest counter value to assigns its variable, and sends the OK message (which contains the partial solution) to the agent which has the highest counter value (and FC messages to other linked agents). If there is a tie, the agent can use the chronological order. At start, all the counter values are equally zeros.

When a dead end is detected by an agent, the dead end discovering (DED) agent performs updating its priority lists in two steps. In the first step, it decreases the counter of the culprit agent (the agent whose value causes the dead end), then it sends the BT message to the culprit agent. The culprit agent, upon receiving the BT message, increases the counter of the sender (the DED agent) and increases the counter of its

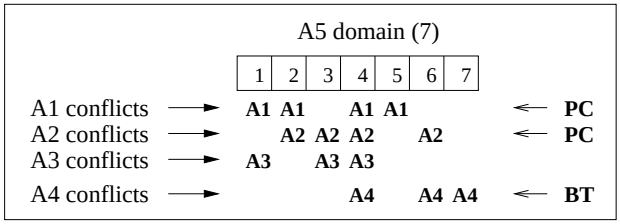


Fig. 2. An example of the heuristics: Agent A5 comes to a dead end, sends a BT message to culprit agent A4, sends "potential conflict" - PC messages to A1, A2

value that causes the backtrack, then it follows the backjumping procedure. In second step, the DED agent determines its "potential conflicting agents" (PC agents). A PC agent is the `first` agent whose value conflicts with a value in the domain of the DED agent. The DED agent increases the counters of the PC agents, sends a "potential conflict" - PC message to the PC agents. The PC agents, after receiving the PC message, increase the counters of their values (that cause the dead end), increase the counter of the DED agent. The idea here is to give more priority to the agents at higher top level of the search tree to change their values. The heuristics of dynamic ordering of value and variable would intuitively help to avoid thrashing on values selected by the very first agents and improve the ordering of agents.

An example is shown in Fig.2 to illustrate how the heuristics work. Agent A5 has 7 values in its domain. The value of agent A1 conflicts with the values (value id) 1, 2, 4, 5 of agent A5, thus these values are removed from the available values of agent A5. The value of agent A2 conflicts with the values 2, 3, 4, 6. The value of agent A3 conflicts with the values 1, 3, 4. The value of agent A4 conflicts with the values 4, 6, 7 where the value 7 is the last available value in the domain of agent A5. Thus A4 is the culprit agent with respect to agent A5. Following the first step, agent A5 increases the counter of agent A4, sends a BT to agent A4. Agent A4, upon receiving the BT, increases the counter of agent A5 and increases the counter of its corresponding value.

In the second step, agent A5 determines that A1 and A2 are the PC agents, as they are first agents who remove the values 1, 2, 3, 4, 5, 6 from its domain. Agent A3 is not a PC agent, since its value conflicts with the values 1, 3, 4 of A5 that have been removed by conflicting with the value of agents A1, A2. Thus, agent A5 increases the counters of A1 and A2, sends PC messages to A1 and A2. Agent A1 and A2, when receive the PC message, increase the counter of A5 and increase the counter of their corresponding value.

Detailed Algorithm Description

The *DDBJ* algorithm uses 8 types of messages as follows:

1. SUCCESS: a *termination* message which is broadcasted to all agents, by the last assigned agent, when a solution has been found.
2. FAILURE: a *termination* message which is broadcasted to all agents, by the first agent, when it has determined the problem has no solution.

3. ERROR: a *termination* message which is broadcasted to all agents when the algorithm encounters error (e.g. exceeded limit of time/resources).
4. OK: a message which contains the current partial solution (CPS) composed of a list of $(variable, value)$ tuples and their associate $TimeStamp$'s. This message is sent to the next agent according to the sending agent's decision of ordering.
5. FC: a message which contains a copy of OK message. This message is sent by the assigning agent to the linked agents that have not been assigned, according to its $AgentView$.
6. NG: a message which contains a nogood partial solution. It is sent to the linked agents that have not been assigned, according to its $AgentView$.
7. BT: a message which contains a nogood partial solution. It is sent back to the culprit agent (the last agent in the nogood partial solution).
8. PC: a message which contains a nogood partial solution. It is sent to potential conflicting agents determined by the agent when a conflict occurs.

The *DDBJ* algorithm is executed simultaneously on all agents in parallel. An appropriate function is called depending on the type of the received message. At start, an empty OK message is sent to the first agent for initialization.

Upon receiving an OK message, function receiveOK() is executed. It first checks if the message is valid (line 1); otherwise, it is older than, or equally timely to, the stored $TimeStamp$s [3] and discarded. Next, $TimeStamp$s get updated (line 2). It then checks whether the message's partial solution (MPS) contains *the previously determined nogood* (meaning current $AgentView.consistent = false$ and the MPS contains $AgentView$). If it is the case, the agent simply does nothing and returns (line 3,4). Otherwise, it updates its context by the MPS (line 6). If the update succeeds, meaning its consistent domain of values is not empty, the agent assigns the value (line 8). Otherwise, it backtracks to the last assigned agent (line 10).

Function receiveFC() is called when an FC message is received. The agent checks and discards obsolete message (line 1), otherwise updates its $TimeStamp$s (line 2). It then checks whether the message does not contain the previously determined nogood. If it is the case, it resets the consistency state to $true$ (line 3,4). Whenever the consistency state is $true$ (line 5), the agent updates its context (line 6). If the update does not succeed, it does the following: sending NG messages to linked agents that are not assigned, sending PC messages to the determined PCAs, updating its memory of PCAs and backjumping to the culprit agent.

When receiving an NG message, the function receiveNG() checks to see if $AgentView$ contains the MPS. If it is the case, it removes last one or more tuples in its $AgentView$ to be the same as the received nogood, restores the values accordingly (which are associate with those tuples) (line 2) and resets the consistency state (line 3). Otherwise, if the message is newer than its $AgentView$, the agent updates its context (line 5,6,7). If the update does not succeed, it functions similarly to function receiveFC(). In both cases, if the agent is an assigned agent, it has to reset itself unassigned (line 11,12).

[3] the latter happens when the agent has already received an NG message which contains the same time flag

```
procedure receiveOK()
 1: if Msg is newer than AgentView then
 2:    update TimeStamps
 3:    if previously determined nogood then
 4:       return
 5:    set AgentView.consistent = true
 6:    updateDomain(MPS)
 7:    if success then
 8:       assignVal()
 9:    else
10:       backJump(previous)
end
procedure receiveFC()
 1: if Msg is newer than AgentView then
 2:    update TimeStamps
 3:    if not previously determined nogood then
 4:       set AgentView.consistent = true
 5:    if AgentView.consistent then
 6:       updateDomain(MPS)
 7:       if not success then
 8:          update PCA
 9:          send NG to unassigned agents; PC to agents in PCA
10:          backJump(culprit)
end
procedure receiveNG()
 1: if AgentView orderly contains Msg then
 2:    restoreDom()
 3:    set AgentView.consistent = false
 4: else if Msg is newer than AgentView then
 5:    set AgentView.consistent = false
 6:    update TimeStamps
 7:    updateDomain(MPS-last)
 8:    if not success then
 9:       update PCA
10:       send NG to unassigned agents; PC to agents in PCA
11:       backJump(culprit)
12: if self is assigned then
13:    reset to unassigned
end
```

Function receivePC() simply updates the agent's memory of PCAs and value priority. Function receiveBT(), when a BT message is received, first updates the memory of PCAs and value priority (line 1,2). It then finds the next available value, by calling function assignVal(). Note that it has to check if the message is still valid (meaning that its variable is assigned and the message is not too old), (line 3,4,5), since several BT messages can be sent simultaneously to the agent, and some have already arrived and been processed.

```
procedure receivePC()
 1: update value priority / PCA
end
procedure receiveBT()
 1: update value priority / PCA
 2: if self is assigned then
 3:    if my AgentView is NOT newer Msg then
 4:       assignVal()
end
procedure assignVal()
 1: findNextVal()
 2: if found a consistent value then
 3:    Increase TimeStamp
 4:    if self is last agent then
 5:       broadcast SUCCESS to all agents
 6:    else
 7:       send OK to next agent; FC to connected agents
 8: else
 9:    backJump(previous)
end
procedure backJump(AgentIndex)
 1: if self is first agent then
 2:    broadcast FAILURE to all agents
 3: else
 4:    set AgentView.consistent = false
 5:    reset to unassigned
 6:    send BT to agent AgentIndex
 7:    update PCA
end
```

Function assignVal() tries to find a next consistent value (line 1), forwards the CPS to the next agent (line 7), otherwise it backtracks (line 9). Function backJump($AgentIndex$) performs the backjumping by resetting the agent context and sending BT message to agent $AgentIndex$. Function updateDomain(MPS) simply updates its value domain, $AgentView$ with the input MPS. As soon as it finds the domain empty, the function returns the detected nogood.

5 Soundness, Completeness and Termination

The argument for soundness is close to the one given in [8]. The fact that agents only forward consistent assignments in OK messages at only one place in function assignVal(), line 7, implies that the receiving agents receive only consistent assignments. A solution is reported by the last agent only in function assignVal() at line 5. At this point, all the agents have assigned their variables, and the assignments are consistent. Thus the algorithm is sound.

For completeness, we need to show that *DDBJ* is able to produce all solutions and terminate. The algorithm only backtracks, by sending BT messages, in function backJump(), which implements the graph-based backjumping. It has been shown in [10] that graph-based backjumping only makes *safe jumps*. In other words, the algorithm backjumps to the culprit variable, and this jump does not lead to missing any solution. Similarly in *DDBJ*, multiple *safe jumps* may be performed at the same time simultaneously which are caused by different culprits detected by different agents. The re-assignments of the culprit agents then happen simultaneously. However, the one with the highest level in the search hierarchy tree will eventually replace all others. Thus the algorithm performs an exhaustive search and is able to produce all solutions. Hence, it is complete.

In each backtrack step, there is at least one value of a variable that is removed (line 5 in backJump()). The domains of variables are finite implies finite number of backtracks, or BT messages, until FAILURE messages are broadcasted (line 2 in backJump()). Similarly, each OK message (only sent in assignVal(), line 7) increases the number of assigned variables by 1, until the last variable where SUCCESS messages are broadcasted. Therefore, the algorithm terminates,

In *DDBJ*, agents do not have to store nogoods. An agent has to keep only the current $AgentView$ and the associated $TimeStamp$'s, which have at most n elements. In addition, an agent also needs to maintain two priority lists of its value domain and other agents. Thus, the algorithm's spatial complexity is linear.

6 Experimental Results

This section gives an experimental evaluation of our algorithm *DDBJ* in comparison with two other well known algorithms, the distributed asynchronous algorithm - *DiDB* ([6]) and the distributed synchronous algorithm - *AFC* ([8]). The *DDBJ* is tested in 2 versions: one version is without the dynamic ordering heuristics, called *DBJ*, to measure the performance of the semi-asynchronous backjumping procedure itself, and the other version is the full *DDBJ* algorithm.

The algorithms are tested on distributed binary CSPs which are randomly generated using the problem generator *JavaCSP* ([11]). The problems are generated based on 4 setting parameters:

- v - The number of variables (or number of agents),
- d - The number of values in the domain of each variable (domain size),
- c - The constraint density (which reflects the number of constraints), and
- t - The constraint tightness (which refers to the number of value pairs which are disallowed by the constraint).

These settings are commonly used in experimental evaluation of CSP algorithms ([12], [13], [8]). The problem generator has the ability to generate only feasible problem instances (having solutions). Thus, it is advantage to generate only feasible problem instances for problems in transition phase which are most hardest to solve and so it is easy to highlight differences in algorithm performance ([3]). Note that the problem instances are generated with the setting parameters applied globally, not by interleaving of independent subproblems.

We recall the distinction between Distributed Systems and Distributed Computing ([3]). The latter is belong to the research field of High Performance Computing, where the problem is to divide/distribute, in a efficient way, some computation load onto several connected (or distributed) computing machines. The efficiency is then defined as $speedup/N$ where N is the number of distributed machines ([14]).

In this work, we are concerning the former case, Distributed Systems, where the problems in question have their distributed characteristics in nature: they are spread over a number of distributed agents. As in [3], [6], [8], [5], we use the following measures as the criteria for evaluation:

- *Number of cycles* (or running time): to estimate the algorithm concurrency / asynchrony, as used in [2].
- *Number of messages*: to estimate the overhead of the algorithm affecting on the distributed environment, where the cost of sending messages is usually considered being more expensive than local computation of agents ([8]).
- *Number of constraint checks*: to evaluate computational efforts done locally by the agents.
- *Number of value assignments*: to represent the cost of value changes committed that may be high in some applications.

The first two measures are the most important factors in measuring the efficiency of distributed algorithms. The number of cycles indicates the running time of an algorithm. More importantly, it shows how much parallelism is exploited in asynchronous algorithms compared to synchronous ones. The notion of "concurrent checks" is discussed in [15]. In this work, we make an assumption that the constraints are simple so that an agent is able to process incomming messages, perform necessary constraint checks and send out messages in one clock cycle ([2]). Thus, the ratio "N.Constraint checks/N.Cycles" gives a good estimate of the average number of concurrent constraint checks. As argued in [16], synchronous distributed algorithms usually have better efficiency than asynchronous ones (in terms of overheads, redundant efforts, etc.), but asynchronous algorithms can exploit concurrency, thus resulting in better running time (or less number of running cycles). The latter issue is not discussed in [8] when the authors compare *AFC* with asynchronous algorithms *ABT* and *DiDB*.

The messages are set up to be delivered to destination *not necessarily in the order in which they were sent*, except for the algorithm *DiDB* where it requires the messages are delivered in order. The number of messages is an important measure for DisCSP algorithms, since in distributed environment, sending messages to other distributed agents is considered expensive ([3]).

To simulate a distributed environment and asynchronous execution, we use a discrete event simulator. We have a global discrete clock counting in cycles to simulate a realtime clock. At each cycle, all agents read the incomming messages, process the computation and send out messages to other agents. If there is not any incomming message, an agent simply sits idly. We recall the assumption that an agent is able to process incomming messages, perform necessary constraint checks and send out messages in one clock cycle. The algorithm is executed simultaneously in parallel on all agents. All agents terminate when an termination message is broadcasted and the algorithm finishes. The algorithm's running time is counted as the number of global clock cycles.

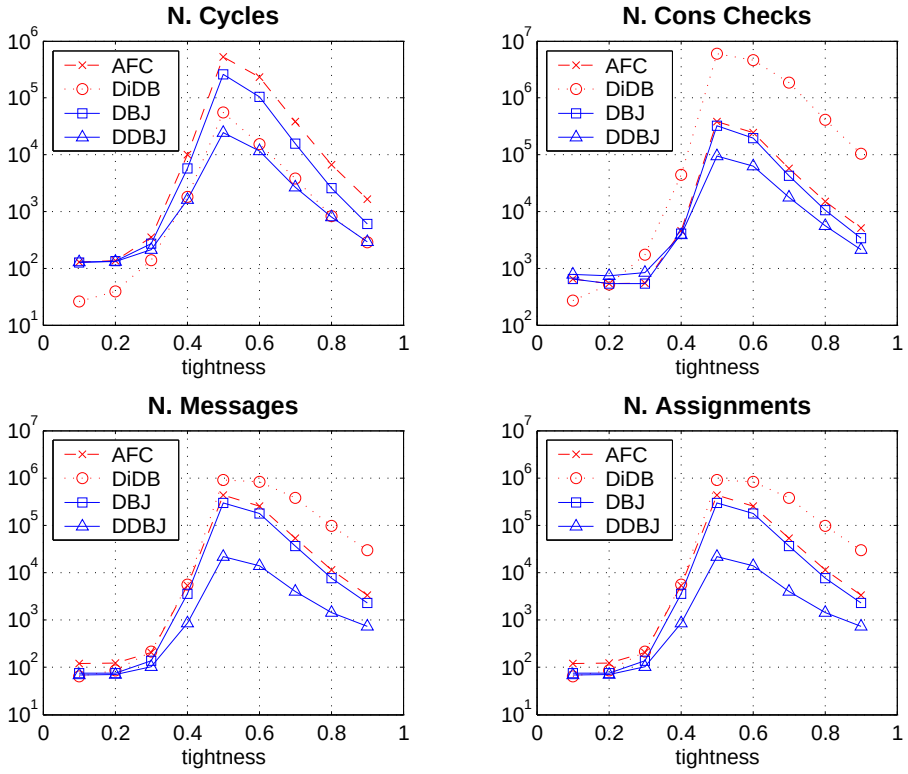


Fig. 3. Results (in *log10* scale) for N.vars v=15, domain d=15, density c=0.5. At transition phase when tightness $t = 0.5 - 0.7$, *DiDB* solved $50\% - 80\%$, *AFC*, *DBJ* and *DDBJ* solved 100% of 100 generated instances

Furthermore, to simulate the real distributed environment as close as possible, we set up the link channels between agents such that the delivery time is randomly generated between 1 and the total number of agents, which best reflects the effect of the size of the constraint network. Because the concurrency of computation of asynchronous algorithms is difficult to see from other measurements (number of constraint checks, number of messages), this setting helps to differentiate asynchronous and synchronous (or sequential) execution schema. The same argument for fairness comparison is also pointed out in [15].

Because of limited space, the results of 2 test sets are presented. The first test set includes problems with the number of variables $n = 15$, the variable domain $d = 15$, the constraint density probability $c = 0.5$ and the constraint tightness varying from 0.1 to 0.9 in 0.1 steps. The results in *log10* scale are shown in Figure 3. Each plot point is the average of results taken from 100 randomly generated instances. An algorithm is stopped when the number of running cycle reaches a limit of $10,000,000$ cycles or the number of messages sent *in one cycle* exceeds $100,000$.

In term of running time, the *DBJ* is about 2-4 times faster than the *AFC* at transition phase. The difference indicates the concurrency effect of the asynchronous backward phase of *DBJ*. The *DiDB*, because of its fully asynchronous nature, is better than the *DBJ* and *AFC*. However, when combined with the dynamic ordering heuristics, the *DDBJ* is the best algorithm among the four for most cases.

On number of messages, the *DDBJ* is better than the other three algorithms by a factor of one order approximately. The only drawback is that the message OK of *DDBJ* (and *AFC*, *DBJ*) is longer than that of *DiDB*. However, since the number of elements in a message is at most equal to the number of variable n and each element contains agent id, value id and its associate $Timestamp$, that all can be represented by 3 integer numbers, the size of a message is not more than $3n$ integer numbers.

In term of computational performance, *DDBJ* outperforms both algorithms *DiDB* and *AFC* by a factor of 5 to 100 on hard instances, where the *DBJ* comes next. This can be explained by the fact that by combining good value/variable ordering heuristics and exploiting concurrency, it also helps to increase the algorithm's computational efficiency and reduces the number of messages. Note that the synchronous algorithm *AFC* always performs better than the fully asynchronous algorithm *DiDB*, that it agrees with the result obtained in [8].

In more details, at transition phase where problems are hardest to solve (constraint tightness is between 0.5 and 0.7), *DiDB* is only able to solve $50\% - 80\%$ of the generated problem instances: we stop the algorithm when the number of messages sent in one cycle exceeds the limit of $100,000$ messages, since most of the time and memory resources are consumed by processing duplicated messages. This message duplication problem arises significantly when the messages are delivered with some random delay. The other three algorithms are able to solve all the problems within the limits of running cycles and messages.

In the second test set, we evaluate the algorithms by 4 feasible, high dimension problems, with the number of variables equals 20, 30, 30 and 40, respectively. The constraint tightness is set to a value close to 0.5 so that the problems are in the transition phase. The limit of number of cycles is now set to $100,000,000$. We exclude *DiDB* because of its limited capacity of solving high dimension problems: the number of messages explodes exponentially so that after a few hundred running cycles, the number of messages soon exceeds the limit of available resource. The results in *log10* scale are shown in Figure 4. The percentages show the numbers of problems solved by the algorithms. Each subgraph shows the median value of the results of 50 generated instances. The reason of taking the median value instead of the mean value is that in the transition phase, the variance of the results is too high, thus the median value indicates better the result average.

It is clear that the semi-asynchronous algorithm *DBJ* always performs better than *AFC* by a factor of 2 or more. It shows the effect of the asynchronous backjumping phase on the algorithm efficiency. The *DDBJ* outperforms both the others by a factor of one to two orders for all measures. On the number of problems solved, the *DDBJ* is able to solve all the problem instances for the 4 cases within the time limit, where the other two algorithms can not. This measure again confirms the high efficiency of the heuristics used in *DDBJ*. For the last two problems where the numbers of variables are

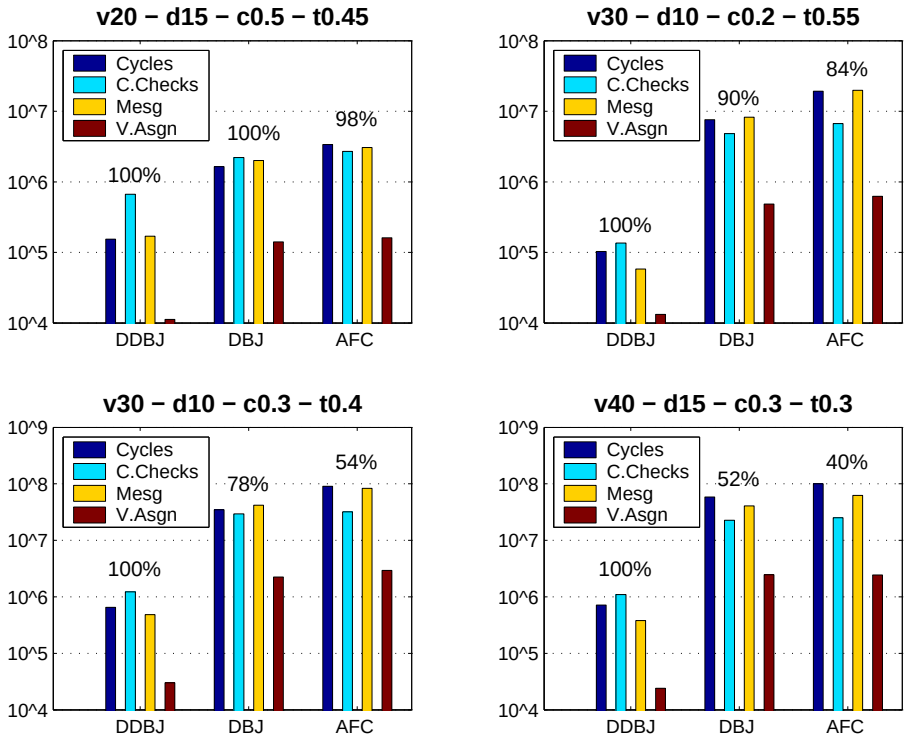


Fig. 4. Results (in *log10* scale) of feasible, high dimension problems. The percentages represent the number of problems solved within a time limit.
a) N.vars v=20, domain d=15, density c=0.5, tightness t=0.45
b) N.vars v=30, domain d=10, density c=0.2, tightness t=0.55
c) N.vars v=30, domain d=10, density c=0.3, tightness t=0.4
d) N.vars v=40, domain d=15, density c=0.2, tightness t=0.4

30 and 40, *AFC* is able to solve only 54% and 40% of the instances. The performance measures of *AFC* are at least one order behind those of *DDBJ*. These factors will be larger if we increase the running time limit for *AFC* to solve more instances.

One can also notice that as the number of variables increases, the performance difference between the *DDBJ* and the other algorithms increases. When v=15, *DDBJ* is faster by a factor of about one order, when v=30,40, *DDBJ* outperforms the others by a factor of about two orders of magnitude on number of running cycles and number of messages.

7 Conclusion

A new complete, distributed, semi-asynchronous algorithm, *DDBJ*, is presented. The algorithm adopts a sequentially assigning procedure, an asynchronous forward checking scheme in its *advancing phase* and an asynchronous graph-based safe-backjumping

scheme in its *backjumping phase*. The sequentiality of variable assignment enables *DDBJ* to integrate the powerful heuristics of dynamic value and variable ordering and still easily to control the algorithm completeness. Experimental results show that the *DDBJ* algorithm outperforms the *DiDB* and the *AFC* algorithms by a factor of *one to two* orders of magnitude on hard instances of randomly generated DisCSPs, both on concurrent running time, number of messages and on other measures of number of constraint checks, number of variable assignments.

Acknowledgments

We would like to thank Prof. Amnon Meisels for his visiting presentation on the *AFC* algorithm. We also thank Arnold Maestre, Dr. Christian Bessière for their helpful explication of the *DiDB* algorithm. Many thanks to Dr. Bart Craenen for his problem generator *JavaCSP*. This work was performed at the Artificial Intelligence Laboratory of the Swiss Federal Institute of Technology in Lausanne and was sponsored by project COCONUT under contract number IST-2000-26063.

References

1. Yokoo, M., Hirayama, K.: Algorithms for Distributed Constraint Satisfaction: A Review. In: Proceedings of Autonomous Agents and Multi-Agent Systems. (2000)
2. Yokoo, M., Durfee, E., Ishida, T.: Distributed constraint satisfaction for formalizing distributed problem solving. In: Proceedings DCS. (1992)
3. Yokoo, M.: Distributed Constraint Satisfaction. Springer-Verlag (2001)
4. Ginsberg, M.: Dynamic Backtracking. Journal of Artificial Intelligence Research **1** (1993) 25–46
5. Hamadi, Y.: Interleaved backtracking in distributed constraint networks. International Journal on Artificial Intelligence Tools **11** (2002) 167–188
6. Bessière, C., Maestre, A., Meseguer, P.: Distributed Dynamic Backtracking. In: Proceedings of the IJCAI'01 workshop on Distributed Constraint Reasoning. (2001)
7. Silaghi, M., Sam-Haroud, D., Faltings, B.: Asynchronous Search with Aggregations. In: Proceedings AAAI'00. (2000)
8. Meisels, A., Zivan, R.: Asynchronous Forward-checking on DisCSPs. In: Proceedings of the Workshop on Distributed Constraints (DCR-03), Acapulco, August 2003. (2003)
9. Dechter, R.: Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. Artificial Intelligence **41(3)** (1990) 273–312
10. Dechter, R., Frost, D.: Backtracking algorithms for constraint satisfaction problems - a tutorial survey. Technical report, University of California, Irvine (1998)
11. Craenen, B.: JavaCsp package. http://www.xs4all.nl/˜bcraenen/JavaCsp/ (2003)
12. Prosser, P.: Binary constraint satisfaction problems: some are harder than others. In: Proceedings of the 11th European Conference on Artificial Intelligence - ECAI'94. (1994)
13. Bessiere, C.: Random Uniform CSP Generators. http://www.xs4all.nl/˜bessiere/generator.html (1996)
14. Dowd, K., Severance, C.: High Performance Computing. Second edn. O'Reilly & Associates (1998)
15. Meisels, A., Kaplansky, E., Razgon, I., Zivan, R.: Comparing performance of distributed constraints processing algorithms. In: Proceedings of the Workshop on Distributed Constraint Reasoning, in AAMAS-2002. (2002)
16. Barbosa, V.C.: An Introduction to Distributed Algorithms. The MIT Press (1996)

Incremental Constraint Propagation for Interleaved Distributed Backtracking *

Georg Ringwelski

Cork Constraint Computation Center, University College Cork, Ireland
g.ringwelski@4c.ucc.ie

Abstract. This paper describes a way to integrate established propagation techniques known from monolithic algorithms in the asynchronous distributed tree-based algorithm IDIBT without extending its protocol. It furthermore provides some preliminary experimental results of the integrated algorithms. These results show that propagation can often, especially in very hard problems, reduce the number of required messages to solve the problem. The trade-off for this, i.e. the number of required (concurrent) constraint checks, naturally increases when more propagation is applied. However, this computational effort can be reduced when propagation is performed incrementally as in the algorithms introduced in this paper.

1 Introduction

Constraint propagation is known to massively improve the performance of search in monolithic Constraint Satisfaction Problems (CSP). With propagation the search space can be significantly pruned such that solutions are found quicker. In the past two decades several propagation algorithms to enhance tree-based search have been investigated. The most popular are Forward-Checking and Look-Ahead which improve the performance of complete depth-first search for almost all problems although they produce some computational overhead.

Consequently, an integration of Propagation in distributed search algorithms was a topic of various research efforts in recent years. It could be very successfully applied in synchronized distributed search algorithms [7] (the propagation may still be asynchronous). Synchronized distributed algorithms with propagation currently seem to be among the fastest algorithms to solve Distributed CSP. It seems that the pruning of the search space makes up the drawback of synchronization, namely that no use is made of parallel computational power. In completely asynchronous search it is unclear whether propagation will have a positive effect at all. The crucial point is that in asynchronous algorithms the processes have to reason on the basis of beliefs which can be wrong. If the results of such error-based reasoning are propagated, this may handicap the global

* This work has received support from the Embark Initiative of the Irish Research Council of Science Engineering and Technology under Grant PD2002/21 and from Science Foundation Ireland under Grant 00/PI.1/C075

approximation to a solution. I know of only one family of algorithms that performs propagation during *asynchronous* distributed search is DMAC-ABT [9, 5]. These algorithms are said to maintain any kind of consistency (not necessarily AC) during the complete and asynchronous distributed search with the algorithm AAS [10]. Using bounds-consistency DMAC-ABT is said to use in average 10 times fewer messages than AAS on random problems.

In this paper I investigate to which extend constraint propagation can improve the performance of the asynchronous distributed search algorithm Interleaved Distributed Backtracking [4]. I integrate Forward-Checking and Partial-Look-Ahead in the complete algorithm IDIBT and analyze the performance of the resulting algorithms.

The paper is organized as follows: In the next Section I provide some basic definitions from CSP, Distributed CSP, Distributed Systems and give a short Introduction to Forward-Checking and Look-Ahead; In Section 3 I outline the simplified version of the IDIBT algorithm that I will use as a basis for the new algorithms IDIBT_FC and IDIBT_LA which are described in Sections 4 and 5; In Section 6 I prove the correctness of the new algorithms and provide some preliminary experimental results; Section 7 concludes and describes some directions for future work.

2 Preliminaries

(Distributed) Constraint Satisfaction Problems A Constraint Satisfaction Problem (CSP) is given by a triple (C, X, D) where C is a set of constraints, $X = \{x_1, ..., x_n\}$ is a set of variables and $D = \{D_1, ..., D_n\}$ are their respective domains. In a binary CSP each constraint $C_{ij} \in C$ is associated to a binary relation $sem(C_{ij}) \subseteq D_i \times D_j$ in the Cartesian Product of the domains of the constraint's two variables x_i and x_j. A Distributed CSP (DisCSP) is given by a tuple (C, X, D, A) such that (C, X, D) is a CSP which is distributed among a set of uniform agents A.

Constraint Solving and Constraint Propagation With Constraint Solving we refer to the process of finding a solution to a CSP or DisCSP or proving that no solution exists. A solution to $P = (C, X, D)$ is a variable assignment which maps to every variable $x_i \in X$ a value $d_i \in D_i$, such that $\forall C_{ij} \in C : (d_i, d_j) \in sem(C_{ij}) \wedge \forall C_{ji} \in C : (d_j, d_i) \in sem(C_{ji})$ holds. Given a total order $\prec$ on X the complete labeling tree associated to P and $\prec := \{(x_i, x_j) | i < j\}$, is given by

- the direct descendants of the root are (x_1, d) for each $d \in D_1$
- The direct descendants of node (x_j, d) are the nodes (x_{j+1}, e) where $e \in D_{j+1}$.

In a complete labeling tree, each path from the root to any leaf represents one distinct assignment of all variables to values from their respective domain. Such a tree can be pruned without restricting the set of solutions. Whenever no solution

is represented by the assignments in a subtree, this subtree can be safely pruned. **Forward-Checking** considers the constraints between any node (x_i, d_i) and all nodes (x_j, d_j) in the subtree with root (x_i, d_i). Whenever a constraint C_{ij} exists such that $(d_i, d_j) \notin sem(C_{ij})$, then the node (x_j, d_j) and all its descendants can be pruned. Informally speaking this means that with the choice of a value for a variable all values from future variables that are inconsistent with that choice are not considered during search. **Look-Ahead** prunes even more from the search tree. It uses the notion of Arc-consistency (AC), which is defined as follows. A constraint C_{ij} (i.e. an arc in the constraint graph associated to a CSP [3]) is said to be arc-consistent if and only if for each value $d_i \in D_i$ at least one value $d_j \in D_j$ exists such that $(d_i, d_j) \in sem(C_{ij})$. A binary CSP (resp. its associated graph) is said to be arc-consistent if and only if all its constraints are arc-consistent. Partial-Look-Ahead enforces AC after Forward-Checking for all future variables of the search, i.e. AC of all constraints between variables that are not yet labeled. Full-Look-Ahead or MAC (Maintaining Arc-consistency) enforces arc-consistency after Forward-Checking for the whole CSP. Look-Ahead may exclude further values from the domain of un-instantiated variables and thus prune the respective subtrees in the complete labeling tree. For a more detailed description of these techniques please refer to [1].

Distributed Systems Distributed Systems (DS) are given by a set of agents which are executed in concurrent processes and communicate by messages. Each agent can send any other a message provided it knows its address. In this paper, I use the following communication primitives for DS:

send(R,msg) sends the message **msg** to the each agent in **R**. The message can be any string;

behavior receive(msg) provides an agent-behavior, i.e. a procedure which is called as soon as a certain message **msg** arrives. **msg** can be or can contain variables which will be unified with the received values. Thus **msg** is usually a pattern which an incoming message must match in order to trigger the behavior. The pattern is usually a structured term which identifies the kind of message received, and its arguments are variables which are unified with the actual information contained in the message;

Furthermore, I assume that every message eventually arrives, and that between every directed pair of agents the order in which messages are sent is the same as that in which they are received. Whenever an agent refers to itself it will use the synonym `self`. A more detailed introduction to distributed computing can be found in [2].

3 Background: The IDIBT Algorithm

Interleaved Distributed Backtracking (IDIBT) is defined in [4]. It solves DisCSPs, where each variable imposes one agent and the same assumptions concerning the

message passing are made that I described in the previous Section. That paper introduces a protocol for finding a global static variable ordering and a protocol for search. The first is not of interest in this paper. I assume any static order $\prec$ of variables to be given. From this order, IDIBT infers for each agent x_i the sets of children $\Gamma^+ = \{x_j | x_i \prec x_j \wedge \exists C_{ij} \in C\}$ and parents $\Gamma^- = \{x_j | x_j \prec x_i \wedge \exists C_{ij} \in C\}$.

The search algorithm is given by a protocol for multiple uniform worker agents and one controller agent which starts the algorithm and is to detect termination. Each agent executes NC parallel search contexts to speed up the algorithm. For simplicity I omit this technique and assume NC=1 throughout this paper without restricting generality. Upon initialization, each worker x_i chooses any value d_i from its domain and sends an `infoVal(`x_i, d_i`)` message to each agent in Γ^+. Then search is performed by two behaviors: One processes `infoVal` messages and one processes `btSet` messages, they are presented in Algorithms 1 and 2 and outlined in the following text. The behaviors use the following local data structures and procedures:

Γ^- **and** Γ^+ as described above
`myDomain` the allowed values for `self`
`constraints` the constraints over `self`
`myVal` the current value of `self`
`myCpt` the current instantiation number of `self`. This is used as a time-stamp of the assignment to `myVal`
`value[p]` the currently known value of agent `p` for each `p` $\in \Gamma^-$
`cpt[p]` the time-stamp associated to `value[p]`
procedure getValue(*type***)** returns a value which is consistent with Γ^- and increments `myCpt`. If no such value exists it does not increment the counter. If *type* equals $info$, then it returns the first value from `myDomain` starting at `myVal` in a circular manner. If *type* = bt, it returns the first value that is larger than `myVal`.
procedure contextConsistency(*rcpt***):boolean** returns true, iff the context represented in *rcpt* is consistent with `myCpt` and `cpt`, i.e. if the timestamps for all elements in rcpt and the local knowledge are equal. Otherwise it returns false.
procedure nearest(*A***):***agent* returns the nearest agent from A, which is in IDIBT the smallest which is greater than `self` with respect to the chosen order of agents.

When an agent receives the information of a new assignment $x = d$ of one of its parents with an `infoVal(`x, d`)` message, it updates its local view to x by adapting `value[`x`]` and incrementing `cpt[`x`]`. Then it tries to find a value which is compatible with the new knowledge. If it finds one and if it is different[1] from the previous one it sends respective `infoVal` messages to all children. If it cannot find a consistent value in its domain, it initiates backtracking and sends a `btSet`

[1] This is not described in [4], but implemented in Youssef Hamadi's version of the algorithm

Algorithm 1: IDIBT behavior to process assignments of remote agents

```
behavior receive(infoVal(x, d)) begin
    value[x] := d;
    cpt[x] := cpt[x] +1 ;
    oldVal := myVal;
    myVal := getValue(info);
    if myVal ≠ nil AND myVal ≠ oldVal then
        sendMsg(Γ+,infoVal(self,myVal));
    else
        sendMsg(nearest(Γ−),btSet(Γ−,cpt[Γ−]);
end
```

message to its nearest parent. This message contains the set of all parents and the currently known context `cpt` of `self`.

Algorithm 2: IDIBT behavior to process backtracking messages

```
behavior receive(btSet(set, rcpt)) begin
    if contextConsistency(rcpt) then
        myVal := getValue(bt);
        if myVal ≠ nil then
            sendMsg(Γ+,infoVal(self,myVal));
        else
            if Γ− ∪ set = ∅ then
                broadcast(noSolution);
            else
                sendMsg(nearest(Γ− ∪set),btSet(Γ− ∪set,cpt[Γ− ∪rcpt]);
end
```

When an agent receives a `btSet` message, it checks whether the context in that the message was created matches the currently known local context. If not, the backtracking was initiated on different beliefs and is thus obsolete. If the context matches, `self` tries to find another consistent value from its domain which was not used before. If such a value exists, it is communicated to the children. If no consistent value in `myDomain` remains, further backtracking becomes necessary. However, if there are no parents, i.e. `self` is the root node of the labeling tree we can deduce that the DisCSP is insoluble. This is broadcast to all other agents to make them terminate. If `self` has no values left, but has parent agents, it tries to make them backtrack and change their values. This is done by sending a `btSet` message to the nearest parent or the nearest node in the set of parents of the children that initiated the backtracking.

4 The IDIBT_FC Algorithm

IDIBT already uses a form of Forward-Checking by choosing only values for `myVal` which are consistent with Γ^-. This is indeed Forward-Checking, because Γ^- includes all agents that have constraints with `self` and are located closer to the root of the labeling tree. The procedure **getValue** in IDIBT checks every new value against all constraints with parent variables. However, the domain of the variable is not changed in IDIBT and the computation of consistent values must be performed from scratch with every received `infoVal` message. IDIBT_FC does the constraint checking incrementally and prunes the variable domain such that the procedure **getValue** can return any value from the current variable domain while still implementing Forward-Checking. With this we can omit the constraint checks performed by **getValue** in IDIBT. The IDIBT_FC behavior to process `infoVal` messages extends the IDIBT algorithm by pruning and relaxing `myDomain` incrementally with every received new knowledge about remote assignments. IDIBT_FC uses Algorithm 2 to process `btSet` messages just as IDIBT, but differs in the behavior to process `infoVal` messages. The IDIBT_FC version of the latter behavior is presented in Algorithm 3. When an

Algorithm 3: IDIBT_FC behavior to process assignments of remote agents

```
   behavior receive(infoVal(x, d)) begin
1  |  if value[x] ≠ nil then
   |  |  relax(x,value[x],cpt[x]);
   |  value[x] := d;
   |  cpt[x] := cpt[x] +1 ;
2  |  propagate(x,d,cpt[x]);
   |  oldVal := myVal;
   |  myVal := getValue(info);
   |  if myVal ≠ nil AND myVal ≠ oldVal then
   |  |  sendMsg(Γ+,infoVal(self,myVal));
   |  else
   |  |  sendMsg(nearest(Γ−),btSet(Γ−,cpt[Γ−]);
   end
```

IDIBT_FC agent receives an assignment of a remote agent, it relaxes its domain by "de-propagating" the formerly known assignment (line 1) of that variable. After updating its local knowledge it prunes its domain with Forward-Checking the new assignment (line 2). The manipulation of `myDomain` is performed by the procedures **relax** and **propagte** which are presented in Algorithms 4 and 5. They use the the following additional data structures:

`initDomain` the initial domain of `self`

`counter[`d`]` the number of removals of value $d \in$`initDomain`. It counts, how often the value d had to be pruned due to Forward-Checking

`removed[`$(var, val, ccnt)$`]` the set of all values that are pruned due to the assignment of the remote variable var to val when `cnt[`var`]` equals $ccnt$.

Furthermore the procedure **getValue(***type***)** is changed, such that in the case of $type = bt$ it returns the value from `initDomain` which is greater than `curVal`. With this adaptation it is invariant against the changes of `myDomain` and thus behaves exactly as in in IDIBT. In the case of $type = info$ the procedure **getValue** does not perform any constraint checks, but just delivers the next value from `myDomain`. Thus, it differs from the respective procedure in IDIBT which checks consistency with Γ^- for each value to be returned.

Algorithm 4: Procedure to relax former assignment

```
procedure relax(var, val, ccnt) begin
    foreach d ∈ removed[(var, val, ccnt)] do
        counter[d] := counter[d] -1;
        if counter[d] = 0 then
            myDomain := myDomain ∪{d};
end
```

The procedure **relax** (Alg. 4) decrements the `counter` for all values which were pruned due to the obsolete assignment $(var, val, ccnt)$. If one of these `counter`-values thus becomes zero, no justification remains to exclude the respective value from `myDomain` and it is consequently re-inserted in `myDomain`.

Algorithm 5: Procedure to propagate new assignment

```
procedure propagate(var, val, ccnt) begin
    D := getInconsitentValuesFC(var, val);
    foreach d ∈ D do
        myDomain := myDomain \{d} ;
        counter[d] := counter[d] +1;
    removed[(var, val, ccnt)] := D;
end
```

The procedure **propagate** (Alg. 5) computes all values from `initDomain` which are inconsistent with the assignment $(var, val, ccnt)$. This is done with the procedure **getInconsistentValuesFC** which is presented in Algorithm 6. All of these values d, non-regarding whether $d \in$ `myDomain`, are removed from `myDomain` and the pruning is counted by incrementing `counter[`d`]`. Finally the pruned values are stored in `removed[`$(var, val, ccnt)$`]`.

The procedure **getInconsistenValuesFC(***var*, *val***)** (Alg. 6) checks for each value $d \in$ `initDomain` whether it is consistent with the constraint c over `self` and var. If no such constraint exists d is consistent, otherwise d is consistent if

Algorithm 6: Procedure to compute inconsistent values for IDIBT_FC

```
procedure getInconsistenValuesFC(var, val):set begin
    res := ∅;
    foreach d ∈initDomain do
        if (c(self, var) ∈ constraints ∧ (d, val) ∉ sem(c)) ∨ (c(var, self) ∈
        constraints ∧ (val, d) ∉ sem(c)) then
            res := res ∪{d};
    return res;
end
```

$(d, val) \in sem(c)$ or $(val, d) \in sem(c)$. If d is not consistent it is added to the set of values which can be pruned, i.e. the set which is returned by the procedure.

5 The IDIBT_LA Algorithm

The IDIBT_LA algorithm works very similar to the IDIBT_FC algorithm. The difference is that the procedure **getInconsistentValuesFC** is replaced by **getInconsistentValuesLA**, which is capable of detecting more values which can be safely removed from `myDomain`. It computes all values that are inconsistent in the sense of Arc-consistency with the current assignment of parent variables. It thus prunes all non-arc-consistent values from unlabeled variables making it a Partial-Look-Ahead search algorithm. A Full-Look-Ahead respectively a MAC algorithm would require a view to the assignments of children as well and is thus not applicable to IDIBT without extending the protocol. For the extra propagation in Look-Ahead we have, however, to pay a high price (with my current implementation): We have to store the entire CSP in each agent. This is certainly not a practical approach to a distributed Look-Ahead algorithm and I am trying to find decentralized ways to compute the set of those inconsistent values in a distributed algorithm in my ongoing research (cf. [8]). For the current topic of my research, namely the investigation of the speedup we can expect from propagation in distributed search, it is sufficient to use the Look-Ahead algorithm described below.

The procedure **getInconsistentValuesLA(**var, val**)** (Alg. 7) uses an additional data structure `csp` which stores the entire CSP as it was when `self` was created. First `csp` is copied to ccsp and the domain of var is assigned to $\{val\}$. Then Arc-consistency is enforced for ccsp. If this leads to the detection of an inconsistency `initDomain` is returned to make Algorithm 3 backtrack because no values are left that **getValue** could choose. If the resulting CSP is consistent, the pruned values of `self` are returned allowing Alg. 3 to selet one of remaining values and post it to the children of `self`.

The IDIBT_LA algorithm is only used for the experimental evaluation of the integration of propagation in IDIBT. I do not consider it in the theoretical evaluations because of its practical irrelevance.

Algorithm 7: Procedure to compute inconsistent values for IDIBT_LA

```
procedure getInconsistenValuesLA(var, val):set begin
    ccsp := csp.clone();
    ccsp.assign(var, val);
    ccsp.enforce-AC();
    if inconsistent(ccsp) then
        return initDomain
    return initDomain \ ccsp.getDomain(self);
end
```

6 Evaluation

6.1 Correctness of IDIBT_FC

Assuming that IDIBT is sound and complete I can show that IDIBT_FC is also. For this I show that no solutions are pruned by the propagation (completeness), that it does not allow for more solutions (soundness) and that it terminates.

Theorem 1. *The IDIBT_FC algorithm is correct.*

Proof. **Completeness and Soundness.**
The algorithm IDIBT is sound and complete, thus it is sufficient to show that the procedure **getValue** in IDIBT_FC will return just the same values that it returns in IDIBT. Algorithm 5 incrementally prunes all values from `myDomain` that are not consistent with any value known from the view to parent agents stored in `value`. The procedure **getValue(**$info$**)** will only return values from the thus reduced `myDomain`. Whenever an assignment of a remote agent is known to be obsolete, all its consequences are removed from the current knowledge by Algorithm 4. This includes that all values for which no justification remains to exclude them are re-inserted in `myDomain`. All values that are not returned by **getValue(**$info$**)** are thus inconsistent with the current knowledge on parent values and would thus not be returned by **getValue(**$info$**)** in IDIBT. The procedure **getValue(**bt**)** operates on `initDomain` and behaves thus just as in IDIBT. Consequently, **getValue** in IDIBT_FC will return just the same values as it does in IDIBT.
Termination.
Since all variable domains are finite, the sets `removed[`$(var, val, ccnt)$`]` and `initDomain` are finite and thus the loops in Algorithms 4,5 and 6 will terminate. These algorithms send no messages such that no deadlocks may occur. Since IDIBT terminates and the additionally executed algorithms (4, 5 and 6) all terminate, IDIBT_FC will also always terminate.

6.2 Empirical Evaluation

I have implemented the IDIBT, IDIBT_FC and IDIBT_LA algorithms in a multi-threaded Java program. Each agent constitutes one concurrent thread and the

agents communicate by dropping messages to other agents' message-channels. A random delay between 10 and 300 msec is applied for each message delivery. Another source of randomness results from the scheduling of the concurrent threads by the Operating System. This scheduling can be assumed to be fair, but not follow any regular patterns. The common memory of the threads is not used except the references to the channels. To evaluate the efficiency I used the following measures:

- The number of (sequential) messages sent until one solution is found or the algorithm detected that no solution exists. Sequential messages represent the largest number of messages the were executed consecutively. The number of messages exposes the required communication effort;
- The number of concurrent constraint checks [6] which exposes the computational effort of the agents.

I ran two sets of tests to evaluate the performance of the algorithms with Java 1.4.2 on a Linux desktop computer one 1.8GHz Pentium processor and 512MB memory.

rbc-t Random binary CSPs (v, x, d, t) with v variables, domains size x (for each variable), density $= d$ and varying tightness t. The sample size was 20 and the figures show median values.

n-queens The n-queens problem. For each **n** I used a sample size of 20 and the figures show median values.

The experimental results for $(20, 8, 0.2, t)$ instances of **rbc-t** are shown in Figures 1 and 3. All the problems with tightness smaller than 0.53 had solutions, some of the problems with tightness 0.53 to 0.6 had solutions while almost none of the problems with tightness larger than 0.63 had solutions. The results for the bf n-queens are shown in Figures 2 and 4. All instances of this problem have comparably many solutions. Please note that all figures have logarithmic scales!

It can be seen in Figures 1 and 2 that the median number of concurrent constraint checks can almost always be reduced by using incremental constraint propagation in IDIBT_FC compared to IDIBT. When IDIBT_LA is applied, more constraint checks become necessary to enforce the stronger form of local consistency. However, in the tight problems inconsistencies can be detected faster such that the difference of ccc between IDIBT_MAC and the other algorithms becomes smaller.

The median number of messages can be reduced when more propagation is used as can be seen in Figures 3 and 4. The first figure represents the absolute number of messages while the latter represents the largest number of sequential messages. When Look-Ahead is applied IDIBT is almost always significantly faster then with Forward-Checking. Especially insoluble problems can be "solved" much faster with more propagation, because inconsistencies are detected earlier. The number of required messages does not always match between IDIBT and IDIBT_FC, although the same propagation (namely Forward-Checking) is performed. This seems to be an effect of the small sample size I

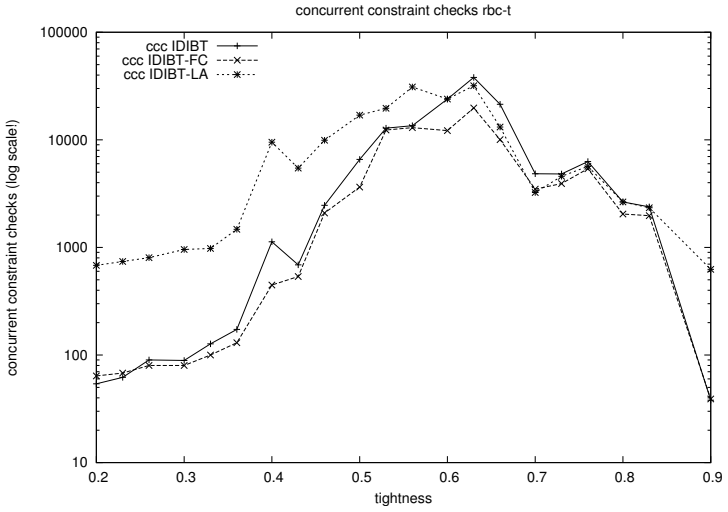


Fig. 1. Required concurrent constraint checks to solve random binary CSP $(20, 8, 0.2, tightness)$.

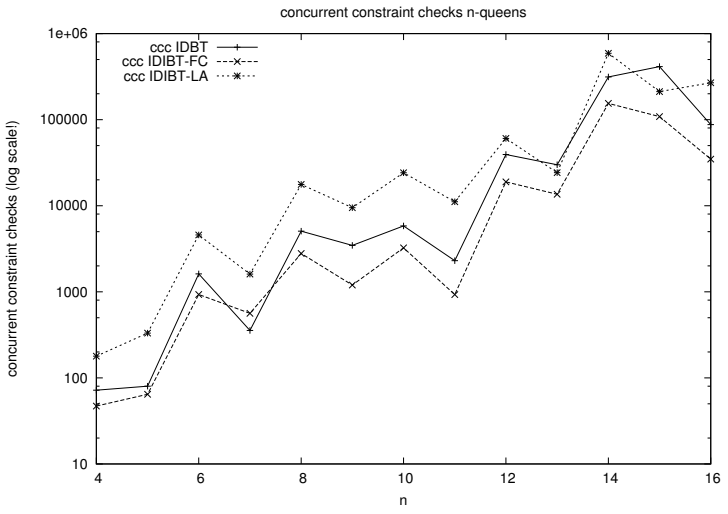


Fig. 2. Required concurrent constraint checks to solve n-queens problem.

used in this set of experiments. The standard deviation of these numbers is very large in all experiments.

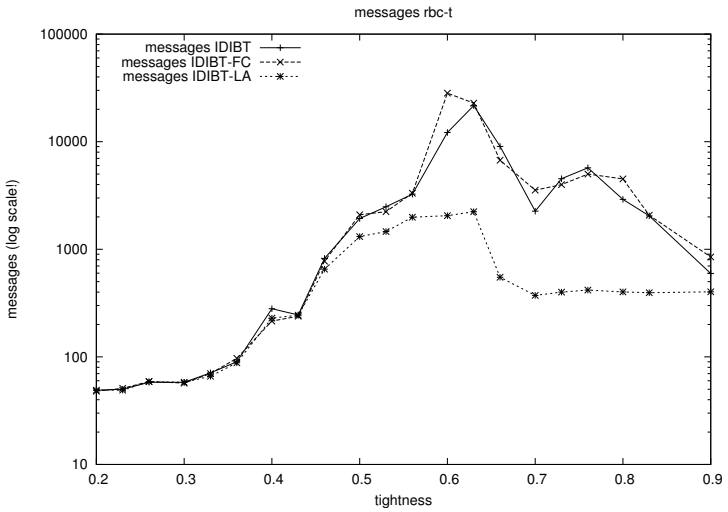


Fig. 3. Absolute number of messages required to solve random binary CSP $(20, 8, 0.2, tighness)$.

7 Conclusion

The number of messages required to solve a DisCSP with the IDIBT algorithm can in most cases be reduced when constraint propagation is applied. The more values are pruned during propagation, the less messages become necessary. The trade-off for this reduction of communication is that more constraint checks become necessary inside the agents. A trade-off between constraint checks and search is well known from monolithic CSP. For most CSPs the integration of some, usually not the most restrictive, propagation into some, usually not the most clever, search algorithm yields the fastest runtime results. In DisCSP, this trade-off has to be re-investigated with respect to the number of required messages. This common metric for the evaluation of distributed algorithms may lead to completely different ideal combinations of search and inference.

Incrementality is an important prerequisite for efficient constraint processing. When solving NP-complete problems, it is often highly desirable to maintain previously processed results and not to recompute everything from scratch. Although constraint propagation is not NP-complete, it still is performed incrementally in most state-of-the-art professional constraint solvers. The algorithms presented in this reduce the required number of constraint checks and thus the

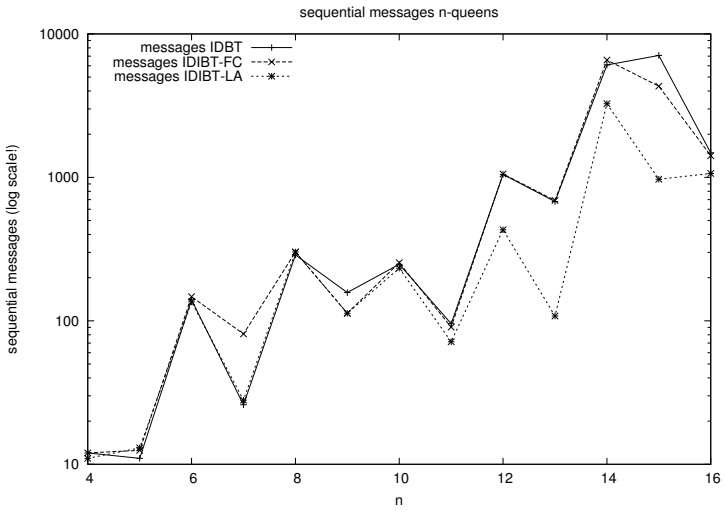


Fig. 4. Required sequential messages to solve n-queens problem.

local computational effort of the agents by using incremental constraint propagation.

In future work we will investigate more precisely the trade-off between computation and communication. Another topic to be looked at is the robustness of the algorithm against message delays. For this we will check the standard deviation of our metrics after running the tests on larger samples. Furthermore we will investigate extensions of the algorithms presented here, which may use additional messages for propagation purposes. This will be done by integrating distributed propagation into a distributed search.

References

1. K.R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.
2. George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems, 3rd edition.* Addison Wesley, 2001.
3. Rina Dechter. *Constraint Processing.* Morgan Kaufmann, 2003.
4. Youssef Hamadi. Interleaved backtracking in distributed constraint networks. *International Journal on Artificial Intelligence Tools*, 11(2):167–188, 2002.
5. M.C.Silaghi, D.Sam-Haroud, and B.Faltings. Maintaining hierarchical distributed consistencies. In *CP2000 DCS Workshop*, 2000.
6. A. Meisels, E. Kaplansky, I. Razgon, and R. Zivan. Comparing performance of distributed constraint processing algorithms. In *Proc. 4th Workshop on Distributed Constraint Reasoning, Bologna, Italy*, 2002.
7. Igor Razgon and Amnon Meisels. Distributed forward checking with dynamic ordering. In *Proc. IJCAI01-Workshop on Distributed Constraint Reasoning*, 2001.

8. Georg Ringwelski. The ddac4 algorithm for arc-consistency enforcement in dynamic and distributed csp. In *Proc. 5th workshop on Distributed Constraint Reasoning, Toronto*, 2004.
9. M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Consistency maintainance for abt. In Toby Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001*, pages 271–285. Springer LNCS 2239, 2001.
10. Marius-Calin Silaghi, Djamila Sam-Haroud, and Boi Faltings. Asynchronous search with aggregations. In *Proc. AAAI/IAAI 2000*, pages 917–922, 2000.

Synchronous, Asynchronous and Hybrid Algorithms for DisCSPs*

Ismel Brito and Pedro Meseguer

Institut d'Investigació en Intel.ligència Artificial
Consejo Superior de Investigaciones Científicas
Campus UAB, 08193 Bellaterra, Spain.
{ismel|pedro}@iiia.csic.es

Abstract. There is some debate about the kind of algorithms that are most suitable to solve DisCSP. Synchronous algorithms exchange updated information with a low degree of parallelism. Asynchronous algorithms use less updated information with a higher parallelism. Hybrid algorithms combine both features. Lately, there is some evidence that synchronous algorithms could be more efficient than asynchronous ones for one problem class. In this paper, we present some improvements on existing synchronous and asynchronous algorithms, as well as a new hybrid algorithm. We provide an empirical investigation of these algorithms on n-queens and binary random DisCSPs.

1 Introduction

In the last years, the AI community has shown an increasing interest in distributed problem solving. Regarding distributed constraint reasoning, several synchronous and asynchronous backtracking procedures have been proposed to solve a constraint network distributed among several agents [15, 16, 6, 13, 1, 14, 4].

Broadly speaking, a synchronous algorithm is based on the notion of *privilege*, a token that is passed among agents. Only one agent is active at any time, the one having the privilege, while the rest of agents are waiting [1]. When the process in the active agent terminates, it passes the privilege to another agent, which now becomes the active one. These algorithms have a low degree of parallelism, but their agents receive updated information. In an asynchronous algorithm every agent is active at any time. They have a high degree of parallelism, but the information that any agent knows about other agents is less updated than in synchronous procedures.

There is some debate around the efficiency of these two type of algorithms. The general opinion was that asynchronous algorithms were more efficient than the synchronous ones, because of their higher concurrency [2]. In the last decade, attention was

* This research is supported by the REPLI project TIC-2002-04470-C03-03.

[1] Except for special topological arrangements of the constraint graph. See [3] for a synchronous algorithm where several agents are active concurrently.

[2] However, a careful reading of [17] shows that "synchronous backtracking might be as efficient as asynchronous backtracking due to the communication overhead" (footnote 15).

mainly devoted to the study and development of asynchronous procedures, which represented a new approach with respect to synchronous ones, directly derived from centralized algorithms.

Recently, Zivan and Meisels reported that the performance of a distributed and synchronous version of Conflict-Based Backjumping (*CBJ*) surpasses Asynchronous Backtracking (*ABT*) for the random problem class $\langle n = 10, m = 10, p_1 = 0.7 \rangle$.

In this paper we continue this line of research, and we study the performance of three different procedures, one synchronous, one asynchronous and one hybrid, for solving sparse, medium and dense DisCSPs. The synchronous algorithm is *SCBJ*, a distributed version of the Conflict-Based Backjumping (*CBJ*) [12] algorithm. The asynchronous algorithm is the standard *ABT* enhanced with some heuristics. The hybrid algorithm is *ABT-Hyb*, a novel *ABT*-like algorithm, where some synchronization is introduced to avoid redundant messages. In addition, we present a detailed approach for processing messages by packets instead of processing messages one by one, in *ABT* and *ABT-Hyb*. We also provide an experimental evaluation for new low-cost heuristics for variable and value reordering.

The rest of the paper is organized as follows. In Section 2 we recall some basic definitions of DisCSP. In Section 3 we recall two existing algorithms for DisCSP solving: the synchronous *SCBJ*, and the asynchronous *ABT*. In Section 4 we present *ABT-Hyb*, a new hybrid algorithm that combines asynchronous and synchronous elements, proving its soundness and completeness. In Section 5 we describe the experimental setting (including some implementation details) and discuss the experimental results. Finally, Section 6 contains several conclusions and directions of further work.

2 Distributed CSP

A constraint network is defined by a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of n variables, $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is the set of their respective finite domains, and $\mathcal{C}$ is a set of constraints declaring those value combinations which are acceptable for variables. The CSP involves finding values for the problem variables satisfying all constraints. We restrict our attention to constraints relating two variables, namely *binary* constraints. A constraint among the variables x_i and x_j will be denoted by c_{ij}.

A distributed CSP (DisCSP) is a CSP where the variables, domains and constraints of the underlying network are distributed among automated agents. Formally, a finite variable-based distributed constraint network is defined by a 5-tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \phi)$, where $\mathcal{X}$, $\mathcal{D}$ and $\mathcal{C}$ are as before. $\mathcal{A} = \{1, \ldots, p\}$ is a set of p agents, and $\phi : \mathcal{X} \rightarrow \mathcal{A}$ is a function that maps each variable to its agent. Each variable belongs to one agent. The distribution of variables divides $\mathcal{C}$ in two disjoint subsets, $\mathcal{C}_{intra} = \{c_{ij} | \phi(x_i) = \phi(x_j)\}$, and $\mathcal{C}_{inter} = \{c_{ij} | \phi(x_i) \neq \phi(x_j)\}$, called intra-agent and inter-agent constraint sets, respectively. An intra-agent constraint c_{ij} is known by the agent owner of x_i and x_j, and it is unknown by the other agents. Usually, it is considered that an inter-agent constraint c_{ij} is known by the agents $\phi(x_i)$ and $\phi(x_j)$ [6, 17].

A solution of a distributed CSP is an assignment of values to variables satisfying every constraint (although distributed CSP literature focuses mainly on solving inter-agent constraints). Distributed CSPs are solved by the collective and coordinated action

of agents $\mathcal{A}$. Agents communicate by exchanging messages. It is assumed that the delay in delivering a message is finite but random. For a given pair of agents, messages are delivered in the order they were sent.

For simplicity purposes, and to emphasize on distribution aspects, along the rest of the paper we assume that each agent owns exactly one variable. We identify the agent number with its variable index ($\forall x_i \in \mathcal{X}, \phi(x_i) = i$). For this assumption, in the following we do not differentiate between a variable and its owner agent.

3 Existing Algorithms for DisCSP

3.1 Synchronous Search: SCBJ

Synchronous procedures can be directly derived from constraint algorithms in centralized search when extended to distributed environments. Generally, only one agent is active at any time in a a synchronous algorithm. Because of this, the active agent has always updated information, in the form of either a partial solution (from the part of the problem already assigned) or a backtracking.

The synchronous backtracking (*SBT*) algorithm for DisCSP was presented in [17]. Synchronous Conflict-Based Backjumping (*SCBJ*) [21] is a distributed version of the centralized Conflict-Based Backjumping (CBJ) algorithm [11]. While *SBT* performs chronological backtrackking, *SCBJ* does not. Each agent keeps the *conflict set* (*CS*), formed by the assigned variables which are inconsistent with some value of the agent variable. Let *self* be a generic agent. When a wipe-out occurs, it allows to *self* to backtrack directly to the closest conflict variable in CS_{self}, say x_i and sends $CS_{self} - \{x_i\}$ to be added to CS_i. Like *SBT*, *SCBJ* exchanges *Info* and *Back* messages, which are processed as follows (*self* is the receiver):

- *Info(partial-solution)*. *self* receives the partial solution, assigns its variable consistently, selects the next variable and sends the new partial solution to it in a *Info* message. If it has no consistent value, *self* sends a *Back* message to the closest variable in CS_{self}.
- *Back(conflict-set)*. *self* has to change its value, because *sender* has no value consistent with the partial solution. The current value of *self* is discarded, and the new conflict-set of *self* is the union of its old conflict-set and the one received. After this, *self* behaves as after receiving a *Info* message.

After receiving any of these messages, *self* becomes the active agent. *self* passes the privilege to other agent sending to it an *Info* or a *Back* message. The search ends unsuccessfully when any agent encounters an empty domain and its *CS* is empty. Otherwise, a solution will be found when the last agent is reached and there is a consistent value for it.

3.2 Asynchronous Search: ABT

In asynchronous search, all agents are active at any time, having a high degree of parallelism. Asynchronous Backtracking (*ABT*) [15, 17–19] was a pioneer asynchronous

algorithm to solve DisCSP. *ABT* requires a total agent ordering. Agent *i* has higher priority than *j* if *i* appears before *j* in the ordering. Each agent keeps its own agent view and nogood store. Considering a generic agent *self*, the agent view of *self* is the set of values that it believes to be assigned to its higher priority agents. The nogood store keeps nogoods as justifications of inconsistent values.

When *self* makes an assignment, it sends *Info* messages, to its lower priority agents, informing about its current assignment. When *self* receives a *Back* message, the included nogood is accepted if it is consistent with *self*'s agent view, otherwise it is discarded as obsolete. An accepted nogood is added to *self*'s nogood store to justify the deletion of the value it targets. In standard *ABT*, when *self* cannot take any value consistent with its agent view, because of the original constraints or because of the received nogoods, new nogoods are generated as inconsistent subsets of the agent view, and are sent, as *Back* messages, to the closest agent involved, causing backtracking.

In our *ABT* implementation, we keep a single nogood per removed value. When there is no value consistent with the agent view, a new nogood is generated by resolving all nogoods, as described in [1]. This nogood is sent in a *Back* message.

If *self* receives a nogood mentioning another agent not connected with it, *self* requires to add a link from that agent to *self*. *self* sends an assignment to that agent and after received, a link from the other agent to *self* will exist. The search terminates when achieving quiescence in the network, meaning that a solution has been found because all agents are agree with their current assignment, or when the empty nogood is generated, meaning that the problem is unsolvable.

4 Hybrid Search: ABT-Hyb

In *ABT*, many *Back* messages are obsolete when they arrive to the receiver. *ABT* could save much work if these messages were not sent. Although the sender agent cannot detect those messages that will become obsolete when reaching the receiver, it is possible to avoid sending those which are redundant.

Let *self* be a generic agent. When *self* sends a *Back* message, it performs a new assignment and informs of it to lower priority agents, without waiting to receive any message showing the effect of the *Back* message in higher agents. This can be a source of inefficiency in the following situation. If k sends a *Back* message to j causing a wipeout in j, then j sends a *Back* message to some previous agent i. If j takes the same value as before and sends an *Info* message to k before i changes its value, k will find again the same inconsistency so it will send the same nogood to j in a *Back* message. Agent j will discard this message as obsolete, sending again its value in an *Info*. The process is repeated generating useless messages, until some higher variable changes its value and the corresponding *Info* arrives to j and k.

Based on this intuition, we present *ABT-Hyb*, a hybrid algorithm that combines asynchronous and synchronous elements. *ABT-Hyb* behaves like *ABT* when no backtracking is performed: agents take their values asynchronously and inform lower priority agents. However, when an agent has to backtrack, it does it synchronously as follows. If *self* has no value consistent with its agent view and its nogood store, it sends a *Back* message and enters in a *waiting* state. In this state, *self* has no assigned value, and it

does not send out any message. Any received *Info* message is accepted, updating *self*'s agent view accordingly. Any received *Back* message is rejected as obsolete, since *self* has no value assigned. *self* leaves the waiting state when receiving one the following messages,

1. an *Info* message that allows *self* to has a value consistent with its agent view or,
2. an *Info* message from the receiver of the last *Back* message (the one causing to enter the waiting state) or,
3. a *Stop* message informing that the problem has not solution.

When *self* receives one of these messages, it leaves the waiting state. At this point, *ABT-Hyb* switches to *ABT*.

Like in *ABT*, the problem is unsolvable if during the search an empty nogood is derived. Otherwise, a solution is found when no messages are travelling through the network (i.e.quiescence is reached in the network). No matter the synchronous backtracking, *ABT-Hyb* inherits the good theoretical properties of *ABT*, namely soundness, completeness and termination. To proof these properties, we start with some lemmas.

Lemma 1. *In ABT-Hyb, no agent will stay forever in a waiting state.*

Proof. In *ABT-Hyb*, an agent enters the waiting state after sending a *Back* message to a higher priority agent. The first agent (x_1) in the ordering will not enter in the waiting state because no *Back* message departs from it. Suppose that no agent in $x_1, x_2, \ldots, x_{k-1}$ is waiting forever, and suppose that x_k enters the waiting state after sending a *Back* message to x_j ($1 \leq j \leq k-1$). We will show that x_k will not be forever in the waiting state.

When x_j receives the *Back* message, there are two possible states:

1. x_j is waiting. Since no agent in $x_1, x_2, \ldots, x_{k-1}$ is waiting forever, x_j will leave the waiting state at some point. If x_j has a value consistent with its new agent view, it will send it to x_k in an *Info* message. If x_j has no value consistent with its new agent view, it will backtrack and enter again in a waiting state. This can be done a finite number of times (because there is a finite number of values per variable) before finding a consistent value or discovering that the problem has no solution generating a *Stop* message. In both cases, x_k will leave the waiting state.
2. x_j is not waiting. The *Back* message could be:
   (a) Obsolete in the value of x_j. In this case, there is an *Info* message travelling from x_j to x_k that has not arrived to x_k. After receiving such a message, x_k will leave the waiting state.
   (b) Obsolete not in the value of x_j. In this case, x_j resends to x_k its value by an *Info* message. After receiving such a message, x_k will leave the waiting state.
   (c) Not obsolete. The value of x_j is forbidden by the nogood in the *Back* message, and a new value is tried. If x_j finds another value consistent with its agent view, it takes it and send an *Info* message to x_k, which will leave the waiting state. Otherwise, x_j has to backtrack to a previous agent in the ordering, and enters the waiting state. Since no agent in $x_1, x_2, \ldots, x_{k-1}$ is waiting forever, x_j will leave the waiting state at some point, and as explained in the point 1 above, it will cause that x_k will leave the waiting state as well.

Therefore, we conclude that x_k will not stay forever in the waiting state. □

Lemma 2. *In ABT-Hyb, if an agent is in a waiting state, the network is not quiescent.*

Proof. An agent is in a waiting state after sending a *Back* message. Because Lemma 1, this agent will leave the waiting state in finite time. This is done after receiving an *Info* or *Stop* message. Therefore, if there is an agent in a waiting state, the network cannot be quiescent at least until one of those messages has been produced. □

Lemma 3. *A nogood, discarded as obsolete because the receiver is in a waiting state, will be resent to the receiver until the sender realizes that it has been solved, or the empty nogood has been derived.*

Proof. If an agent k sends a nogood to an agent j that is in a waiting state, this nogood is discarded and agent k enters the waiting state. From Lemma 1, no agent can stay forever in a waiting state, so agent k will leave that state in finite time. This is done after receiving either,

1. An *Info* message from j. If this message does not solve the nogood, it will be generated and resend to j. If it solves it, this nogood is not generated, exactly in the same way as *ABT* does.
2. An *Info* message allowing a consistent value for k. In this case, the nogood is solved, so it is not resent again.
3. A *Stop* message. The process terminates without solution.

Therefore, we conclude that the nogood is sent again until it is solved (either by an *Info* message from j or from another agent) or the empty nogood is generated. □

Proposition 1. *ABT-Hyb is sound.*

Proof. From Lemma 2, *ABT-Hyb* reaches quiescence only when no agent is in a waiting state. From this fact, *ABT-Hyb* soundness derives directly from *ABT* soundness: when the network is quiescent all agents satisfy their constraints, so the current assignments of agents form a solution. If this would not be the case, at least one agent would detect a violated constraint and it would send a message, breaking the quiescence assumption. □

Proposition 2. *ABT-Hyb is complete and terminates.*

Proof. From Lemma 3, the synchronicity of backtracking in *ABT-Hyb* does not cause to ignore any nogood. Then, *ABT-Hyb* explores the search space as good as *ABT* does. From this fact, *ABT-Hyb* completeness comes directly from *ABT* completeness. New nogoods are generated by logical inference from the initial constraints, so the empty nogood cannot be derived if there is a solution. Total agent ordering causes that backtracking discards one value in the highest variable reached by the *Back* message. Since the number of values is finite, the process will find a solution if it exists, or it will derive the empty nogood otherwise.

To see that *ABT-Hyb* terminates, we have to prove that no agent falls into an infinite loop. This comes from the fact that agents cannot stay forever in the waiting state (Lemma 1), and that *ABT* agents cannot be in an endless loop. □

Alternatively to synchronous backtracking, we can avoid resending redundant *Back* messages assuming exponential-space algorithms. Let assume that *self* stores every nogood sent, while it is not obsolete. If a wipe-out occurs in *self*, if the new generated nogood is equal to one of the stored nogoods, it is not sent. This allows *self* not sending identical nogoods until some higher agent changes its value and the corresponding *Info* arrives to *self*. But it requires exponential space, since the number of nogoods generated could be exponential in the number of agents with higher priority than *self*. A similar idea is also found in [16] for the asynchronous weak-commitment algorithm (*AWC*).

5 Experimental Results

We have tested *SCBJ*, *ABT* and *ABT-Hyb* algorithms on the distributed n-queens problem and on random binary problems. Algorithmic performance is evaluated considering computation and communication costs. In synchronous algorithms, the computation effort is measured by the total number of constraint checks (cc), and the global communication effort is evaluated by the total number of messages exchanged among agents (msg).

For the asynchronous algorithms *ABT* and *ABT-Hyb*, computation effort is measured by the number of "concurrent constraint checks" (ccc), which was defined in [8], following Lamport's logic clocks [10]. Each agent has a counter for its own number of constraint checks. The number of concurrent constraint checks is computed by attaching to every message the current counter of the constraint checks of the sending agent. When an agent receives a message, it updates its counter to the higher value between its own counter and the counter attached to the received message. When the algorithm terminates, the highest value among all the agent counters is taken as the number of concurrent constraint checks. Informally, this number approximates the longest sequence of constraint checks not performed concurrently. As for synchronous search, we evaluate the global communication effort as the total number of messages exchanged among agents (msg).

5.1 Implementation Details

Nogood management. To assure polynomial space in *ABT* and *ABT-Hyb*, we keep one nogood per forbidden value. However, if several nogoods are available for each value, it may be advisable to choose the most appropriate resolvent in order to speed up search. With this aim, we implement the following heuristic. If a value is forbidden for some stored nogood, and a new nogood forbidding the same value arrives, we store the nogood with the highest possible lowest variable involved. Notice that, even those nogoods which are obsolete on the value of the receiving variable can be used to select the most suitable nogood with respect to the heuristic.

Saving messages. In asynchronous algorithms, some tricks can be used to decrease the number of messages exchanged. We implement the following:

1. *Value in* AddL. When a new link with agent k is requested by *self*, instead of sending the *AddL* message and assuming this assignment until a confirmation is received, *ABT* include in the *AddL* message the value of x_k recorded in the received nogood. After reception of the *AddL* message, agent k informs *self* of its current value only if it is different from the value contained in the *AddL* message. In this way, some messages may be saved.
2. *Avoid resending same values*. *ABT* can keep track of the last value taken by *self*. When selecting a new value, if it happens that the new value is the same as the last value, *self* does not resend it to $\Gamma^+(self)$, because this information is already known. Again, this may save some messages.

Processing Messages by Packets. *ABT* agents can process messages one by one, reacting as soon as a message is received. However, this strategy of *single-message process* may cause some useless work. For instance, consider the reception of an *Info* message reporting a change of an agent value, immediately followed by another *Info* from the same agent. Processing the first message causes some work that becomes useless as soon as the second message arrives. More complex examples can be devised, causing to waste substantial effort.

To prevent useless work, instead of reacting after each received message, the algorithm reads all messages that are in the input buffer and stores them in internal data structures. Then, the algorithm processes all read messages as a whole, ignoring those messages that become obsolete by the presence of another message. We call this strategy *processing messages by packets*, where a packet is the set of messages that are read from the input buffer until it becomes empty. Somehow, this idea was mentioned in [17] and [21]. In the latter, a comparison between *single-message process* and *processing messages by packets* is presented. However, in none of them a formal protocol for *processing messages by packets* is completely developed.

When an agent processes messages by packets, it reads all messages from its input buffer, and processes them as a whole. The agent looks for any consistent value after its agent view and its nogood store are updated with these incoming messages. To do that, we propose a protocol which requires three lists to store the incoming messages, the *Info*-$List$, *Back*-$List$ and the *AddL*-$List$. In each list is stored the messages of the corresponding type, following the reception order. Each list of messages is processed as follows.

1. *Info*-$List$. First, the *Info*-$List$ is processed. For each sender agent, all *Info* messages but the last are ignored. The remaining *Info* messages update *self* agent view, removing nogoods if needed.
2. *Back*-$List$. Second, the *Back*-$List$ is processed. Obsolete *Back* messages are ignored. *self* stores nogoods of no obsolete messages, and it sends *AddL* messages to unrelated agents appearing in those nogoods. For those messages containing the correct current value of *self*, the sender is recorded in *RemainderSet*.
3. *AddL*-$List$. Third, the *AddL*-$List$ is processed updating $\Gamma^+(self)$ without sending the *Info* message.

lex	SCBJ		ABT		ABT-Hyb	
n	cc	msg	ccc	msg	ccc	msg
10	1,612	170	2,223	740	1,699	502
15	31,761	2,231	56,412	13,978	32,373	6,881
20	6,518,652	306,337	11,084,012	2,198,304	6,086,376	995,902
25	1,771,192	70,336	3,868,136	693,832	1,660,448	271,092
rand	SCBJ		ABT		ABT-Hyb	
n	cc	msg	ccc	msg	ccc	msg
10	965	91	1,742	332	916	238
15	4,120	247	7,697	1,185	4,007	786
20	19,532	921	20,661	4,772	15,720	2,748
25	21,372	746	31,849	6,553	27,055	3,863
min	SCBJ		ABT		ABT-Hyb	
n	cc	msg	ccc	msg	ccc	msg
10	2,800	204	3,716	896	2,988	555
15	35,339	2,210	49,442	11,055	32,303	5,906
20	215,816	10,765	320,278	63,378	165,338	28,686
25	19,949,074	791,089	38,450,786	6,716,505	17,614,330	2,795,319

Table 1. Results for distributed n-queens with lex, random and min-conflict value ordering.

4. Consistent value. Fourth, *self* tries to find a value consistency with the agent view. If a wipe-out happens in this process, the corresponding *Back* message is sent, and a consistent value is searched.
5. *Info* sent. Fifth, *Info* messages containing *self* current value are sent to all agents in $\Gamma^+(self)$ and to all agents in *RemainderSet*. The three lists become empty.

As described in Section 3.2, the search ends when quiescence is reached (i.e. all agents are happy with their current assignment) or an empty nogood is derived.

5.2 Distributed n-queens Problem

The distributed n-queens problem is the classical n-queens problem (locate n queens in an $n \times n$ chessboard such that no pair of queens are attacking each other) where each queen is hold by an independent agent. We have evaluated the algorithms for four dimensions $n = 10, 15, 20, 25$. In Table 1 we show the results in terms of constraint checks/concurrent constraint checks and total number of messages exchanged, averaged over 100 executions with different random seeds (ties are broken randomly). Lexicographic (static) variable ordering has been used for *SCBJ*, *ABT*, and *ABT-Hyb*. Three value ordering heuristics have been tested *lex* (lexicographic), *rand* (random) and *min* (min-conflicts) [9] on all the algorithms. Given that an exact *min* computation requires extra messages, we have made an approximation, which consists of computing the heuristic assuming initial domains. With this approximation, the *min* value ordering heuristic can be computed in a preprocessing step.

We observe that the random value ordering provides the best performance for every algorithm and every dimension tested. Because of that, in the following we concentrate our analysis on the results of random value ordering.

Considering the relative performance of asynchronous algorithms, *ABT-Hyb* is always better than *ABT*, in both number of concurrent constraint checks and total number of messages. It is relevant to scrutinize the improvement of *ABT-Hyb* over *ABT* with respect to the type of messages. In Table 2, we provide the total number of messages per message type for *SCBJ*, *ABT* and *ABT-Hyb* with random value ordering. In *ABT-Hyb* the number of obsolete *Back* messages decreases in one order of magnitude with respect the same type of messages in *ABT*, causing *ABT-Hyb* to improve over *ABT*. However, this improvement goes beyond the savings in obsolete *Back* messages, because *Info* and *Back* messages decrement to a larger extent. This is due to the following collective effect. When an *ABT* agent sends a *Back* message, it tries to get a new consistent value without knowing the effect that backtracking causes in higher priority agents. If it finds such a consistent value, it informs to lower priority agents using *Info* messages. If it happens that this value is not consistent with new values that backtracking causes in higher priority agents, these *Info* messages would be useless, and new *Back* messages would be generated. *ABT-Hyb* tries to avoid this situation. When an *ABT-Hyb* agent sends a *Back* message, it waits until it receives notice of the effect of backtracking in higher priority agents. When it leaves the waiting state, it tries to get a new consistent value. At this point, it knows some effect of the backtracking on higher priority agents, so the new value will be consistent with it. In this way, the new value has more chance to be consistent with all higher priority agents, and the *Info* messages carrying it will be more likely to make useful work.

Considering the performance of synchronous vs. asynchronous algorithms, we compare *SCBJ* against *ABT-Hyb* with random value ordering. In terms of computation effort (constraint checks) *SCBJ* performs better than *ABT-Hyb* for $n = 25$ and worse for $n = 20$, with very similar results for $n = 10, 15$. In terms of communication cost, *SCBJ* uses less messages than *ABT-Hyb* for the four dimensions tested. This comparison should be qualified, noting that the lenght of *Info* messages differ from synchronous to asynchronous algorithms. In *SCBJ*, an *Info* message contains the partial solution which could be of size n, while in *ABT-Hyb* an *Info* message contains a single assignment of size 1. Assuming that the communication cost depends more crucially on the number of messages than on their length, we conclude that *SCBJ* is more efficient in communication terms than *ABT-Hyb*. Considering both aspects, computation effort and communication cost, *SCBJ* seems to be the algorithm of choice for the n-queens problem.

5.3 Random Problems

Uniform binary random CSPs are characterized by $\langle n, d, p_1, p_2 \rangle$ where n is the number of variables, d the number of values per variable, p_1 the network *connectivity* defined as the ratio of existing constraints, and p_2 the constraint *tightness* defined as the ratio of forbidden value pairs. We have tested random instances of 16 agents and 8 values per agent, considering three connectivity classes, sparse (p_1=0.2), medium (p_1=0.5) and dense (p_1=0.8).

rand	SCBJ		ABT			ABT-Hyb		
n	Info	Back	Info	Back	Obsol	Info	Back	Obsol
10	55	36	251	81	24	195	43	2
15	146	101	901	284	91	649	137	10
20	539	382	3,612	1,160	408	2,293	455	38
25	452	294	5,027	1,526	520	3,240	623	50

Table 2. Number of messages exchanged by *SCBJ*, *ABT* and *ABT-Hyb* per message type, for the distributed n-queens problem with random value ordering.

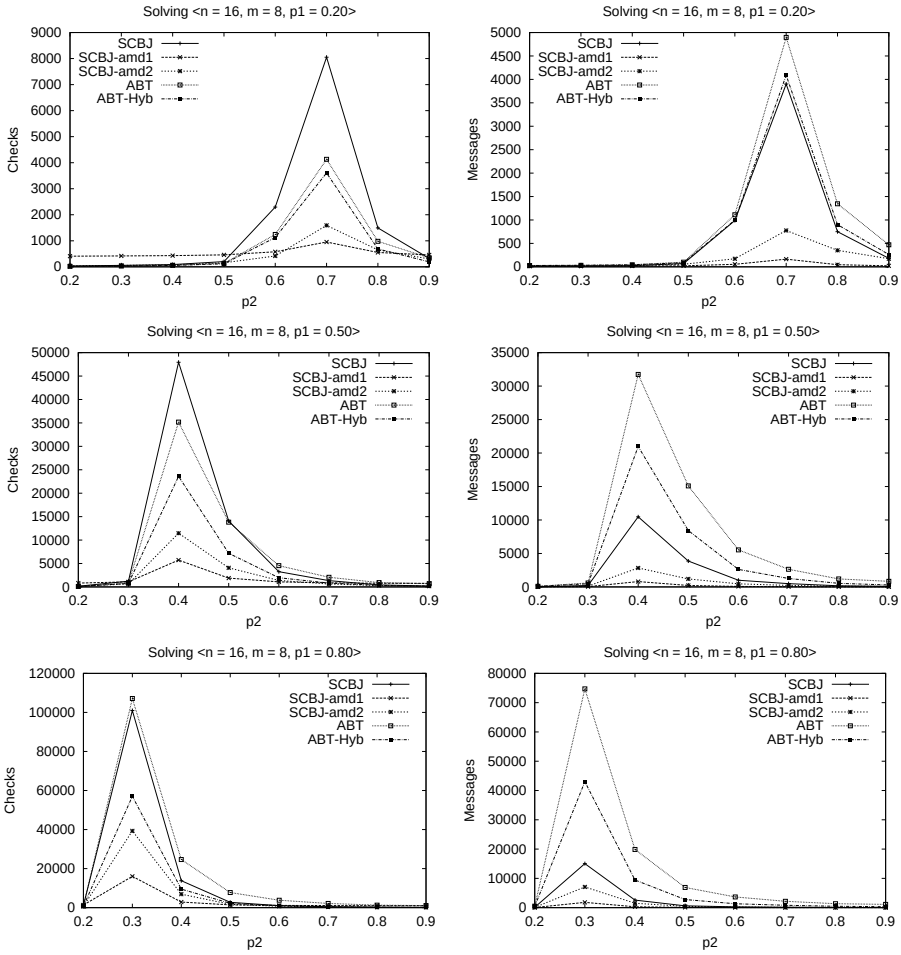


Fig. 1. Constraint checks and number of messages for *SCBJ*, *SCBJ-amd1*, *SCBJ-amd2*, *ABT* and *ABT-Hyb* on binary random problems.

In a synchronous algorithm, it is simple to implement some heuristic for dynamic variable ordering. Considering the heuristic of minimum domain, an exact computation

rand	SCBJ		SCBJ-amd1		ABT				ABT-Hyb			
p_2	Info	Back	Info	Back	Info	Back	Obsol	Link	Info	Back	Obsol	Link
0.20	2,647	1,254	100	63	3,587	1,310	320	26	3,141	949	53	24
0.50	6,913	3,556	477	321	24,725	7,025	2,336	40	17,650	3,335	321	37
0.80	9,761	5,265	1,052	758	58,283	16,432	6,497	19	37,046	5,956	755	18

Table 3. Number of messages exchanged by *SCBJ*, *SCBJ-amd1*, *ABT* and *ABT-Hyb* per message type, for random binary problems with random value ordering.

requires extra messages. To avoid this, we have implemented the following approximations,

- AMD1. Each agent computes the interval $[min_i, max_i]$ of the minimum and maximum number of inconsistent values in the domain of every unassigned variable x_i with the partial solution. This interval is included in the *Info* message. Then, the next variable to be assigned is chosen as follows: (i) if there is x_i such that $min_i \geq min\{d, max_j\}, \forall x_j$ unassigned, selects x_i (where d is the domain size); (ii) otherwise, selects the variable with maximum max_j.
- AMD2. This approach only computes the current domains of the unassigned variables after *Back* messages. When *self* sends a *Back* message to x_j, instead of sending it directly to x_j it goes chronologically. Each intermediate variable recognizes that it is not its destination, and it includes the current size of its domain in the message. This messages ends in x_j and after assigning it, the minimum domain heuristic without considering the effect of x_j's assignment can be applied on the subset of intermediate variables. It causes some extra messages, but its benefits pay-off.

In Figure 1, we report results averaged over 100 executions for *SCBJ*, *SCBJ-amd1*, *SCBJ-amd2*, *ABT* and *ABT-Hyb*, with random value ordering.

Considering synchronous algorithms, approximating minimum domains heuristic is always beneficial both in computation effort and in communication cost. Consistently in the three classes tested, the approximation *amd1* provides better results than *amd2*, both in terms of checks and messages. When using *amd1*, the baseline of constraint checks is not zero, due to the heuristic computation done as a preprocessing step.

Considering asynchronous algorithms, we observe again that *ABT-Hyb* is always better than *ABT* for the three problem classes, in both computation effort and communication cost. We believe that this is due to the effect already described for the distributed n-queens problem. This is confirmed after analyzing the number of messages per message type of Table 3.

Comparing the performance of synchronous vs. *ABT-Hyb*, we observe the following. In terms of computation effort (constraint checks), *SCBJ* is always worse than *ABT-Hyb*, and *SCBJ* is often the worst algorithm (except in the $\langle 16, 8, 0.8 \rangle$ class, where it is the second worst). This behaviour changes dramatically when adding the minimum domain heuristic approximations: *SCBJ-amd1* and *SCBJ-amd2* are the best and second best algorithms in the three classes tested, and they are always better than *ABT-Hyb*.

min	SCBJ		SCBJ-amd1		SCBJ-amd2		ABT		ABT-Hyb	
p_1	cc	msg	cc	msg	cc	msg	ccc	msg	ccc	msg
0.20	7,100	3,277	907	153	1,811	687	3,771	4,006	3,448	3,535
0.50	44,024	9,367	5,637	783	11,677	2,669	30,719	26,840	22,227	19,141
0.80	102,153	15,111	16,206	1,843	40,449	7,142	101,492	70,033	58,428	43,459

Table 4. Results near of the pick of difficulty on binary random classes $\langle n = 16, m = 8 \rangle$ with min-conflict value ordering.

Regarding communication costs, synchronous algorithms are always better than asynchronous ones: consistently in the three classes tested, *SCBJ-amd1*, *SCBJ-amd2* and *SCBJ* are the three best algorithms (in this order). Again, the addition of minimum domain approximations is very beneficial. As mentioned in Section 5.2, $Info$ messages are of different sizes in synchronous and asynchronous algorithms. Under the same assumptions (communication costs depends more on the number of messages exchanged than on their length), we conclude that for solving random binary problems, *SCBJ-amd1* is the algorithm of choice.

We have also tested the three problem classes using the min-conflict value ordering. Results appear in Table 4 for the peak of maximum difficulty. We observe a minor but consistent improvement of all the algorithms with respect to the random value ordering. In this case, the relative ranking of algorithms obtained with random value ordering remains, *SCBJ-amd1* being the algorithm with the best performance.

We have also tested *ABT* and *ABT-Hyb* with random message delays. This issue was raised first in [5], and subsequently in [21]. Preliminary results show that *ABT* decreases performance and also *ABT-Hyb* does, but to a lesser extent. This last algorithm exhibits a more robust behavior in presence of random delays. It is worth noting that synchronous algorithms do not increase the number of checks or messages in presence of delays.

6 Conclusions

We have presented three algorithms, one synchronous *SCBJ*, one asynchronous *ABT* and one hybrid *ABT-Hyb*, the two first being already known. We have proposed *ABT-Hyb*, a new algorithm that combines asynchronous and synchronous elements. *ABT-Hyb* can be seen as an *ABT*-like algorithm where backtracking is synchronized: an agent that initiates backtracking cannot take a new value before having some notice of the effect of its backtracking. This causes a kind of "contention effect" in backtracking agents. Their decisions tend to be better founded than the corresponding decisions taken by *ABT* agents, and therefore they are more likely to succeed. *ABT-Hyb* inherits the good theoretical properties of *ABT*: it is sound, complete and terminates.

We have implemented *ABT* and *ABT-Hyb* with a strategy for processing messages by packets, together with some simple ideas to improve performance. On *SCBJ* we have proposed two approximations for the minimum domain heuristic. Empirically we have observed that *ABT-Hyb* clearly improves over *ABT*, in both computation effort and communication costs. Comparing *SCBJ* with *ABT-Hyb*, we observe that *SCBJ* always requires less messages than *ABT-Hyb*, for both problems tested. Considering computation effort, *SCBJ* requires a similar effort as *ABT-Hyb* in distributed n-queens, while

SCBJ requires more effort than *ABT-Hyb* for binary random problems. However, when enhanced with minimum domain approximation for dynamic variable ordering, *SCBJ-amd1* is the best algorithm in terms computation effort and in number of messages exchanged. Grouping these evidences together, we conclude that synchonous algorithms enhanced with some minimum domain approximation are globally more efficient than asynchronous ones. This does not mean that synchronous algorithms should always be preferred to asynchronous ones, since they offer different functionalities (synchronous algorithms are less robust to network failures, privacy issues are not considered, etc.). But for applications where efficiency is the main concern, synchronous algorithms seems to be quite good candidates to solve DisCSP.

References

1. Bessière C., Maestre A. and Meseguer P. Distributed Dynamic Backtracking. *IJCAI-01 Workshop on Distributed Constraint Reasoning*, 9-16, Seattle, USA, 2001.
2. Bitner J. and Reingold E. Backtrack programming techniques. *Communications of the ACM*, **18**:11, 651–656, 1975.
3. Collin Z., Dechter R., Shmuel K. On the Feasibility of Distributed Constraint Satisfaction. *In Proc. of the 12th International Joint Conference on Artificial Intelligence, IJCAI-91*, 318–324, 1991.
4. Dechter R. and Pearl J. Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence*, **34**, 1–38, (1988).
5. Fernandez C., Bejar R., Krishnamachari Gomes, K. Communication and Computation in Distributed CSP Algorithms. *In Proc. Principles and Practice of Constraint Satisfaction Programming (CP-2002)*, 664–679, Ithaca NY, USA, July, 2002.
6. Hamadi Y., Bessière C., Quinqueton J. Backtracking in Distributed Constraint Networks. *In Proc. of the 13th ECAI*, 219–223, Brighton, UK, 1998.
7. Hirayama K. and Yokoo M. The Effect of Nogood Learning in Distributed Constraint Satisfaction. *In Proceedings ICDCS'00*, 169–177. 2000
8. Meisels A., Kaplansky E., Razgon I., Zivan R. Comparing Performance of Distributed Constraint Processing Algorithms. *AAMAS-02 Workshop on Distributed Constraint Reasoning*, 86–93, Bologna, Italy, 2002.
9. Minton S. and Johnston M. and Philips A. and Laird P. Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence*, **58**, 161–205, 1992.
10. Lamport L. Time, Clock, and the Ordering of Evens in a Distributed System. *Communications of the ACM*, **21(7)**, 558–565, 1978.
11. Prosser, P. Hybrid Algorithm for the Constraint Satisfaction Problem. *Computational Intelligence*, **9**, 268–299, 1993.
12. Prosser, P. Domain Filtering can Degrade Intelligent Backtracking Search. *Proc. IJCAI*, 262–267, 1993.
13. Silaghi M.C., Sam-Haroud D., Faltings B. Asynchronous Search with Aggregations. *In Proc. of the 17th AAAI*, 917–922, 2000.
14. Silaghi M.C., Sam-Haroud D., Faltings B. *Hybridizing ABT and AWC into a polynomial space, complete protocol with reordering*. Tech. Report EPFL, 2001.
15. Yokoo M., Durfee E.H., Ishida T., Kuwabara K. Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. *In Proc. of the 12th International Conference on Distributed Computing System*, 614–621, 1992.

16. Yokoo M. Asynchronous Weak-commitment Search for Solving Distributed Constraints Satisfaction Problems. *In Proceeding of the First International Conference on Principles and Practice of Constraint Programming (CP-1995)* 88–102, 1995.
17. Yokoo M., Durfee E.H., Ishida T., Kuwabara K. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Trans. Knowledge and Data Engineering* **10**, 673–685, 1998.
18. Yokoo M., Ishida T. Search Algorithms for Agents. In *Multiagent Systems*, G. Weiss editor, Springer, 1999.
19. Yokoo M. *Distributed Constraint Satisfaction*, Springer, 2001.
20. Yokoo M. , Suzuki, Hirayama K. Secure Distributed Constraint Satisfaction: Reaching Agreement without Revealing Private Information. *In Proc. of the 8th CP*, 387–401, 2002.
21. Zivan, R. and Meisels, A. *Synchronous and Asynchronous Search on DisCSPs.* In Proc. of EUMAS-2003, Oxford, UK, 2003

E-PRIVACY REQUIREMENTS FOR DISTRIBUTED E-SERVICES

Solange GHERNAOUTI-HÉLIE, Mohamed Ali SFAXI
Ecole des HEC, Université de Lausanne
CH – 1015 Lausanne
{sgh, mohamedali.sfaxi}@hec.unil.ch

Privacy is a real concern for e-services users. In digital environments (digital information, dematerialization of actors, computers and networks operating mode), technologies don't preserve, in native mode, user's privacy. Cyber crime, Eavesdropping, theft of strategic information, commercial proposition, etc. could give economic advantage to unfair competitors. The privacy concept in the cyberspace looks like a luxury. Nowadays, cyber-criminal, hacker or cracker, what ever we call them, represent a real threat to the society, causing malicious harm to ICT resources, to individuals, organizations and states. Police investigations in information and communication environments are more and more necessary and frequent. Cyber crime, as the increased justice and police investigations needs affect effective e-privacy solutions.
Our paper presents the actual privacy concern over the Internet, describes in details our enterprises privacy study, and analyzes the benefits and the limits of P3P approach. Finally, recommendations are proposed to preserve privacy and satisfy security objectives for e-services.

To see how enterprises deal with e-privacy concern, we analyze, through a study, several privacy criteria in enterprise privacy policies. The sample is composed of 200 websites (including amazon.com, nfl.com, nba.com...) taken at random. The main criteria are about the use of personal data, security issues, cookies manipulation, etc... We propose a graphic representation to visualize main criteria that enterprises must bet on and how they can improve their privacy policy. The result shows that the majority of enterprises don't say anything about the notification of their users when the privacy policy changes, how they deal with users' IP addresses, the presence of third parties and the fact of sharing or selling personal data to other entities.
We can easily notice that the needs to privacy and security are not yet well identified and satisfied for individuals and organizations. To contribute to satisfy these needs, the World Wide Web Consortium (W3C) tries, with the Platform for Privacy Preferences (P3P) approach, to ensure privacy. The Platform for Privacy Preferences (P3P) is indented to be a simple, automated way for Internet users to have more control over the use of personal information on visited website. In fact, P3P imposes that privacy policies covering a page are easy to find so that users can find the policy from the site they are visiting and the policies of other websites that are contributing to that page. In addition,

the privacy policies, using P3P, are easy to understand and do not consist of pages of legalese. However, P3P has its limits for guaranteeing privacy over the Internet. As, the W3C is a specification setting organization; it does not have the ability to make a public policy guarantee that its specifications be followed over the Internet. This specification needs to be used in concert with effective legislation, strategic policy and other privacy enhancing tools.

From the study done, many recommendations can be extracted. Some recommendations are related to the form of the privacy policy published on the WEB by enterprises, the others affect directly the contents of these policies. The enterprise must present the privacy policy document in a readable and ergonomic form. The policy document must be clear (with a medium police size and paragraph separators) and easy to understand by all users (not a complex model full of technical terms). The published policy must give at least an answer to the users' needs of understanding privacy concern. Simple and clear answers must obligatorily be given to crucial privacy questions. In addition, the use of encryption can solve many issues. In fact, Encryption is a cheap way to obtain integrity, authentication and confidentiality. Small and Medium size enterprises can use these technique to reach security objectives.

To conclude, legal framework and security solutions must be developed to satisfy e-privacy needs taking into account the respect of fundamental human rights. In the meantime enterprises have to implement available e-security and e-privacy solutions. Most of them are accessible and enough effective to satisfy current needs of enterprises and organizations.

Keywords:
E-privacy, e-security architecture, e-business, white collar crime, police and justice investigations, e-economy, information society.

The DDAC4 Algorithm for Arc-Consistency Enforcement in Dynamic and Distributed CSP *

Georg Ringwelski

Cork Constraint Computation Center, University College Cork, Ireland
g.ringwelski@4c.ucc.ie

Abstract. This paper presents the new DDAC4 algorithm for dynamic arc consistency enforcement in Distributed Constraint Satisfaction Problems. The algorithm is an adaptation of the well known AC-4 algorithm to system settings where constraints can be added and deleted in concurrent processes. It is the first algorithm for arc-consistency enforcement in this system setting. Arc-consistency is achieved whenever the overall system reaches quiescence after a constraint is added or deleted.

1 Introduction

Constraint propagation has become one of the most successful methods for constraint processing. If applied as a preprocessing step or during search this technique can significantly reduce the required effort of tree-based search methods to solve Constraint Satisfaction Problems (CSP). However, its applicability is currently often restricted to monolithic and/or static problems. In todays software systems, for example in the field of global computing, these preconditions can usually not be met. Applications that use the Internet as a platform of constraint satisfaction are often required to be able to adapt dynamically to newly emerging knowledge in a distributed and completely asynchronous manner. It is often not possible and very rarely desirable to gather information centrally for constraint processing. Consequently there is a strongly increasing need for methods to perform dynamic and distributed constraint processing.

In this paper I describe the new DDAC4 algorithm which implements the successful constraint propagation technique of arc-consistency enforcement for a distributed and dynamic setting. To the best of my knowledge it is the first algorithm that provides this functionality.

2 Preliminaries

2.1 Constraint Satisfaction Problems

A Constraint Satisfaction Problem (CSP) is given by a triple (C, X, D) where C is a set of constraints, $X = \{x_1, ..., x_n\}$ is a set of variables and $D = \{D_1, ..., D_n\}$

* This work has received support from the Embark Initiative of the Irish Research Council of Science Engineering and Technology under Grant PD2002/21 and from Science Foundation Ireland under Grant 00/PI.1/C075

are their respective domains. In a binary CSP each constraint $C_{ij} \in C$ is associated to a binary relation $sem(C_{ij}) \subseteq D_i \times D_j$ in the Cartesian Product of the domains of the constraint's two variables x_i and x_j. In this paper I denote a constraint c over variables x and y with an associated relation s by the term $c(x, y, s)$. A constraint C_{ij} (i.e. an arc in the constraint graph associated to a CSP [4]) is said to be arc-consistent if and only if for each value $d_i \in D_i$ at least one value $d_j \in D_j$ exists such that $(d_i, d_j) \in sem(C_{ij})$. A binary CSP (resp. its associated graph) is said to be arc-consistent if and only if all its constraints are arc-consistent. A detailed introduction to CSP can be found in [4].

2.2 Multi-Agent-Systems

Multi-Agent-Systems (MAS) are given by a set of agents which are executed in concurrent processes and communicate by messages. Each agent can send any other a message provided it knows its address. In this paper, I use the following communication primitives for MAS:

send(r,msg) sends a message **msg** to the agent **r**. The message can be any string;

broadcast(msg) sends **msg** to each known agent;

behavior receive(s,msg) provides an agent-behavior, i.e. a procedure which is called as soon as a certain message **msg** from a sender **s** arrives. Both, **s** and **msg** can be or can contain variables which will be unified with the received values. Thus **msg** is usually a pattern which an incoming message must match in order to trigger the behavior. The pattern is usually a structured term which identifies the kind of message received, and its arguments are variables which are unified with the actual information contained in the message;

blockingReceive(s,msg) waits for a message **msg** from the agent **s**. It halts the execution of its thread until a message which fits the pattern is received. Please note that while a thread is halted, other threads (i.e. behaviors) of the same agent may be active, and receive and process messages.

Furthermore, I assume that every message evetnually arrives, and that between every directed pair of agents the order in which messages are sent is the same as that in which they are received. Whenever an agent refers to itself it will use the synonym `self`. A more detailed introduction to distributed computing can be found in [2].

3 Background

Finding solutions for arc-consistent CSPs can be much easier than finding solutions to equivalent (wrt. the set of solutions) CSPs which are not arc-consistent. Thus the enforcement of arc-consistency both as a preprocessing step and during search has a long tradition in CSP research.

3.1 AC-4

One method for the enforcement of arc-consistency is the algorithm AC-4 [6]. The input of that algorithm is any binary CSP which AC-4 transforms into an equivalent arc-consistent binary CSP. The idea of the algorithm is that in an arc-consistent CSP, each value x in each variable domain must have at least one value y in the domain of another variable such that (x, y) is an element of the relation associated to any constraint. The value y is called the support of x. To be more precise: the set of all supports of a value $d_i \in D_i$ are elements of the set $support(d_i) = \{d_j | \exists C_{ij} \in C, d_j \in D_j : (d_i, d_j) \in sem(C_{ij})\}$. AC-4 uses a data structure *counter* which associates the number of values which use d_i as support to each value d_i, i.e. $counter(d_i) = |\{d_j \in D_j | d_i \in support(d_j)\}|$. Whenever the counter of a value d_i is zero, it does not support any other value via any constraint and can thus be removed from its domain D_i without reducing the set of solutions to the CSP.

AC-4 computes the counter for each value in a CSP and deletes any value if its counter is zero. To do this, the algorithm consists of two stages: first the data structures *support* and *counter* are set up for each value and then the values are iteratively pruned until no more zero counters exist. A more detailed description of the algorithm can be found in [6, 4].

3.2 DisAC-4

The distributed version of AC-4 [7] is implemented in a set of "worker"-agents and a central "controller"-agent. All agents communicate by messages and the same assumptions concerning the protocol I described in section 2.2 are made. The controller is used to detect termination of the algorithm and to stop all workers if one has detected that no solutions exist. The algorithm terminates as soon as all workers have reached quiescence, i.e. there are no messages to be processed and all computations are finished. The actual arc-consistency enforcement is performed by the workers. The main difference to the centralized AC-4 is that the knowledge stored in the data structures *support* and *counter* is partitioned among the agents. Each worker w is related to a set of variables $X_w \subseteq X$ which are located in its process. The worker w stores the sets $counter(x)$ and $support(x)$ for each $x \in X_w$.

DisAC-4 works in two similar stages as AC-4. It differs from AC-4 by keeping track of all values that can be pruned in two lists *list* and *toSendList*. After the first stage the *toSendList* is broadcast to other workers and the *list* is processed in the loop that implements the second stage of the algorithm similar to AC-4. Whenever the *list* is empty, the *toSendList* is broadcast to other workers. Upon the receipt of such a *toSendList* the workers add all its elements to their local *list* to check against their local knowledge. For a more detailed description of the algorithm, please refer to [7].

3.3 DnAC-4

Like AC-4 and DisAC-4, the dynamic DnAC-4 algorithm [1] uses the data structures *counter* and *support* to find values that must be pruned to achieve arc-consistency. These structures are incrementally updated whenever a constraint is added.

In order to be able to relax constraints an additional structure $justif$ which stores justifications for value prunings is incrementally processed. It associates to each variable x_j a removed variable-value pair (x_i, d_i) such that the constraint between x_i and x_j caused the removal of d_i. With this storage of justifications it may be possible that justifications build circular dependencies and will not be "well-founded" as described in [1]. When a constraint is deleted values that have potentially been pruned due to that constraint are re-inserted in their domain. Then the new values are checked against the other constraints and may be excluded again if another constraints prohibits them.

4 The distributed algorithm DDAC4

Like DisAC-4, the DDAC4 algorithm uses one controller and several worker agents. The controller starts and terminates the workers. However, the algorithm is not terminated as quiescence is reached, but only if this is explicitly requested. Whenever quiescence is reached in DDAC4, arc-consistency will be present in the currently existing distributed CSP. In contrast to the DisAC-4 algorithm, I assume for DDAC4 in this paper that every worker represents exactly one variable. The worker agent and the variable are considered to be the same and refer to itself with `self`. This one-to-one topology is no restriction to generality as the algorithm can easily be adapted to the general case by extending the internal data structures `counter` and `support` (see below) with another dimension for the respective local variable. A more significant difference to DisAC-4 and AC-4 is that DDAC4 is not implemented in two stages: the initialization phase is omitted. As in DnAC-4 the necessary computations of this phase are performed whenever a new constraint is added.

The main difference to DnAC-4 is that the justifications for pruned values are handled differently. Most importantly, the justifications are not variables, but constraints and multiple justifications for the removal of every pruned value can be stored. Consequently, DDAC4 does not have to consider the "well-foundedness" as described in [1]. Furthermore, this allows the relaxation of constraints in one phase and omit the need for a subsequent propagation phase to ensure arc-consistency as for instance DnAC-4 uses it.

All worker agents run the same code using different locally stored data. Each worker uses the following private data structures, procedure and behaviors:

`domain` the set of all allowed values of `self`;
`initDomain[x]` , the domain of variable `x` as it was when the respective agent was created;

`list` a set of triples (var, val, j). As in DisAC-4 this set contains values val of variables var that can be pruned. The justification for this is the constraint j, which may be directly or indirectly responsible for this pruning;

`toSendList` a similar set to `list` which aggregates relevant information to be sent to adjacent workers;

`knownAsDeleted` a set which contains all constraints that `self` knows to be deleted;

`constraints` the set of all constraints over `self`;

`neighbors` the set of all adjacent agents;

`support[x][y]` the set of values from the domain of `self` that support variable-value pairs `(x,y)` of other agents as known from DisAC-4;

`counter[x][y]` the number of values of variable `y` that use the value `x` of `self` as *support*;

`removeList` a list of triples (var, val, j) where `counter[`val`][`var`]` was reduced because of the propagation of j.

procedure sendList() sends the `toSendList` to all `neighbors` and assigns it to the empty list afterwards;

behavior receive(sender, addConstraint(c)) invokes local procedure **addConstraint(c)** and sends `initialDomain[self]` to **sender**;

behavior receive(sender, deleteConstraint(c)) invokes local procedure **deleteConstraint(c)**

behavior receive(sender, relax(c)) invokes local procedure **relax(c)**

Each worker provides methods to add and to delete constraints. The constraint addition method and its propagation are presented in Algorithms 1 and 2 and the deletion method, including the problem relaxation, is shown in Algorithm 3.

I start with the description of Algorithm 1. First, the algorithm makes sure that the new constraint was not deleted before and that there is no other constraint between the two respective variables. The first prohibits the re-addition of constraints after deleting it, such that a new constraint has to be created in such a situation. The second prohibits several constraints between the same pair of variables. Multiple constraints between same variables are also not supported in Dis-AC4 and other DisCSP algorithms. However, the DDAC4 algorithm can be easily extended to support multiple constraints between same variable pairs by adding a further dimension to store the constraint in the support and counter arrays. Then, the new constraint c is integrated properly in the system by adding it to the set `constraints` of `self` and the agent A which holds the other variable. Furthermore, the address of A respectively `self` is added to the set of `neighbors` of `self` respectively A. The manipulation of agent A is performed by sending it a `newConstraint` message. After that the `self` waits for the reply containing the initial (and thus invariant) domain of A. Then the algorithm checks for all combinations of values of c's variables if they are consistent. If they are consistent it updates the `counter` and `support` arrays. If the counter of a local value v is zero after checking all values of the remote variable, v can be pruned as it does not support any other value. Thus it is removed from the variable domain and a respective triple is added to `list, toSendList` and `removedList`.

Finally the `toSendList` is sent to the `neighbors`. This is an improvement to the corresponding method in the DisAC-4 algorithm. In DisAC-4 the `toSendList` is broadcast in the system, i.e. sent to *all* agents. With this slight change the required number of messages coul;d be reduced significantly. This efficiency improvement does not change the outcome of the algorithm, since all propagation has to begin with the neighboring variables anyway.

Algorithm 1: Worker method for constraint addition.

```
   procedure addConstraint(c(x,y,sem)) begin
       if this = x then
           other := y;
       else
           other := x;
1      if other ∈ neighbors ∪ knownAsDeleted then
           return;
2      constraints := constraints ∪ c;
       neighbors := neighbors ∪ other;
       send(other,newConstraint(c));
       blockingReceive(other,domain(otherDomain));
       initDomain[other] := otherDomain;
3      foreach d ∈ initDomain[this] do
           foreach d' ∈ otherDomain do
               if (this = x AND (d,d') ∈ sem) OR (this = y AND (d',d) ∈ sem)
               then
                   counter[d][other] := counter[d][other]+1 ;
                   support[other][d'] := support[other][d'] ∪ {d}
           if counter[d][other] = 0 then
               list := list ∪ {(this,d,c)};
               toSendList := toSendList ∪ {(this,d,c)};
4              removedList := removedList ++ {(this,d,c)};
5              domain := domain \{d};
6      sendList();
   end
```

Algorithm 2 implements constraint propagation throughout the CSP. When a worker receives a `propagate` message, it performs similar steps as for the integration of a new constraint. The respective behavior is represented in Algorithm 2. First it adds all the triples received from another agent to its local `list`. Then, for each element (var, val, j) of the `list`, it performs the following propagation steps: it retrieves from the `support` array all values v of the local variable that support the value val for variable var; if j is not known to be deleted, the worker decrements the `counter` for v and stores a respective triple in `removedList`. If the counter reaches zero and v is in the current domain, then

v can be pruned as it was described in the previous paragraph. Finally, the newly updated `toSendList` is sent to all neighbors.

Algorithm 2: The propagation behavior of DDAC4 workers.

```
   behaviour receive(sender,propagate(l)) begin
       list := list ∪ l ;
       while list ≠ ∅ do
           remove any (var, val, j) from list;
           foreach v ∈ support[var][val] do
1              if j ∉ knownAsDeleted then
                   counter[v][var] := counter[v][var] -1;
2                  removedList = (var, v, j) ++ removedList;
                   if counter[v][var] = 0 AND v ∈ domain then
                       list := list ∪ {(this, v, j)};
                       toSendList := toSendList ∪ {(this, v, j)};
3                      domain := domain \{v};
4      sendList();
   end
```

The constraint deletion method each worker provides and the necessary behaviors to relax the CSP, i.e. to de-propagate the deleted constraint, are presented in Algorithm 3. Upon a call to the **deleteConstraint** method, the deleted constraint is removed from the local set of constraints and its other variable is removed from the set of neighbors. The same is done asynchronously in the remote agent that hosts the constraint's other variable by sending it a `delConstraint` message. After that, the `support` and `counter` arrays are adapted inversely to the way they were changed when the constraint was added. Finally a `relax` message is broadcast to all agents (including `self`) to make them relax their variables for the deleted constraint. The method `relax` is invoked upon receipt of a `relax` message with a reference to the deleted constraint. This method adds the newly deleted constraint to the set `knownAsDeleted`. Then it checks for all values in the `removedList` to find out whether they were removed because of the deleted constraint. Is so, the respective triple is removed from that list and the counter is incremented. If the counter reaches one, there are no justifications left to exclude the value and it is added to the `domain`.

4.1 Difficulties and Pitfalls

Concurrent Propagation and Relaxation In a dynamic and distributed system, constraints can be added and deleted concurrently in separate processes. These concurrent events may yield consequences for common variable domains which can lead to non-determined results and even non-terminating runs. Consider for example three variables x, y and z which all have the domain $\{1, 2, 3\}$

Algorithm 3: Worker method and behavior for constraint deletion.

```
   procedure deleteConstraint(c(x,y,s)) begin
       if c ∉ constraints then
           return ;
       constraints := constraints \{ c};
       if this = x then
           other := y;
       else
           other :=x ;
       neighbors := neighbors \{other};
       send(other,deleteConstraint(c));
       foreach d ∈ initDomain[this] do
           foreach d' ∈ initDomain[other] do
               counter[d][other] := counter[d][other] −1;
               support [other][d'] := support[other][d']\{d};
1      broadcast(relax(c));
   end
   procedure relax(c) begin
       if c ∈ knowAsDeleted then
           return ;
2      knownAsDeleted := knownAsDeleted ∪{c};
       foreach (var, val, j) ∈ removedList do
           if j = c then
               counter[val][var] := counter[val][var] +1;
               removedList := removedList − (var, val, j);
3              if counter[val][var] = 1 then
                   domain := domain ∪{val};
   end
```

and a constraint $c_1(x, y, \{(1, 2)\})$ which has been posted. If in this situation the concurrent events $addConstraint(c_2(x, z, \{(2, 2)\}))$ and $deleteConstraint(c_1)$ occur, the outcome of the constraint processing is non-determined: the value 3 may or may not be in the domain of x, depending on the order of the processing of the events. The problem is that due to the addition of c_2 the value is pruned while due to the deletion of c_1 it is added at the same time. Clearly, the correct (arc-consistent) result should exclude 3 from x, but this may not be the result of the concurrent algorithm.

DDAC4 prevents non-determined results by storing justifications for every pruned value and by allowing values to be "pruned multiply". In the data structure `removedValues` the algorithm keeps track of each value that must be pruned and stores the respective constraint which justifies this pruning. Please note that this list may contain several entries regarding the same value. `removedValues` contains all justifications for each excluded value of the initial variable domain. Thus, there can be several justifications for each pruned value. Unlike AC-4

or DisAC-4 the `counter` values can consequently be below zero. Whenever the `counter` passes zero the `domain` is manipulated: when it changes from one to zero, a value is pruned (as in AC-4) and when it changes from zero to one, a value is added. As soon as there is at least one justification to remove a value, it is actually removed from the variable's domain. However, when a value is to be added due to a relaxation, this is only performed if there are no more justifications for its pruning (Alg.3, line 3).

Message-Waves take Shortcuts Another problem occurring in the dynamic and distributed setting arises from message delays in the concurrent execution. Assume an application which adds and deletes a constraint within a short time: the processing of the constraint addition may not be globally completed before the same constraint is deleted. With "not globally completed" I mean that the wave of the respective `propagate` messages has not yet reached every relevant variable. Consequently, there may be `propagate` and `relax` messages regarding the same constraint to be processed in parallel. From the general assumption that messages always arrive in the same order they were sent, it *cannot* be deduced that a wave of `relax` messages cannot "overtake" an earlier initiated wave of `propagate` messages. This follows from message delays or the used protocols. For example in DDAC4 the `relax` messages are broadcast, while the `propagate` messages are handed over from agent to agent along the lines of the constraint graph. If the wave of `relax` messages takes a "shortcut", it may overtake the wave of `propagate` messages. Thus a variable domain may be relaxed before it was actually pruned. The pruning will then be performed afterwords yielding incorrect results as the constraint is actually obsolete.

I solve this problem by storing the set of all deleted constraints in the set `knownAsDeleted` in each agent. Whenever an agent discovers that a constraint was deleted, it stores this information. After that, each agent will not prune any more values due to this constraint (Alg.2, line 1). Furthermore, the propagation will be stopped as no triples will be added to the `toSendList` in this algorithm. The use of `knownAsDeleted` is not ideal programming style as the set will constantly be extended and never reduced in continuous program executions. Thus, the complexity of Alg.2 (which checks every constraint against this set) is in the order of a potentially infinite number of deleted constraints. However, in the implementation of a concrete application this problem may be solved by deleting references from this set as soon as it can be assumed (with a sufficient likelihood) that no further propagation regarding the deleted constraint is to be performed.

5 Evaluation

5.1 Correctness

First I show that DDAC4 propagation is correct and achieves arc-consistency. Arc-consistency can be expected to be achieved as soon as the entire network has reached quiescence, i.e. the algorithm has terminated. This is the case when the following conditions are satisfied:

– no workers are executing any procedures or behaviors
– no worker has stored messages to be processed
– no messages were sent and not received

Any global state that satisfies all three conditions is called globally stable. For such states it can be shown that adding constraints will retain arc-consistency.

Lemma 1. *If no constraint deletion occurs, then constraint addition with DDAC4 will result in an arc-consistent globally stable state if all other constraints were also added with DDAC4.*

Proof. Since the algorithm of the workers is based on the correct AC-4 algorithm [6] it is sufficient to show that: (i) for each added constraint all variable-value pairs that it prohibits are detected; (ii) this inconsistency is reported to all agents that can deduce further inconsistencies and (iii) the algorithm reaches quiescence.

(i) Since all values of *both* involved workers are checked against any new constraint (Alg. 1 lines 2–5) the two workers to which the constraint was added will detect all inconsistent values.

(ii) Each inconsistent variable-value pair (i, v) is sent to all neighboring (Alg.1 line 6 and Alg. 2 line 4) workers. These include all workers that host a variable which is adjacent to `self` in the constraints graph and thus have common constraints with the sender (Alg.1 line 2). No other worker can detect further inconsistencies from (i, v) since it will not host a constraint over i. Each worker will process the inconsistencies (Alg.2) and detect all possible new inconsistencies because it checks all values from its initial domain (Alg.1 line 3) against all its constraints (Alg.2 lines 1–3). Since the system will not reach quiescence before all workers have finished processing their local `list` and no more inconsistencies are being communicated, there will be no inconsistencies (i, v) in the CSP that were not checked against all constraints over i.

(iii) Since there is only a finite number of values in each domain and there is no constraint deletion, there can only be a finite number of necessary value deletions and therefore the loop in Alg. 2 will always terminate. There are no deadlocks in the algorithm since all messages, except of `domain`, are asynchronous. The `domain` messages will not lead to a deadlock since it is triggered uniquely by the asynchronous message `newConstraint`.□

Now I will show that the constraint deletion algorithm will always reach arc-consistency. This also requires that quiescence is reached in the network.

Lemma 2. *Given a globally stable arc-consistent state of a CSP, the constraint deletion with DDAC4 will result in an arc-consistent globally stable state iff no constraint addition occurs.*

Proof. The constraint deletion removes the constraint properly from the CSP as can be seen in the procedure **deleteConstraint(c)** (Alg.3), it resets all changes

that were performed by **addConstraint(c)** (Alg.2) before. Every constraint deletion will cause a broadcast of `relax` messages (Alg.3 line 1). In the method **relax(c)** `self` will consider for every removed value whether it can be put back in its `domain`. It puts only those values back that were directly or indirectly removed by **c** and no other constraint. Thus it keeps all values excluded that are not arc-consistent in the CSP without **c**. The constraint deletion protocol reaches quiescence, since in each worker only finitely many values can have been pruned such that `removedList` is finite and the loop in the procedure **relax** terminates.□

Now it can also be deduced that the concurrent execution of addition and deletion and thus DDAC4 is correct.

Theorem 1. *Any CSP which is processed exclusively by DDAC4 is arc-consistent in any globally stable state.*

Proof. Regarding Lemmas 1 and 2, it remains to be shown that the concurrent addition and deletion of constraints is correct. For this I show that: (i) the constraint addition is independent of the progress of execution of any constraint deletion; (ii) constraint deletion will not miss any values to be added to the domain and (iii) constraint deletion and addition of identical constraints are always synchronized.

(i) Constraint addition and propagation exclusively use the initial variable domains for their inference, and are thus independent of the sets `domain` and `removedValues` which may or may not have been manipulated by the concurrently running constraint deletion algorithm.

(ii) According to Lemma 2, the constraint deletion algorithm will always put all values that have lost all justifications to be excluded back in the domain . But for every pruned value it knows all justifications from the set `removedValues`.

(iii) Due to the use of `knownAsDeleted` it will never be the case in any worker that propagation is performed after relaxation for the same constraint. It is not possible to add the same constraint twice (Alg.1 line 1). □

5.2 Complexity

For the following complexity analysis I leave out the required effort of the termination detection, as this is not part of the DDAC4 algorithm. As I mentioned before the complexity depends on the size of the set `knownAsDeleted` which should (and can) be kept low in typical applications. In the following I assume it contains k elements. Furthermore I refer to the number of valid constraints as c, to the number of variables as n and to the size of the largest domain of any variable as d. I specify the time, space and message complexity of DDAC4 algorithms. For the specification of the time complexity of distributed algorithms one must assume the worst case in which no computations are performed in parallel. Thus I specify the time complexity of the procedures **addConstraint** and **deleteConstraint** in the overall system. The required space of each worker

is considered separately. In distributed settings, the global space complexity is usually not of interest. The number of required messages is generally considered one of the most important features of distributed algorithms.

Theorem 2. *The time complexity of the procedure* **addConstraint** *is* $O(n^2 * d^2 * k)$.

Proof. Lines 1-2 Algorithm 1 in use $O(k) + O(c)$ steps, lines 3-5 take $O(d^2)$ and line 6 takes $O(c)$. This is performed in two workers yielding $2(O(k) + O(c) + O(d^2) + O(c)) = O(k) + O(c) + O(d^2)$. Algorithm 2 is executed in the worst case for every deleted value (there are $n * d$ values) by every worker and thus $n^2 * d$ times. Algorithm 2 itself has the following complexity: "foreach $v \in$ `support`" takes at most $O(d)$ steps, line 1 takes $O(k)$ steps, and line 4 $O(c)$. Overall Alg.2 is thus $O(d) * O(k) + O(c)$. Putting things together and knowing that $O(c) \subseteq O(n^2)$ holds in binary CSP the complexity of `addConstraint` is $O(k) + O(c) + O(d^2) + n^2 * d * (O(d) * O(k) + O(c)) = O(n^2 * d^2 * k)$.

Theorem 3. *The time complexity of the procedure* **deleteConstraint** *is* $O(n * k) + O(n^2 * d * c)$.

Proof. The procedure **deleteConstraint** takes $O(c) + O(d^2) + O(n)$ steps for checking if the constraint exists, updating `support` and `counter` and broadcasting a message to all workers. This is done twice. The procedure **relax** is executed once in every worker. It takes $O(k)$ steps to check whether the constraint is known as deleted. Then it traverses `removedList`, which may contain for every value of every variable and every constraint an element. It's traversal may thus cost up to $O(n * d * c)$ steps. Putting things together the complexity of **deleteConstraint** is $2(O(c) + O(d^2) + O(n)) + n * (O(k) + O(n * d * c)) = O(n * k) + O(n^2 * d * c)$

Theorem 4. *The space complexity of each worker is* $O(k) + O(n * d * c)$.

Proof. I define the function *size* which assigns to each data structure $x \in X$ in the DDAC4 workers its worst case size: $size = \{$(`domain`, $O(d)$), (`initDomain`, $O(d)$), (`list`, $O(n * d)$), (`toSendList`, $O(n * d)$), (`knowAsDeleted`, $O(k)$), (`constraints`, $O(c)$), (`neighbors`, $O(n)$), (`support`, $O(d * n)$), (`counter`, $O(d * n)$), (`removedList`, $O(n * d * c)$)$\}$. The space complexity of each worker is thus $\sum_{x \in X} size(x) = O(k) + O(n * d * c)$

Theorem 5. *The DDAC4 algorithm requires* $O(n^3 * d)$ *messages for constraint propagation.*

Proof. The procedure **sendList** sends $O(n)$ messages each time it is invoked. For every new constraint it is invoked twice by the procedure **addConstraint** and once in each execution of Algorithm 2. Alg. 2 is executed at most $n^2 * d$ times as I have shown in the proof of Theorem 2. Thus propagation requires overall $O(n) * (2 + n^2 * d) = O(n^3) * d$ messages.

Theorem 6. *The DDAC4 algorithm requires* $O(n)$ *messages for constraint relaxation. This complexity is optimal.*

Proof. As can be seen in Algorithm 3, the procedure **deleteConstraint** broadcasts a message to all n workers. This is executed twice and there is one `delConstraint` message. Thus constraint deletion requires $1 + 2 * O(n) = O(n)$ messages. This is optimal, since every constraint may yield consequences in every variable and thus every worker. Consequently each worker must be notified for constraint relaxation.

5.3 Empirical Evaluation

I have implemented the DisAC-4 and DDAC4 algorithms in a multi-threaded Java program. Each agent contsitutes one concurrent thread and the agents communicate by dropping messages to other agents' message-channels. The common memory is not used except the references to the channels. For evaluation I measured the runtime, which is the overall CPU time when the concurrent program is executed on one processor, the total number of messages and the number of(concurrent) constraint checks. I ran four tests to evaluate the performance of DDAC4 with Java 1.4.2 on a Linux desktop computer one 1.8GHz Pentium processor and 512MB memory:

rbc-x Random binary CSPs with 30 variables, domains size 15 (for each variable), density = 0.5 and varying tightness **x** between 0.2 and 0.8. The sample size was 30. For these problems I measured the average effort to enforce arc-consistency by DisAC-4 and DDAC4. In the latter, all constraints were iteratively added.

rq30-x The 30-queens problem where **x** queens are placed on (consistent) positions. I compared the performance of DisAC-4 and DDAC4 to achieve arc-consistency.

inc30q-x This test evaluates the incrementality of constraint addition. In the 30-queens problem, I check the effort of DDAC4 to put the **x**-th queen (consistently) in the **x**-th coloumn, while all queens in more left hand colomns remain set. Setting a queen is performed by adding a new binary constraint over **x** and a new variable such that only one value is allowed for **x**.

dyn30q-x This evaluates the efficiency of constraint deletion. One queen is set on an empty chessboard in colomn **x** and removed again. This is done in two ways: first, the system waits for global quiescence after adding the constraint and second the constraint is deleted immediately after it was sent such that the propagation will most likely not be perfomed completely.

The results of **rbc-x** and **rq30-x** are shown in Figure 1. These diagrams compare the effort to enforce arc-consistency. In all our experiments both the number of sent messages and the runtime of DDAC4 is better than that of DisAC-4. The number of (concurrent) constraint checks was also monitored and, as expected, it was equal for both algorithms in all tests. The results of the **inc30q-x** test are shown in the left diagram in Figure 2. It can be seen that due to the incremental constraint addition, the propagation effort decreases as the search space is pruned. The cost to add new constraints gets as low as the cost to check

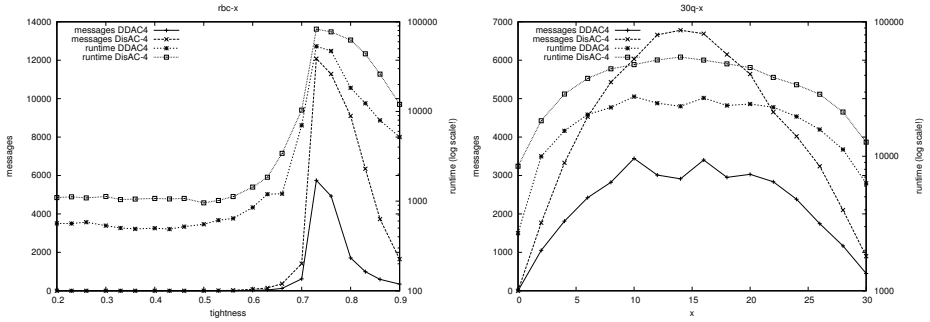


Fig. 1. Effort to enforce arc-consistency.

its consistency once. The results of **dyn30q-x** are shown in the right diagram of Figure 2. It can be seen that the effort to delete is always much smaller with a constant factor than the effort to add a constraint. In the comparison of the test perfomed with and without waiting for quiescence after the constraint addition it can be seen that the runtime without waiting is smaller than the sum of both tasks performed separately. Partially, this speedup is gained by leaving out the effort for termination detection in between both steps. However, there can also be less propagation performed if the constraint is deleted immediately after adding it. This results from the fact that the `relax`-messages can reach agents faster than their repsective `propagate` messages as described in Section 4.1 and thus make the propagation obsolete. This could also be seen by the number of constraint checks in this experiement. In the n-queens example, this gain of efficiency can only be reflected in runtime or constraint checks, as the number of messages will always be the same since the constraint graph is fully connected.

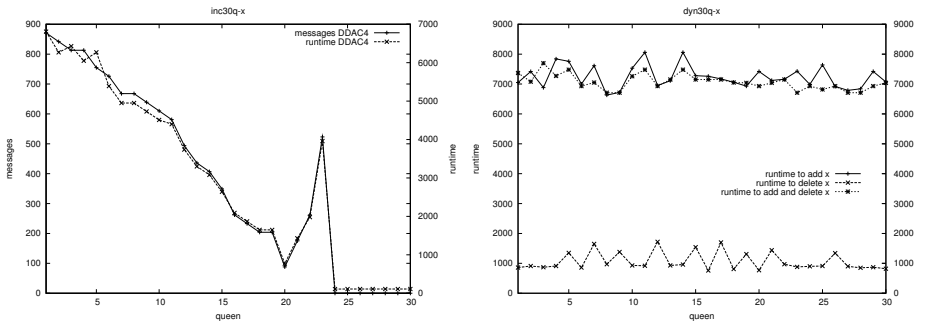


Fig. 2. Effort of incremental constraint addition (left) and effort to add and delete a constraint (right).

6 Future Work

The AC-4 algorithm is not the most efficient algorithm for arc-consistency enforcement, neither in a centralized [10], a distributed [5] nor dynamic [3] setting.

Thus I plan to use the experience gained in the development of the DDAC4 algorithm to find more efficient AC algorithms for distributed and dynamic problems. The distributed DisAC-9 [5] algorithm which is proven to be optimal with respect to the number of required messages will be the starting point for my future research. I expect that the DisAC-9 and DnAC-6 [3] algorithms can be integrated in a similar way to the integration of their AC-4 counterparts I presented in this paper.

With a dynamic AC algorithm constraint programmers are not only able to add (and propagate) and delete (and de-propagate) binary, but also unary constraints such as variable instantiations. With this investigate look-ahead algorithms for distributed problems which are based on IDIBT [8]. The implementation of look-ahead in asynchronous systems is highly complex as no central structures exist to trail the history of the search [9]. The distributed search cannot restore a former global state upon backtracking, but may still have to relax potentially every variable domain in the CSP. My approach will be to use distributed and incremental constraint addition and deletion. If the addition and deletion of instantiations can be (de-)propagated in an efficient way, a distributed look-ahead algorithm can be expected to significantly improve the efficiency of today's asynchronous search algorithms. This expectation follows from the efficiency improvement in monolithic systems, from other distributed search with propagation [9] and prelimenary experiements [8].

References

1. Christian Bessiére. Arc-consistency in dynamic constraint satisfaction problems. In *Proc. AAAI'91*, pages 221–226, 1991.
2. George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems, 3rd edition*. Addison Wesley, 2001.
3. Romuald Debruyne. Arc-consistency in dynamic csps is no more prohibitive. In *8th conference on Tools in Artificial Intelligence (TAI96)*, pages 299–306, 1996.
4. Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
5. Youssef Hamadi. Optimal distributed arc-consistency. In *Principles and Practice of Constraint Programming*, pages 219–233, 1999.
6. R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
7. T. Nguyen and Yves Deville. A distributed arc-consistency algorithm. *Science of Computer Programming*, 30(1-2):227–250, 1998.
8. Georg Ringwelski. Incremental constraint propagation for interleaved distributed backtracking. In *Proc. 5th workshop on Distributed Constraint Reasoning, Toronto*, 2004.
9. M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Consistency maintainance for abt. In Toby Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001*, pages 271–285. Springer LNCS 2239, 2001.
10. Richard J. Wallace. Why ac-3 is almost always better than ac-4. In *Proc. IJCAI-93*, pages 239–245, 1993.

Using additional information in DisCSPs search *

Amnon Meisels and Oz Lavee
{am,laveeo}@cs.bgu.ac.il

Department of Computer Science,
Ben-Gurion University of the Negev
Beer-Sheva, 84-105, Israel

Abstract. A method of volunteering information during asynchronous search on DisCSPs is presented. The meeting scheduling problem (MSP) is formulated as a distributed search problem. In order to implement asynchronous backtracking (ABT) for the MSP, a multi-variable version of ABT is described. Agents participate in multiple meetings, where each meeting is represented by a variable that needs to be assigned a time-slot. Assignments are constrained by arrival-time constraints, since meetings take place in different locations. All constraints are local to their agents.
Additional information is in the form of Nogoods. During search for a consistent schedule for all meetings, agents can generate and send additional Nogoods to those sent by the ABT algorithm. When additional Nogoods are sent, the efficiency of asynchronous backtracking is enhanced. This effect grows with the number of additional volunteered Nogoods.

1 Introduction

An important goal of search algorithms for the distributed constraint satisfaction problem is to support agents' privacy. During cooperative search for a globaly consistent solution, agents exchange messages about their assignments and about conflicts with other agents' assignments. This creates a natural trade-off between information disclosure and the efficiency (and correctness) of the distributed search process. The first to investigate measures of privacy for DisCSPs were Meseguer et. al. [Brito and Meseguer2003]. In a series of two papers they presented algorithms for maintaining two types of privacy during the run of the asynchronous backtracking (ABT) algorithm
[Meseguer and Jimenez2000,Brito and Meseguer2003].

A different approach for investigating the privacy of distributed search was presented first by [Wallace and Freuder2002]. This concrete family of problems was used to compare the amount of needed computations for finding a solution, when different quantities of information were exchanged among the searching agents [Wallace2003].

The present paper uses the family of meetings scheduling problems (MSPs) to achieve three goals. First, to define a general family of *Meetings scheduling search problems* that will serve as framework for the study of privacy. For the family of MSPs the agents need to solve a hard search problem. The second goal is to enhance the

* Partially supported by the Lynn and William Frankel Center for Computer Science

asynchronous backtracking algorithm [Yokoo and Hirayama2000], for multiple variables per agent. This is needed in order for each agent to schedule its multiple meetings and then cooperate with other agents to search for compatible schedules for all meetings. The third and main goal of the present study is to define a consistent method for enhancing the information content of messages of the search algorithm (ABT) that will enable a more efficient computation.

This paper examines the effect of volunteering additional information, in the form of additional Nogood messages, on the efficiency of search. The trade-off between information and efficiency is investigated in the context of the meetings scheduling problem [Wallace and Freuder2002].

The meeting scheduling problem *(MSP)*, is the problem of coordinating a meeting among several agents each one with its own calendar and has appeared first in [Garrido and Sycara1995,Sen and Durfee1995]. A very restricted form of the MSP was investigated with respect to privacy by [Wallace and Freuder2002]. The tradeoff between the privacy of agents' meetings and the efficiency of the search process is studied by the use of a simple instance of the meeting scheduling problem. The instances used in [Wallace and Freuder2002] have only one meeting to coordinate, which all agents have to attend. Agents must be able to get from their private meetings to the scheduled meeting according to the traveling time constraints. Each agent has its own private calendar that defines its constraints regarding the time and location of the meetings.

The MSP has two main characteristics that make it into a DisCSP. It is logically distributed among all agents and since calendars are privately owned, it must use a distributed search process in order to find a solution that is consistent with all agents. The meetings of every agent are constrained with each other and the solution is globaly consistent if every agent is able to reach all meetings in which it participates.

The aspect of privacy is very natural to the MSP. Agents do not want to reveal information regarding their calendar. In the studies of [Wallace and Freuder2002,Wallace2003], privacy is in fact measured by the fraction of calendars of agents that becomes known to other agents during the search process.

Another simplified form of the MSP was used by Bessiere et. al. [Bessiere *et al.*2001] for testing the Asynchronous Backtracking algorithm. The problem used in [Bessiere *et al.*2001] has three groups. Each group has to schedule a meeting for all its members, with the constraint that two groups cannot meet at the same time and location. Perceived as a centralized constraints satisfaction problem (CSP), each meeting can be represented by a variable and the values to be assigned are the weekly time-slots. From this point of view, the MSP of [Wallace and Freuder2002] has one variable and the MSP of [Bessiere *et al.*2001] has three variables. The constraints of the [Wallace and Freuder2002] MSP are unary, consisting of all forbidden times and locations. The search space of CSPs is exponential in the number of variables [Dechter2003] and in this respect both of the above problems are not hard search problems.

In [Wallace2003] the family of MSPs as been extended to be the graph coloring problem for n meetings (variables). The problem is to assign time-slots to all n variables (meetings), such that each variable is owned by more than one agent. The constraints among the values assigned to meetings which include a specific agent are inequality constraints. This creates a graph coloring problem of a distributed nature. Each agent

owns the variables corresponding to meetings in which it participates and an inequality constraint holds among them [Wallace2003]. The family of MSPs of the present study are general DisCSPs and its arrival-time constraints are more general than inequalities.

Former studies of privacy eficiency trade-off in distributed search used either a simple iterative algorithm [Wallace and Freuder2002], or a synchronous distributed backtracking for solving the problem [Wallace2003]. The present study, investigates the privacy efficiency trade-off for a general MSP and for the enhanced asynchronous backtracking (ABT) algorithm. It measures the effect of asynchronous exchange of additional information on asynchronous search (see section 4).

In section 2 the Meeting Scheduling Problem (MSP) is defined, as well as its *CSP* representation and its distributed CSP form. Section 3 presents a version of the ABT algorithm [Bessiere *et al.*2001] for multi variable agents. The issue of additional information for search enhancement, in the context of the MSP, is at the center of section 4. It analyses ways of sending additional information during search and presents a form that sends additional Nogoods to standard ABT. An extensive experimental investigation of the behavior of the proposed method of voluntary information, with respect to search efficiency, is described in section 5.

2 The Meeting Scheduling Problem

The definition of the meeting scheduling problem is presented in three stages. First, the logical meeting scheduling problem. Second, its representation as a (centralized) CSP and third, the representation as a distributed CSP. The meeting scheduling problem *(MSP)* has been defined in many versions with different parameters, from duration of meetings [Wallace and Freuder2002] to preferences of agents [Sen and Durfee1995]. The family of MSPs that is at the focus of the present study is defined as follows:

- A group S of m agents
- A set T of n meetings
- Each meeting is associated with a set $s_i \subset S$ of agents that attend it
- Consequently, each agent has a set of meetings that it must attend
- Each meeting is associated with a location
- The scheduled time-slots for meetings in T must enable the participating agents to travel among their meetings

An example of a conflict of an agent's constraint is a meeting A, scheduled to 14:00 in Rome and a meeting B, that includes the same agent, that is scheduled for 16:00 in Paris. Each meeting is one hour long and the traveling time between Rome and Paris is two hours. It is assumed that there are no private meetings for any agent. This generates no loss of generality, since private meetings (or agents' private calendars, as in [Wallace and Freuder2002] for example) can be simply represented by unary constraints, removing values from domains of meetings. The duration of each meeting is one hour and the traveling time between any two locations is equal for all the agents (no agent is faster than another). The agents need to negotiate in order to search for a schedule of all meetings that meets all of the participants arrival-time constraints.

The meeting scheduling problem as described above can be represented as a constraints satisfaction problem in the following way:

– a set of variables Z - $m_1, m_2..., m_n$ the meetings to be scheduled
– domains of values D - all weekly time-slots
– a set of constraints C - for every pair of meetings m_i, m_j there is an arrival-time constraint, if there is an agent that participates in both meetings

As already mentioned, private meetings are equivalent to unary constraints removing values from domains of some meetings. Since all agents have the same arrival-times between any two locations, there is only one type of arrival-time constraint.

arrival-time constraint - Given two time-slots t_i, t_j there is a conflict if
$|time(t_i) - time(t_j)| - duration <= TravellingTime(location(m_i), location(m_j))$

The tightness of the arrival-time constraint can be measured for a given definition of distances between locations. Expressing distances in terms of time-slots, enlarging the arrival-times has the effect of tightening the constraints of the problem.

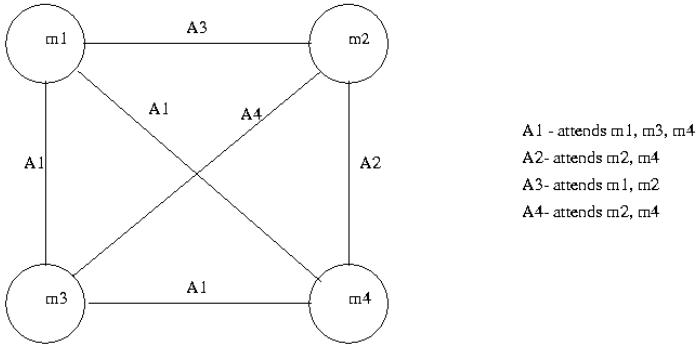


Fig. 1. the Meeting Scheduling Problem as a centralized CSP

Figure 1 presents the representation of a meeting scheduling problem as a CSP. The nodes are the meetings (the variables) and each edge represents a binary arrival-time constraint. Each edge is labeled by the agent, attending both meetings, that generates the arrival-time constraint.

Representing the MSP as a distributed CSP needs to associate variables with the different agents. Our distributed CSP representation can be described as follows:

– Agents - the Group S of agents
– For each Agent $s_i \in S$ there is a variable x_j^i, for Every meeting m_j that s_i attends.
– Each agent s_i includes arrival-time constraint between every pair of its local variables x_j^i, x_k^i.
– for each two agents s_i, s_j that attend meeting m_k there is an equality inter-constraint between the variables x_k^i, x_k^j, corresponding to the meeting m_k.

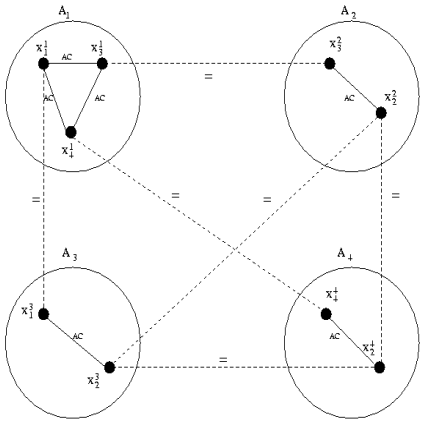


Fig. 2. the Meeting Scheduling Problem as a DisCSP

The representation of the *MSP* of figure 1 as a *DisCSP* can be seen in figure 2, where agents include multiple local variables connected by arrival-time constraints. Edges between variables of different agents represent the equality inter-constraint.

2.1 Random Meeting Scheduling problems (RMSPs)

Random Meeting Scheduling Problems *(RMSPs)* can be parametrized in numerous ways. Parameters can be the number of meetings, meetings' locations, number of agents, etc. To simplify the experimental design, one can use the relevant features of the CSP representation. Let us first denote a set of parameters:

- number of meetings - m
- number of agents - n
- number of meetings per agent- k
- distances between locations of meetings
- domain size - number of time-slots

The meetings are the set of m variables of the constraints network, each representing a meeting at a specific location. The domains of values are the time-slots. An edge between any pair of variables represents an agent that participates in both meetings. The density of the constraints network depends on the number of agents and the distribution of meetings that each agent attends. If each agent participates in k meetings, one can generate the resulting CSP as follows. For each of the n agents a clique of k variables is selected randomly, such that not all of the edges of the clique are already in the network. Each clique is added to the CSP, representing the arrival-time constraint between the meetings of each agent.

Similarly to randomly generated CSPs, one can calculate the resulting density p_1 of the network and the tightness p_2. p_1 is the ratio of the total number of edges to the

maximal number - $m \times (m-1)/2$. The tightness of the generated CSPs, p_2, can be calculated by using the average distance between locations. For two meetings m_i, m_j connected by an arrival-time constraint, each value in the domain of m_i is inconsistent with $2 \times distance(m_i, m_j)$ values (time-slots) in the domain of m_j.

The representation of the above CSP as a distributed CSP is streightforward. Each meeting variable m_j in the CSP corresponds to the variable x_j^i within each agent A_i that participates in m_j. These variables are connected by the equality constraint on x_j^i, meaning that for all agents A_i, the meeting m_j is at the same time. When ordering the DisCSP, the first of the variables that represent the same meeting is connected to all other variables of the same meeting by an equality constraint. Each meeting has one participating agent that is first in the global order. This agent proposes time-slots for the meeting, during the run of asynchronous search. No other pair of participating agents need to be connected by an equality constraint.

3 Multi-variable Asynchronous Backtracking

Every agent of the meeting scheduling problem includes multiple variables, one for each meeting it attends. As a result, the distributed search algorithm must be able to deal with multiple local variables. For asynchronous backtracking (ABT) this is a special version of the algorithm that has not been described in the fundamental publications [Yokoo and Hirayama2000,Bessiere *et al.*2001]. The multi-variables version of ABT that is presented below is an adaptation from [Bessiere *et al.*2001], but, uses conflict- based backjumping (CBJ) [Prosser1993] for the local CSP of each agent.

As in standard ABT, all agents are assumed to be ordered [Bessiere *et al.*2001]. All variables of each agent are ordered successively, so that the variables of agent A_{i+1} follow successively the variables of agent A_i. The pseudo-code of the $ABT-CBJ$ algorithm for multi-variable agents is presented in Figure 3. Agents running the algorithm wait for messages and upon receiving a message call the suitable procedure for the type of the received message. Elimination explanations are kept for each value of every variable (cf. [Ginsberg1993,Bessiere *et al.*2001]. Explanations may contain either local variables with higher priority or variables of other agents, with higher priority.

The ***processInfo*** procedure is called when an **ok?** message is received. It updates the $AgentView$ with the received assignment and removes all eliminating explanations in all the local variables that contain the obsolete assignment of the received variable.

When a backtrack message is received, the ***resolveConflict*** procedure is called. This procedure is similar to the ***resolveConflict*** of ABT in [Bessiere *et al.*2001]. It checks the consistency of the received Nogood with the $AgentView$. If it is consistent, then ***resolveConflict*** updates the relevant assignments in the $AgentView$ (the non Γ^- variables in the $AgentView$, in terms of [Bessiere *et al.*2001]). It also removes the eliminated value from the relevant local variable.

The ***chooseValues()*** procedure assigns values to all the local variables, checking that all the eliminators in all the variables are consistent with the $AgentView$. Lines 1,2 deal with the case that the current assignment of all the local variables is consistent with the $AgentView$ and no changes needed. If this is not the case, then lines 4-18 find a consistent assignment for all the local variables. The order of all variables is

– **ABT-CBJ**:
1.$SelfVars \leftarrow empty, end \leftarrow false$
2.chooseValues()
3.**while** ($\neg end$)
4. $msgs \leftarrow$ recvieve_all()
5. **foreach** $msg \in msgs$ do
6. **switch** ($msg.type$)
7. info : processInfo(msg)
8. Back : resolveConflict(msg)
9. Stop : $end \leftarrow true$

– **processInfo(**msg**)**:
1. update($AgentView, msg.variable, msg.value$)
2. remove eliminators inconsistent with $AgentView$
3. chooseValues()

– **chooseValues**:
1.**if** consistent($SelfVars, AgentView$)
2. **then return**
3.**else**
4. **for** $p = 0$ **to** $SelfVars.size$
5. $found \leftarrow$ find_assignment($selfVars[p]$)
6. **if** found = $false$
7. $Nogood \leftarrow$ resolve($SelfVars[p].NogoodStore$)
8. **if** rhs($Nogood$)$\in SelfVars$
9. $q \leftarrow$ index of rhs($Nogood$)
10. **for** $i = q + 1$ to p
11. remove from $selfVars$ Nogoods containing $SelfVars[i]$
12. removeValue($SelfVars[q], Nogood$)
13. $p = q$
14. **else**
15. remove rhs($Nogood$) from AgentView
16. backtrack($Nogood$)
17. remove all Nogoods containing $variables \in SelfVars$
18. p=0
19. **for each** $agent \in \Gamma^{+}(updatedVariable)$ sendMsg:Info($agent, updatedVariable$)

– **removeValue(**$variable, Nogood$**)**:
1. set eliminator $Nogood$ at $variable$
2. **for each** $var \in SelfVars$
3. remove eliminators contianing $variable$

– **resolveConflict(**msg**)**:
1.**if** consistent($msg.Nogood, \Gamma^{-} \cup \{SelfVars\}$)
2. **for each** $assign \in lhs(msg.Nogood) \setminus \Gamma^{-}$ **do**
3. update($AgentView, ngVar$)
4. remove eleiminators inconsistent with $AgentView$
5. $rhsVariable \leftarrow rhs(Nogood)$
6. removeValue($rhsVariable, Nogood$)
7. chooseValues()
8.**else if** $msg.sender \in \Gamma^{+} \wedge$ Consistent($msg.Nogood, SelfVars[rhs(msg.Nogood)]$)
then sendMsg:Info($msg.sender, SelfVars[rhs(msg.Nogood)]$)

Fig. 3. The multi-variable ABT algorithm

static. In line 5 a search for a value for the current variable x_j^i is performed, such that it is consistent with all assigned local variables and with the $AgentView$. If a value is not consistent an eliminator is added for this value. If no consistent value is found, the eliminators of the current variable are resolved to form a Nogood, in line 7.

When the Nogood points to a local variable x_k^i then a backjump to x_k^i will be performed, with the Nogood as eliminator for the assignment of x_k^i (lines 9-13). The backjump requires the removal of all eliminators from all the local variables $x_{k+1..j}^i$, that were jumped over (lines 10-11). This procedure implements the backjumping algorithm for multi local variables. The backjumping algorithm that is implemented for local variables is similar to [Ginsberg1993]. If the right hand side of the Nogood is a variable of a different agent, then it contains no local variables (since all the local variables are ordered successively). Therefore, the Nogood is sent in a backtrack message and the right hand side assignment of the Nogood is removed from the $AgentView$ (lines 15-16). Next, the local process for consistent assignments to all local variables starts from the beginning (line 17-18). When consistent assignments for all local variables have been found, all new assignments are sent by an **ok?** message to all the agents that are later in the order of the problem (agents in Γ^+ of the updated variable, similarly to standard ABT [Bessiere *et al.*2001]).

4 Volunteering additional information

In studies of privacy issues of DisCSP search, the option of hiding information about assignments was considered by Meseguer and Jimenez [Meseguer and Jimenez2000]. In order to keep privacy of assignments, Meseguer et. al. propose to send forward a list of allowed values instead of the assignment itself. This idea cannot work for the MSP, because the equality constraint identifies the assignment of a single time- slot with the list of legal assignments for the agent receiving the **ok?** message. By the same token, constraints of the MSP cannot be kept private by the method of [Brito and Meseguer2003]. All the constraints of the meeting scheduling problem are arrival-time constraints which are *internal* to each agent. The inter- agent constraints are just equality constraints, for which hiding is meaningless.

Recently, several studies have investigated the trade-off between privacy and the efficiency of search. Wallace and Freuder [Wallace and Freuder2002] have looked at a simple MSP to show that loss of privacy enhances search efficiency. In a later study, Wallace have shown a similar trade-off to hold for a distributed graph coloring problem [Wallace2003]. The version of the MSP that was presented in section 2, is more general than the graph coloring problem of [Wallace2003]. However, by the above analysis of the privacy of MSPs, both types of privacy are not simply connected to the run of the search algorithm. For the family of meeting scheduling problems, the privacy question can be replaced by the possibility of volunteering additional information, to help agents arrive faster at a consistent solution.

Backtracking messages of the ABT algorithm contain Nogoods, which represent unsoluble sub-search spaces. Backtracking messages are based on violation of constraints. Since Nogoods contain the only information about constraints, one can *add Nogoods in order to volunteer relevant information*. Adding Nogoods to the asynchronous back-

tracking algorithm is presented below. It forms a method of adding viable information to agents, to enhance their efficiency in arriving at a solution.

Let us start with an example in which agent A_i has received an **ok?** message from agent A_l, proposing an assignment for its variable $< x_r^i = 15:00 >$. Agent A_i has another meeting, with an assignment $< x_s^i = 14:00 >$, proposed by agent A_j. As a result of arrival-time conflict, A_i has to reject the new assignment by sending the Nogood $\{(x_s^j = 14:00 \rightarrow x_r^l \neq 15:00)\}$. This Nogood informs agent A_l that meeting m_r, for which it is responsible, is in conflict with meeting m_s. It can also deduce that the agent responsible for meeting m_s is A_j. If the arrival time constraint between meetings m_r and m_s is three hours, than the following Nogood holds - $\{(x_s^j = 14:00 \rightarrow x_r^l \neq 16:00)\}$. In other words, agent A_i can generate additional Nogoods when the conflict occurs between its local variables. It is important to note here that Nogoods retain their meaning. In other words, Nogoods are valid during all stages of search. In that sense, the additional information retains its validity through all of the search process.

Different versions of asynchronous backtracking retain Nogoods in different ways. From retaining all of them in the first versions of ABT [Yokoo *et al.*1998], to erasing all Nogoods that are not currently consistent [Bessiere *et al.*2001]. It is important to note that the variety of strategies for retaining received Nogoods during asynchronous backtracking relates to space efficiency and not to completeness. This is why different correct algorithms choose differently [Bessiere *et al.*2001]. The present method does not interfere with the correctness of the multiple-variable ABT and it is independent of the question whether additional (volunteered) Nogoods are retained or not.

An absolute measure of the information content of a Nogood is the size of the eliminated subtree from the search tree. This measure depends on the size of the Nogood, the shorter the Nogood, the larger the eliminated subtree. It is easy to compute the fraction of the search space that is eliminated. If the LHS of the Nogood is $< X_k^1, T_k^1 > \ldots < X_m^i, T_m^i >$ and there are n agents, then the eliminated subtree is of size $D^{i+1} \times \ldots \times D^n$. The fraction of the search space that is eliminated by the Nogood is simply $\frac{D^{i+1} \times \ldots \times D^n}{D^1 \times \ldots \times D^n}$

For the meeting scheduling problem *(MSP)* a Nogood is a partial schedule, that conflicts with a proposed assignment of a time-slot to a given meeting. If a Nogood is generated by a conflict within the sending agent, it reflects a conflict of two or more of the meetings of the sending agent. Alternatively, the Nogood sent has been received (in longer form) by the sending agent and could not be resolved by it (see function **resolveConflicts()** in Figure 3).

The present investigation uses locally generated *additional* Nogoods as a form of volunteering information. The proposed method is to generate additional Nogoods and add them to every backtrack message. In order to add Nogoods to backtrack messages, the refined backtrack procedure is presented in Figure 4. The improved procedure checks for addtional time slots of x_k^i, that create an immidate conflict between the local variable x_k^i of the meeting m_k and local variable x_w^i of meeting m_w. In the code of Figure 4, x_k^i has an equality constraint with the right hand side of the Nogood that forbids $< x_k^j, value >$ and x_w^i has an equality constraint with one of the variables on the left hand side of the Nogood that is being sent back in the backtrack message

(lines 3-5). When the enhanced backtrack procedure finds such a conflict, it adds it to the *additionalNogoods* list (line 6). When the *additionalNogoods* list reaches the size of the predefined parameter - *informationFactor*, it is sent with the original nogood in a backtrack message (lines 7-8).

– **backtrack(**$nogood$**)**:
1.$additionalNogoods \leftarrow \emptyset$
2.$< x_k^j, value > \leftarrow rhs(nogood)$
3.**for each** $< x_w^l, val' > \in lhs(nogood)$ **do**
4. **for each** $val \in domain(x_k^i$ **do**
5. **if not** consistent($< x_w^i, val' >, x_k^i, val$) **then**
6. $additionalNogoods$.add($< x_w^l, val' > \rightarrow < x_k^j, val >$)
7. **if** $additionalNogoods.size \geq informationFactor$ **then**
8. send:BT($Nogood, additionalNogoods$)
9. **return**
10.send:BT($Nogood, additionalNogoods$)

end procedure

Fig. 4. Backtrack procedure - sending additional Nogoods for agent A^i

5 Experimetal Results

To simulate asynchronous agents, a Distributed CSP simulator is used, that implements agents as *Java Threads*. Threads (agents) run asynchronously, exchanging messages by using a common mailer. After the algorithm is initiated, agents block on incoming message queues and become active when messages are received.

Search algorithms on DisCSPs are run concurrently by all agents and their performance must be measured in terms of distributed computation. Two measures are commonly used to evaluate distributed algorithms - time, which is measured in terms of computational effort and network load [Lynch1997]. The time performance of search algorithms on DisCSPs has traditionally been measured by the number of computation cycles or steps (cf. [Yokoo and Hirayama2000]). In order to take into account the effort an agent makes during its local assignment the computational effort can be measured by the number of concurrent constraints checks that agents perform ([Meisels *et al.*2002]). Measuring the network load poses a much simpler problem. Network load is generally measured by counting the total number of messages sent during search [Lynch1997].

In the asynchronous simulator, concurrent steps of computation are counted by a method similar to that of [Lamport1978,Meisels *et al.*2002]. Every agent holds a counter of computation steps. Every message carries the value of the sending agent's counter. When an agent receives a message it updates its counter to the largest value between its own counter and the counter value carried by the message. By reporting the cost of the search as the largest counter held by some agent at the end of the search,

we achieve a measure of concurrent search effort that is similar to Lamport's logical time [Lamport1978].

Meeting scheduling problems where generated randomly, as described in section 2.1. Locations of meetings were selected randomly from a set of 4 $\{P_1, P_2, P_3, P_4\}$. The distances among the 4 locations, in terms of time of travel, generate the arrival-time constraints among meetings. Two sets of experiments were performed. The first set of experiments has 9 meetings and 16 agents. Each agent participates in 3 meetings. The meetings of each agent were selected randomly, as described in section 2.1, by selecting cliques of 3 meetings. The random selection was performed so that no two agents attend exactly the same 3 meetings. The second set of experiments used 9 meetings and 24 agents with 2 meetings per agent. Each agent in this experiment adds an edge and not a clique, during the generation of random problems.

Two sets of distances among the meetings' locations are used in the first set of experiments. The distances are described in Figures 5. The domains of all meetings contain 24 time-slots. Each experiment was performed 10 times and average results are reported.

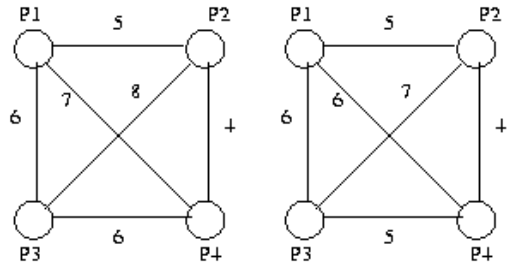


Fig. 5. (a) Distances between locations of meetings, (b) Smaller distances between locations of meetings.

All experiments consist of performing search for a consistent solution of the randomly generated problems, with different amounts of volunteered information. The runs of the problems had 6 different numbers of added Nogoods, or in terms of the parameter of algorithm **backtrack()** of Figure 4 *informationFactor* = $\{0,1,2,4,6,8,10\}$. The *informationFactor* is the maximum number of additional Nogoods per backtrack. The actual number in the experiments was very close and the results are parametrized by the average number of actual Nogoods sent.

Figures 6, 7 present the behavior of the three measures of performance, for growing number of additional Nogoods, in the first set of experiments (16 meetings and 9 agents). Two different distance graphs are presented in these figures. The grey columns use the distance map of Figure 5(a) and the dark columns use a distance map of smaller distances (Figure 5(b)). The smaller distances rule out fewer values per arrival-time constraint, thus generating a CSP with lower tightness.

It is easy to see that the computational effort is decreasing with increasing number of additional Nogoods. The overall factor of improvement in communication load is larger than 3 for the experiment with larger distances. This is the harder problem to solve. For the easier problem, with smaller distances, the overall scale is much smaller (i.e. easier problem) and the improvement is less dramatic. The improvement in communication load is easy to understabd. Backtracking messages rule out a larger number of assignments for the receiving agent. As a result, less **info** messages are sent forward.

The improvement in the number of steps of computation can be explained by the following example. When an additional Nogood eliminating $x_k^i = 4$ is received by agent A_i, it eliminates a cycle of steps: assigning $x_k^i = 4$, assigning the rest of the local variables of A_i, sending **ok?** messages to all agents in Γ^+ and finally receiving a backtrack message that eliminates $x_k^i = 4$.

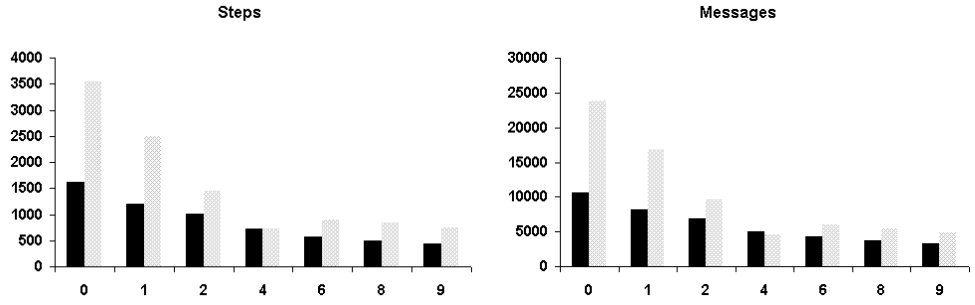


Fig. 6. (a) steps of computation vs. actual additional Nogoods, (b) total number of messages vs. additional Nogoods.

It is important to note that the production of additional Nogoods has a computational cost. For the MSP, the computational effort required for producing additional Nogoods is relatively small. This is a result of the structure of the problem, because all the intra-constraints are equality constraints. In other words, a Nogood ($x_k^i = 16:00 \rightarrow x_r^j \neq 17:00$) can be generated easily in agent A_l that attends both meetings m_k, m_r.

The computational effort of generating the additional Nogoods should affect the CCCs measure most, since the generation of Nogoods requires constraint checks. It is therefore interesting that the CCC performance measure in Figure 7 shows an improvement with increasing number of additional Nogoods. It is important to note that additional Nogoods are not always relevant for the receiving agent. The removed value of the additional Nogood may have already been erased, thus wasting the effort of generating the additional Nogood.

In the second set of experiments we used 9 meetings and 24 agents, with 2 meetings per agent. Each agent in this experiment adds an edge and not a clique, during the generation of random problems. This set of problems require less computational effort

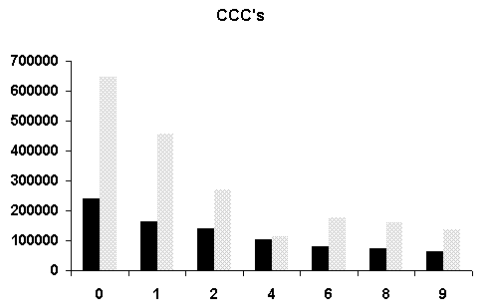


Fig. 7. Concurrent constraints checks (CCCs) vs. actual additional Nogoods

than the first two experiments, but the decrease of computational effort with increasing number of additional Nogoods is clear (Figures 8).

6 Discussion

The first investigation of the trade-off between privacy and efficiency of search was done by [Wallace and Freuder2002]. In their paper the agents tried to find a time-slot for a *single meeting of all agents*. The additional information in [Wallace and Freuder2002] was sets of time-slots that are already taken in individual calendars of agents. The addition of such information is immediately related to the privacy of agents. Sending lists of taken time-slots (i.e. with former meetings) reveals parts of the calendar of the sending agent.

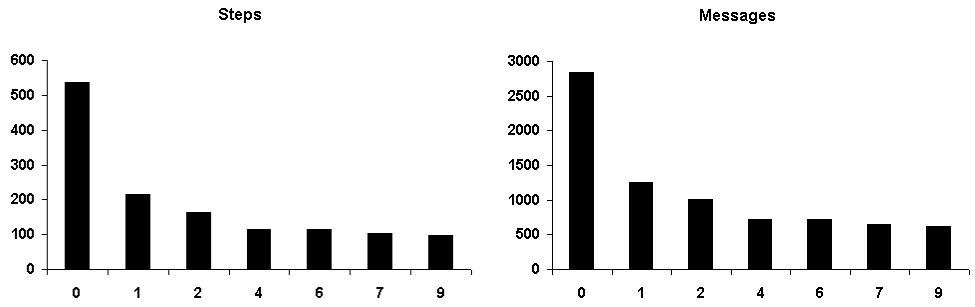


Fig. 8. Two meetings per agent: (a) steps vs. actual additional Nogoods, (b) total number of messages vs. average additional Nogoods.

The present paper differs completely from [Wallace and Freuder2002], in that it solves a search problem. A set of meetings, each with different subsets of the agents, has to be assigned non conflicting time-slots. The assignment problem is exponential in the number of meetings, thus different than a problem with one global meeting to schedule. In the context of a CSP, prior meetings of agents can be represented by different domains for agents. The constraints of the problem are arrival-time constraints, arising from the different locations of the meetings that need assignments. This generates a standard CSP with a clear set of variables, domains of values and constraints among variables (see section 2). When a distributed search process is performed on the MSP, partial assignments are temporary (i.e. the AgentView). This makes the privacy issue less clear. Additional information about the state of assignments carries little information about calendars of agents. Moreover, assignments are dynamic and change during search. In the view of the present paper, the only private information that is revealed during the distributed search process is the meetings in which each agent participates.

Because of the less clear nature of privacy during asynchronous search, the present investigation focuses on volunteered (additional) information. For asynchronous backtracking Nogoods are a clear form of information [Yokoo and Hirayama2000]. In all versions of ABT, differing amounts of Nogoods are kept as constraints discovered during search [Bessiere *et al.*2001]. The choice of the present paper is to volunteer information in the form of additional Nogoods that are sent during search (section 4). Nogoods are units of information about the search space that are of permanent validity. However, their *relevance* to asynchronous backtracking at any given moment can change dynamically (see [Bessiere *et al.*2001]). In other words, the usefulness of additional Nogoods is not guaranteed.

The main experimental result of the present paper is that additional information improves search efficiency. Sending additional Nogoods improves the performance of asynchronous backtracking on the distributed meetings scheduling problem, in 3 different measures. The total number of messages decreases and so does the number of computation cycles and the number of concurrent constraint checks (Figures 6, 7). The largest improvement occurs for adding the first and second additional Nogoods. The marginal gain, as more and more Nogoods are being added, becomes smaller. This is evident for all sets of experiments (see also Figures 8). Problems with a larger number of participating agents (i.e. 24), and a smaller number of meetings per agent are in general easier. It will be interesting to perform further experiments with larger number of meetings per agent. This will need much care during the process of problem generation, as the resulting network is very dense and can become insoluble.

It is interesting to try and clear the impact of the additional information to the present investigation, on the privacy of agents. To this end, one can think of internal constraints of arrival as private information. These constraints represent the set of meetings in which a given agent participates. In the model of the present investigation an arrival-time constraint of agent A_i can be resolved by another agent A_l after receiving a group of immediate Nogoods from A_i. Take for example the following group of immediate Nogoods, sent by agent A_i to agent A_l: $\{(x_k^j = 14:00 \rightarrow x_w^l \neq 14:00), (x_k^j = 14:00 \rightarrow x_w^l \neq 15:00), (x_k^j = 14:00 \rightarrow x_w^l \neq 16:00)\}$ This set of Nogoods can be interpreted by agent A_l as a lower bound on the traveling time of agent A_i from meet-

ing m_k to meeting m_w, in this case two hours. In this way, each immediate Nogood generates knowledge on the meetings and locations of the sending agent.

The effect of additional Nogoods on the efficiency of asynchronous search on general random DisCSPs can also be studied. The generation of additional Nogoods for general DisCSPs can be described as follows. Before sending back a Nogood, for each value in the domain of the destination agent, check whether this value (combined with the left hand side of the Nogood) is in conflict with all remaining values in the domain of the current (sending) agent. It is clear that this computation can be heavy for a general DisCSP and a general set of constraints. For the MSP family of problems the number of allowed domain values is extremely small, due to the equality constraint. Therefore, the computational effort required for additional Nogoods generation in MSP can still reduce the overall concurrent effort.

References

[Bessiere *et al.*2001] C. Bessiere, A. Maestre, and P. Messeguer. Distributed dynamic backtracking. In *Workshop on Distributed Constraints in IJCAI-01*, Seattle, 2001.

[Brito and Meseguer2003] I. Brito and P. Meseguer. Distributed forward checking. In *CP 2003: 9th International Conference*, pages 801–806, Kinsale, Ireland, 2003.

[Dechter2003] R. Dechter. *Constraints Processing*. Morgan Kaufmann, 2003.

[Garrido and Sycara1995] L. Garrido and K. Sycara. Multi-agent meeting scheduling: Preliminary experimental results. In Victor Lesser, editor, *Proc. 1st Intern. Conf. on Multi-Agent Systems (ICMAS'95)*. MIT Press, 1995.

[Ginsberg1993] M. L. Ginsberg. Dynamic backtracking. *Jou. of Art. Intell. Res.*, 1:25–46, 1993.

[Lamport1978] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Comm. of ACM*, 21:558–565, 1978.

[Lynch1997] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1997.

[Meisels *et al.*2002] A. Meisels, E. Kaplansky, I. Razgon, and R. Zivan. Comparing performance of distributed constraints processing algorithms. In *Proc. DCR Workshop, AAMAS-2002*, pages 86–93, Bologna, 2002.

[Meseguer and Jimenez2000] P. Meseguer and M. A. Jimenez. Distributed forward checking. In *CP-2000 Workshop on Distributed Constraint Satisfaction*, Singapore, 2000.

[Prosser1993] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[Sen and Durfee1995] S. Sen and E. H. Durfee. Unsupervised surrogate agents and search bias change in flexible distributed scheduling. In Victor Lesser, editor, *Proc. 1st Intern. Conf. on Multi-Agent Systems (ICMAS'95)*, pages 336–343, San Francisco, CA, 1995. MIT Press.

[Wallace and Freuder2002] R. J. Wallace and E. C. Freuder. Constraint-based multi-agent meeting scheduling: Effects of agent heterogeneity on performance and privacy loss. In M. Yakoo, editor, *AAMAS-02 Workshop on Distributed Constraint Reasoning*, pages 176–182, Bologna, 2002.

[Wallace2003] R. J. Wallace. Reasoning with possibilities in multiagent graph coloring. In *4th Intern. Workshop on Distributed Constraint Reasoning*, pages 122–130, 2003.

[Yokoo and Hirayama2000] M. Yokoo and K. Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents & Multi-Agent Sys*, 3:198–212, 2000.

[Yokoo *et al.*1998] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Trans. on Data and Kn. Eng.*, 10:673–685, 1998.

Multiagent Meeting Scheduling with Rescheduling

Pragnesh Jay Modi and Manuela Veloso

Computer Science Department
Carnegie Mellon University
Pittsburgh PA 15213
{pmodi,mmv}@cs.cmu.edu

Abstract. We are interested in how personal agents who perform calendar management on behalf of their human users can schedule meetings effectively. A key difficulty of concern is deciding when to reschedule an existing meeting in favor of a new meeting. We model the meeting scheduling problem as a special subclass of distributed constraint reasoning (DCR) called the Incremental, Limited Information Exchange Multiagent Assignment Problem (IL-MAP). Key novelties of our approach include i) a focus on incremental scheduling, ii) scheduling under a limited information exchange paradigm and, iii) using models of other agents to schedule more effectively. Our results are the first in DCR to show how models of other agents can be used to improve problem solving performance.

1 Introduction

Meeting scheduling is a time consuming routine task that when delegated to a personal assistant agent promises to significantly reduce daily cognitive load. A key competency of agents who do meeting scheduling is their ability to coordinate schedules such that all attendees of a meeting agree on its start time [3, 9, 2]. The problem is challenging in part because a) each agent chooses its own schedule, i.e., scheduling is *distributed*, b) new meetings are introduced over time, i.e., scheduling is *incremental*, and c) agents are *limited* in the information they can exchange. This article provides an approach to multiagent meeting scheduling using the Distributed Constraint Reasoning (DCR) paradigm [1, 4, 5, 10, 11]. Previous researchers have proposed DCR as a framework for multiagent coordination and considerable progress has been made over the last several years. However, novel techniques are needed to address the challenges described above.

The main idea in this paper is to exploit given models of "scheduling difficulty" with other agents' in order improve meeting scheduling performance. The specific hypothesis we investigate is that an agent can use models of the calendar density of other agents where we assume that the calendar density is correlated with the agent's rank in an organization. This is novel because to our knowledge, existing methods for DCR have not investigated how to take advantage of

learned or given models of other agents to aid in making scheduling decisions. Further, we evaluate our approach in an *incremental scheduling* paradigm, in which new meetings must be scheduled in the context of an existing schedule. Existing DCR approaches have focused primarily on batch problem solving and are not designed for minimizing disruption to an initial given solution. Finally, we assume that communication between agents is limited. We explicitly prohibit the communication of information about variables between agents who are not involved in the variable's value assignment. This restriction is motivated by the meeting scheduling domain in which schedule privacy is a key concern. Existing DCR algorithms typically communicate "context" information which does not adhere to this restriction.

We first formalize the meeting scheduling problem by defining a special form of DCR which we call the Incremental, Limited Information Exchange Multiagent Assignment Problem (IL-MAP). IL-MAP requires agents to assign values to variables where multiple agents must agree on value assignments but are limited in what and to whom information can be communicated. Second, we describe a basic distributed protocol for IL-MAP in which an initiator proposes assignments to others who either agree or refuse the proposed assignments based on their own existing assignments. The protocol conforms to our need for limited information exchange by only communicating allowed information to relevant agents. Third, we use this basic protocol to investigate using models of scheduling difficulty with other agents to increase effectiveness of the multiagent meeting scheduling process. Finally, we demonstrate that our approach improves scheduling effectiveness in an agent organization hierarchy where the lower ranked agents have lower calendar density than the higher ranked agents in the hierarchy.

The multiagent meeting scheduling problem has been previously investigated but methods for making effective rescheduling decisions is lacking. Sen and Durfee [9] formalize the problem and identify a family of negotiation protocols aimed at searching for feasible solutions in a distributed manner. However, rescheduling of existing meetings or modeling of other agents to improve performance is not a major focus. Sen and Durfee also describe a contract-net approach for multiagent meeting scheduling [8] and in this context, rescheduling and cancellation of existing meetings is discussed. The critical issues are raised and a rich decision making framework is presented but is mainly theoretical. Our research represents a further investigation of some of the critical issues raised by them. Freuder, Minca and Wallace [2] have previously investigated meeting scheduling within the DCR framework where the primary motivation was to investigate tradeoffs between efficiency of scheduling and loss of privacy, but not issues of incremental problem solving or agent modeling are not addressed.

2 Meeting Scheduling as Distributed Constraint Reasoning

We view meeting scheduling as a distributed problem in which each agent manages and is responsible for its own calendar. A centralized approach is also possible in which a single server is assumed to have access to each agent's calendar and makes scheduling decisions for all agents. However, a centralized approach has several drawbacks including that it requires agents to reveal potentially private calendar information to the central server.

We use the Distributed Constraint Reasoning (DCR) paradigm [11] to model distributed meeting scheduling. DCR is defined by a set of variables where each variable is assigned to an agent who has control of its value, and agents must choose values for their assigned variables so that a given set of constraints are satisfied or optimized. Constraints between variables assigned to the same agent are called *intra-agent* constraints, while constraints between variables assigned to different agents are called *inter-agent* constraints. To ensure that inter-agent constraints are satisfied, agents must coordinate their choice of values for variables through a communication protocol.

2.1 The Multiagent Assignment Problem (MAP)

In this section, we introduce an important subclass of DCR which we call the *multiagent assignment problem* (MAP) . In MAP, we assume that agents must map elements from one set, which are modeled as the variables, to elements of a second set, which are modeled as the values. Importantly, we assume multiple agents need to agree on the assignment of a value to a given variable. Since decision-making control is distributed among the agents, this "agreement" requirement raises many unique challenges.

We define MAP as follows.

- $\mathcal{A} = \{A_1, A_2, ..., A_n\}$ is a set of *agents*.
- $\mathcal{V} = \{V_1, V_2, ..., V_m\}$ is a set of *variables*.
- $\mathcal{D} = \{d_1, d_2, ..., d_k\}$ is a set of *values*.
- *participants*$(V_i) \subseteq \mathcal{A}$ is a set of agents who are assigned the variable V_i.
- *vars*$(A_i) \subseteq \mathcal{V}$ is a set of variables assigned to agent A_i.
- For each variable V_i, an inter-agent *agreement* constraint is satisfied if and only if the same value from $\mathcal{D}$ is assigned to V_i by all the agents in *participants*(V_i).
- For each agent A_i, an intra-agent *mutual exclusion* constraint is satisfied if and only if no value from $\mathcal{D}$ is assigned to more than one variable in *vars*(A_i).

MAP has some similarities to the classical "assignment problem" from combinatorial optimization research[7]. Two key differences are that a) MAP requires distributed agents to agree on assignments and b) MAP does not yet

model degrees of solution quality, only valid and invalid solutions. Further extension of MAP to model optimization problems is important future work.

2.2 Meeting Scheduling as MAP

We describe the multiagent meeting scheduling problem followed by its formulation as a MAP. Meeting scheduling requires meetings to be paired with timeslots subject to three constraints: a) each meeting is assigned to exactly one timeslot, b) each timeslot is paired with no more than one meeting, and c) all the attendees of a given meeting agree on its assigned timeslot. The goal of the following model is to represent these three constraints.

We define the meeting scheduling problem as follows.

- $\mathcal{A} = \{A_1, A_2, ..., A_n\}$ is a set of agents.
- $\mathcal{M} = \{M_1, M_2, ..., M_m\}$ is a set of meetings. We assume each meeting has the same duration d.
- *attendees*$(M_i) \subseteq \mathcal{A}$ are the attendees of meeting M_i.
- *meetings*$(A_i) \subseteq \mathcal{M}$ are the meetings of which A_i is an attendee.
- *initiator*$(M_i) \in attendees(M_i)$ is the designated initiator of meeting M_i.
- $\mathcal{T} = \{T_1, T_2, ..., T_p\}$ is a set of discrete non-overlapping contiguous timeslots of length d.
- $\mathcal{S}_{init} = \{S_1, S_2, ..., S_n\}$ is a set of calendars. Each S_i is a mapping from the meetings in *meetings*(A_i) to timeslots in $\mathcal{T}$. A calendar S_i is *valid* if and only if a) each meeting is mapped to exactly one timeslot and no timeslot has more than one meeting mapped to it, and b) for each meeting M_k and for all attendees $A_i, A_j \in$ *attendees*(M_k), $S_i(M_k) = S_j(M_k)$. That is, the calendars of all attendees of a meeting agree on its assigned timeslot.

The representation of meeting scheduling as MAP is straightforward. The set of MAP variables $\mathcal{V}$ is given by the set of meetings $\mathcal{M}$ and the set of MAP values $\mathcal{D}$ is given by the set of timeslots $\mathcal{T}$. The *participants* of variable V_i correspond to the *attendees* of meeting M_i. The MAP intra-agent mutual exclusion constraint prevents a timeslot from being double-booked and the inter-agent agreement constraint ensures that meeting attendees agree on the time.

Figure 1 illustrates the multiagent assignment problem (and its solution) with three agents A_1, A_2, A_3, five meetings M_1,M_2,M_3,M_4,M_5 and four timeslots. Note that for each agent, each meeting is assigned to a different value in order to satisfy the intra-agent mutual exclusion constraint. Between agents, the variables corresponding to the same meeting are assigned the same value in order to satisfy the inter-agent agreement constraint.

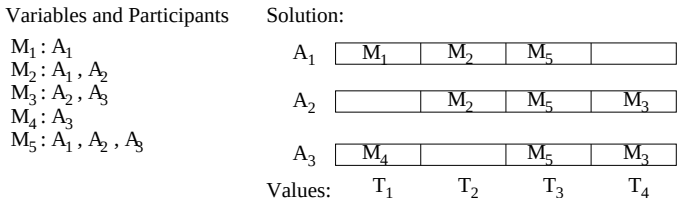


Fig. 1. Meeting Scheduling as the Multiagent Assignment Problem.

2.3 IL-MAP: MAP in Incremental, Limited Information Exchange Domains

We further extend the scheduling problem to introduce the IL-MAP problem in which agents must solve MAP in an incremental fashion while limiting the information they can exchange. These two features are described next.

Incremental In an incremental MAP, new variables and associated constraints are added to the problem over time and must be integrated into an existing assignment. In meeting scheduling for example, new meetings arise over time and must be scheduled in the context of an existing calendar. In addition to the elements of MAP defined above, in the incremental version we are also given:

- $S_{init} = \{(V_1, d_i), (V_2, d_j), ..., (V_m, d_k)\}$ is an initial solution.
- V_{m+1} is a new variable to be assigned a value.
- *participants*$(V_{m+1}) \subseteq \mathcal{A}$ is a set of agents who are assigned the variable V_{m+1}.

The key difficulty that arises in incremental MAP is that existing assignments may need to be changed in order to successfully accommodate the new variable but it is difficult to determine in advance which changes will result in a set of valid schedules.

Limited Information Exchange Although agents must exchange some information in order to obtain feasible solutions, the information exchange process is limited due to the distributed nature of the problem. In particular, we assume the following condition.

- Agents do not communicate information about a variable to agents who are not participants in that variable.

For example, the id of a variable, its current value, or the participants in the variable are not communicated between agents who are not both participants in the variable. A key challenge is to schedule effectively under this condition.

```
procedure initiate(M_j):
(1)   initiator(M_j) ← A_i
(2)   t ← GetTimeslot(M_j)
(3)   if t is null:
(4)      return
(5)   status(M_j,t) ← PENDING
(6)   for each A_k ∈ attendees(M_j):
(7)      send (PROPOSE, M_j, t, A_i) to A_k

procedure when received(PROPOSE,
 M_j, t, initiator):
(8)   if exists M_k where status(M_k, t) is PENDING:
(9)      reply ← IMPOSSIBLE
(10)  else if exists M_k where
      status(M_k, t) is CONFIRMED:
(11)     if BumpingRule(M_j, M_k) is true:
(12)        status(M_k,t) ← BUMPED
(13)        status(M_j,t) ← PENDING
(14)        reply ← PENDING
(15)     else:
(16)        reply ← IMPOSSIBLE
(17)  else:
(18)     status(M_j,t) ← PENDING
(19)     reply ← PENDING
(20)  send (REPLY, M_j, t, reply, A_i) to initiator

procedure when received(REPLY,
 M_j, t, reply, Attendee):
(21)  agentView(M_j,t,Attendee) ← reply
(22)  if exists t' where ∀A_k ∈ attendees(M_j),
      agentView(M_j, t',A_k) is PENDING
      and status(M_j, t') is PENDING
(23)     status(M_j, t') ← CONFIRMED
(24)     resolved(M_j)
(25)     for each A_k ∈ attendees(M_j):
(26)        send (CONFIRM, M_j, t') to A_k
(27)     if exists M_k where status(M_k,t') is BUMPED:
(28)        reschedule(M_k)
(29)  else:
(30)     t" ← GetTimeslot(M_j)
(31)     if t" is null:
(32)        resolved(M_j)
(33)        for each A_k ∈ attendees(M_j):
(34)           send (FAIL, M_j) to A_k
(35)     else:
(36)        status(M_j,t") ← PENDING
(37)        for each A_k ∈ attendees(M_j):
(38)           send (PROPOSE, M_j, t", A_i) to A_k

procedure when received(CONFIRM, M_j, t):
(39)  status(M_j, t) ← CONFIRMED
(40)  resolved(M_j)
(41)  if exists M_k where status(M_k,t) is BUMPED:
(42)     reschedule(M_k)

procedure when received(FAIL, M_j):
(43)  resolved(M_j)

procedure when received(RESCHEDULE, M_j):
(44)  reschedule(M_j)

procedure reschedule(M_j):
(45)  if A_i equals initiator(M_j):
(46)     if exists t where status(M_j,t) is
         BUMPED or CONFIRMED:
(47)        status(M_j, t) ← IMPOSSIBLE
(48)     for each A_k ∈ attendees(M_j):
(49)        for each t where agentView(M_j, t,A_k) is
           IMPOSSIBLE or PENDING:
(50)           agentView(M_j, t ,A_k) ← POSSIBLE
(51)     initiate(M_j)
(52)  else:
(53)     send (RESCHEDULE, M_j) to initiator(M_j)

procedure resolved(M_j):
(54)  for each t where status(M_j, t) is PENDING:
(55)     status(M_j, t) ← POSSIBLE
(56)     if exists M_k where status(M_k, t) is BUMPED:
(57)        status(M_k, t) ← CONFIRMED
```

Fig. 2. Algorithm for Agent A_i

3 A Solution Technique for IL-MAP in Meeting Scheduling

We are interested in solution techniques for IL-MAP in the context of distributed meeting scheduling. We first describe a basic negotiation framework upon which our techniques are applied. Next, we describe the problem of rescheduling existing meetings. Finally, we present our approach for making this rescheduling decision effectively.

3.1 Basic Negotiation Protocol

Sen and Durfee [9] describe a basic negotiation protocol for meeting scheduling in which agents negotiate in *rounds*. Each meeting has a designated initiator who manages the negotiation of the meeting by proposing times and collecting responses from the other attendees in a sequence of rounds. In each round, each attendee responds with a PENDING (accept) or IMPOSSIBLE (reject) message for the proposed time. The initiator collects the responses in each round and does a set intersection to try to find a mutually acceptable time. If a time is found, the meeting is CONFIRMED (scheduled) in one additional round and the process terminates. Otherwise, the process continues in rounds until the initiator runs out of times to propose in which case the process terminates with failure.

We adopt a variant of this basic protocol in which attendees may tentatively bump a CONFIRMED meeting in favor of a new meeting in order to decrease the possibility of scheduling failure. We say it is tentatively bumped because an agent waits until the new meeting is confirmed in the bumped timeslot before initiating rescheduling of the bumped meeting. If the new meeting is confirmed in some other slot or fails to be scheduled, the bumped meeting is re-instated into its original slot. If an agent needs to reschedule a meeting of which it is not the original initiator, it sends a RESCHEDULE message to the initiator, who will be responsible for restarting a negotiation episode for the meeting.

Details of the algorithm are shown in Figure 2. Two functions $GetTimeslot$ and $BumpingRule$ are purposely left unspecified in Figure 2. $GetTimeslot$ returns a free timeslot from the calendar or null if one does not exist. This function encapsulates a local optimization routine which ranks all the free timeslots according to a complex set of user preferences, and returns the top ranked time. Further discussion is out of scope of this paper and we refer the reader to [6] for more details. The $BumpingRule$ function returns true or false, and encapsulates the reasoning of the agent about whether one meeting should be bumped for another. A technique for making this decision is described in rest of this next section.

3.2 The Problem of When to Reschedule

A key algorithmic decision to be made is when to bump an existing meeting in favor of a proposed meeting. More specifically, an attendee A_i must make a rescheduling decision when it receives a proposal for meeting M_1 at time slot T_1 but A_i already has a meeting M_2 confirmed in slot T_1. A_i has to decide between accepting the proposal or rejecting it. If the agent decides to accept the proposal, it may need to reschedule M_2 with the other attendees. This rescheduling may cause the other attendees in turn to bump other meetings, which can result in cascading disruption costs throughout the set of agents. The alternative is for A_i to reject the proposal for M_1, but this entails risk also because the scheduling of M_1 may ultimately fail. It is difficult to determine in advance which is the better decision because other people's schedules are not directly observable.

Fixed strategies such as always rejecting or always bumping fail to be effective. Table 1 shows a comparison of the average performance of the two fixed strategies. (The exact experimental set-up is described in more detail in Section 5. These results are with 20 agents who have initial calendar densities of 85%.) The "failures" column shows that for the Never-Bump strategy a mutually free timeslot could not be found in 49 out of 50 cases. The "timeouts" column shows that for the Always-Bump strategy the negotiation failed to terminate after a given amount of time (10 minutes) in 50 out of 50 cases. In these cases, a cascading effect caused many meetings to be bumped until ultimately a maximum time limit was reached.

Table 1. Empirical analysis of two strawman strategies illustrates the need for intelligent rescheduling techniques

Strategy	Rounds	Msgs	Failures	Timeouts
Never-Bump	6.88	44	49/50	–
Always-Bump	614	2736	–	50/50

3.3 Modeling Scheduling Difficulty

We propose a method for making rescheduling decisions in which agents use a model of "scheduling difficulty" with other agents. Such models can be given to an agent or they can be learned by the agent over time. In this paper we are interested in how a scheduling difficulty model, once obtained, can be used by an agent to improve rescheduling decisions. Also, we note that more complex models of scheduling difficulty are possible than the one presented here. However, such models require more effort to construct and are not guaranteed to

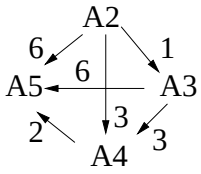


Fig. 3. A model of relative scheduling difficulty with four agents A_2,A_3,A_4 and A_5.

improve scheduling. We opt for the following model which is computationally convenient and can be shown to improve scheduling performance.

Let SD_i be a number denoting the *scheduling difficulty* of an agent A_i, i.e., if $SD_i > SD_j$, then scheduling a meeting with agent A_i is expected to be more "difficult" than with agent A_j. SD is measured in scheduling difficulty "units". We use this factor to encapsulate the many relevant features that contribute to scheduling difficulty with another agent. Assuming that each agent is operating on behalf of a human, SD could take into account factors such as stubbornness or accessibility to email communication. We will consider calendar density as associated with position in a organization as a key factor. We define $k_{i,j}$ as the relative difficulty for scheduling a meeting with A_i versus scheduling a meeting with A_j. It makes natural sense for this relation to be multiplicative and transitive. That is, for three agents A_2, A_3, A_4, we require that $k_{2,3} \times k_{3,4} = k_{2,4}$.

Example: Figure 3 shows A_1's model of relative scheduling difficulty with a group of four other agents A_2,A_3,A_4, and A_5. The arrow from A_3 to A_4 with magnitude 3 represents the relation $SD_{A_4} = 3 \times SD_{A_3}$, i.e., scheduling a meeting with A_4 is 3 times "more difficult than" scheduling a meeting with A_3.

Given a model of scheduling difficulty, we now have a way to define a decision rule for when to reschedule a meeting in favor of another. Given a meeting M_j, A_k computes the difficulty of scheduling M_j as

$$\textit{Difficulty}(M_j) = \sum_{A_i \in attendees(M_j) - \{A_k\}} SD_i \tag{1}$$

Finally, the bumping rule is given as follows. An agent bumps a meeting M_j in favor of a meeting M_i if and only if the following *BumpingRule*(M_i, M_j) evaluates to *true*:

$$\textit{Difficulty}(M_j) < \textit{Difficulty}(M_i) \tag{2}$$

4 Example of a Meeting Scheduling Negotiation

We describe an example scheduling negotiation episode involving an agent A_1. Figure 3 shows A_1's model of relative scheduling difficulty with four other agents A_2,A_3,A_4, and A_5. Details of the negotiation using this model is shown in Figure 4. Each box represents the state of agent A_1's calendar at a given time. Arrows denote incoming and outgoing messages. Each message is 3-tuple of meeting id, time, and meeting status, where status is either *possible, pending, bumped, confirmed* or *impossible*. In this example, $attendees(M1) = \{A_1, A_2, A_3\}$, $attendees(M2) = \{A_1, A_4\}$, and $attendees(M3) = \{A_1, A_5\}$.

At time 1, A_1 has meeting M1 currently confirmed at time 10 am and receives a request from A_4 who is the initiator of meeting M2. The time proposed is 10 am, which conflicts with M1. A_1 must now decide whether to reject A_4's proposal, or accept it and bump meeting M1. Referring to Figure 3 and Equation 1, A_1 computes the scheduling difficulty of M1 as $SD_2 + SD_3 = 1 + 1 = 2$ and the scheduling difficulty of M2 as $SD_4 = 3$. Since *Difficulty*$(M1) <$ *Difficulty*$(M2)$, A_1 decides to bump.

At time 2, A_1 changes the status of M1 to bumped, and sets status of M2 as pending for 10 am, and a response is sent to A_4. At time 3, as an example of concurrency, A_1 receives a request from A_5 for 10 am for a new meeting M3. At time 4, A_1 responds impossible since 10 am is already pending for M2. Pending meetings are never bumped (only confirmed meetings can be bumped). At time 5, A_1 hears back from A_4 that M2 should now be confirmed for the previously proposed time of 10 am. At time 6, A_1 sets the status of M2 to confirmed, and begins the rescheduling of M1 by proposing a new time to the other attendees A_2 and A_3. (This example has assumed that A_1 is the initiator of M1. If it were not, then in our protocol, A_1 would have sent a message to the initiator of M1 indicating that 10 am is now impossible, and the initiator would be responsible for restarting the negotiation and rescheduling M1). At time 7, A_1 hears back from A_2 that 11 am is pending in its calendar for meeting M1. At time 8, A_1 records this information in its current state. At time 9 and 10, A_1 hears back from A_3 and records the response. At time 11, A_1 has now heard back from all attendees for meeting M1, and all have agreed on 11 am. A_1 sends the final confirmation message to all attendees. We end the example here, but realize that since A_2 or A_3 may have bumped meetings at 11 am to accommodate A_1's request for meeting M1, the scheduling episode may not be over.

5 Experimental Results

We present experimental results comparing rescheduling strategies that use a model of "scheduling difficulty" with other agent versus strategies that do not. In

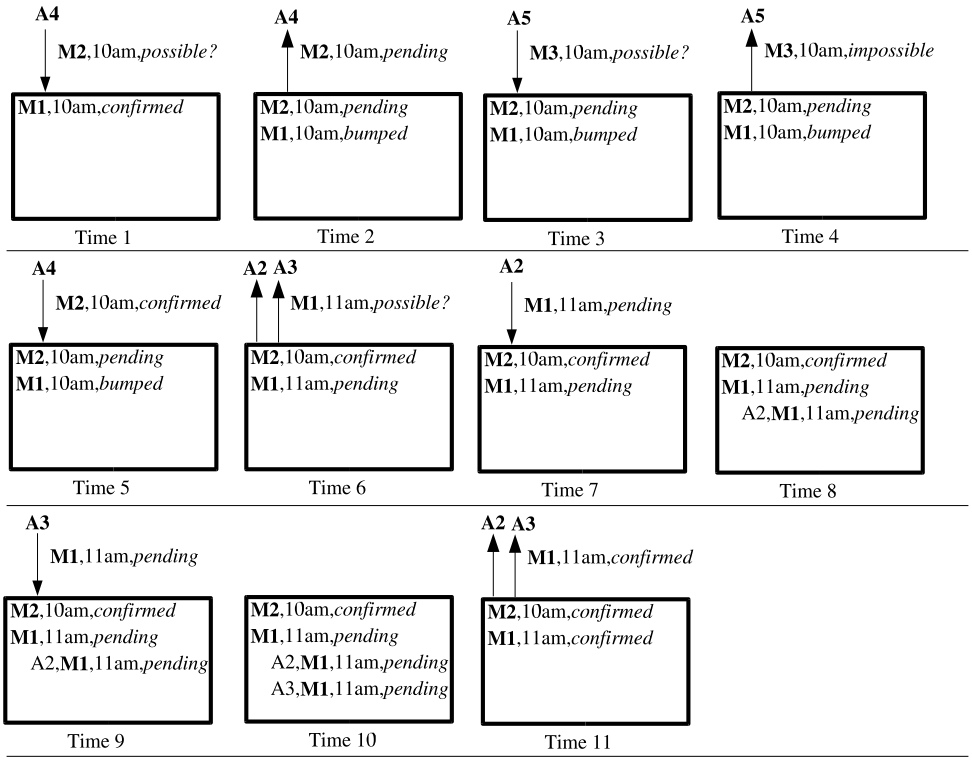


Fig. 4. An example negotiation between an agent and four other agents A_2,A_3,A_4 and A_5.

the first strategy, denoted Att, agents simply compare the number of attendees and bump the meeting with fewer attendees when there is a conflict between two meetings. They do not use knowledge about other agents in making their bumping decisions. In the second strategy, denoted SD, we assume that agents know the rank of other attendees and use this information to make bumping decisions, i.e., they can assign a "scheduling difficulty" to each attendee.

5.1 Experimental Setup

We evaluate each strategy by averaging measurements over a number of "runs". Each run consists of two phases: a problem generation phase followed by a problem solving phase. We describe each phase in turn. In our experiments, we report measurements from the problem solving phase only.

Phase 1 The problem generation phase is centralized. We automatically generate a set of agents $\mathcal{A}$ each with a desired initial schedule density. Each agent's calendar has 50 timeslots to simulate a 5 day 10-hour work week. Next, we automatically generate and schedule meetings between random subsets of the agents until all calendars are filled to their desired density. The attendees of a given meeting are chosen according to a uniform random distribution. The number of attendees for a given meeting is chosen according to a distribution in which meetings of more people are less likely than meetings with fewer people. Every meeting has at least two attendees. Finally, we generate one additional new meeting M_{m+1} that must be scheduled in the problem solving phase. The attendees of the new meeting are chosen to be a random subset of the agents. In our experiments, the number of attendees of the new meeting is fixed to 4. One of them is randomly chosen to be the initiator. Every generated problem is ensure to have a solution.

Phase 2 The problem solving phase is completely distributed. The goal is to find a timeslot for the new meeting $\{M_{m+1}\}$ while successfully rescheduling any bumped existing meetings. That is, the goal is to find an assignment of timeslots to meetings in $\mathcal{M} \cup \{M_{m+1}\}$ that satisfy the intra-agent and inter-agent constraints. We measure number of *failures* which is defined as the number of meetings in $\mathcal{M} \cup \{M_{m+1}\}$ unassigned a timeslot after a given amount of time. Failures may occur either because the initiator gives up scheduling the meeting or a max time elapses. Note that the number of failures in a given run can be greater than one when multiple meetings are bumped and fail to be rescheduled.

5.2 Experiments in a Hierarchical Agent Organization

Human organizations typically have hierarchies in which higher ranked people have denser calendars than lower ranked ones. We hypothesis that the density of

an agents calendar and thus her organizational rank, is a good predictor of the difficulty of scheduling with that person.

To evaluate our hypothesis, we begin with an extreme case – a simple two-level organization hierarchy. We divide agents into two equal size groups of "busy" and "not busy" agents, where the initial density of schedules is fixed to 90 percent and 30 percent, respectively. The scheduling difficulty model used by the SD strategy in this scenario is defined as $SD_{busy} = 3 \times SD_{nonbusy}$.

Figure 5 contrasts two strategies as we increase the total number of agents. The graph shows the SD strategy is more effective in terms of preventing scheduling failures than the Att strategy. At 50 agents, the SD strategy results in a failure rate of 0.28 on average, while the simpler strategy Att results in 0.76 failures on average. Failure rate is computed by summing the number of failures over all runs and then dividing by the total number of runs. We do 50 runs for each datapoint where each run follows the methodology described above. This graph shows that the use of our scheduling difficulty model is able to reduce scheduling failures. Also, the high failure rate caused by uncontrolled cascading of bumps, as we saw in Table 1 for the Always-bump strategy, is avoided.

Next, we evaluate the effect of varying our scheduling difficulty model in the busy/non-busy hierarchy. We use a scheduling difficulty model defined as $SD_{busy} = k \times SD_{nonbusy}$ and examine the effects of varying k. The same set of scheduling problems are used for each value of k, i.e., the only difference is the rescheduling decision rule used by the agents. We expect that changes in performance will level off as the scheduling difficulty multiplier k is increased. This is because after some point, an increase in k no longer modifies an agents rescheduling decisions. For example, a meeting M_1 with 4 non-busy attendees will be bumped in favor a meeting M_2 with one busy attendee when $k = 5$. M_1 will continue to be bumped if k is increased. Thus increasing k should stop having an effect on agent decision making at some point. Figure 6 shows empirical data consistent with our hypothesis. An organization of 10 agents was used. Each datapoint represents the average over 50 runs. The graph shows that the effect on failure rate levels off as predicted.

Finally, we experiment with a more complex scheduling difficulty model where there are four levels rather than just two. We use the organization hierarchy shown in Figure 7 with 8 agents in each level, for a total of 32 agents. We experiment with four levels with initial schedule densities of 90,70,50,30 percent respectively. We define $SD_{L_i} = 2 \times SD_{L_{i+1}}$. That is, the difficulty of scheduling with an agent at level i is twice as difficult as scheduling with an agent at level $i + 1$. The empirical results over 500 runs are shown in Figure 8. The failure rate is reduced from 0.28 using the Att strategy to 0.02 using the

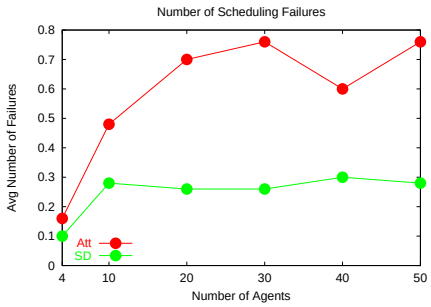


Fig. 5. Comparison of two rescheduling strategies (Att, SD) as a function of organization size. The average number of meetings that failed to be scheduled is shown.

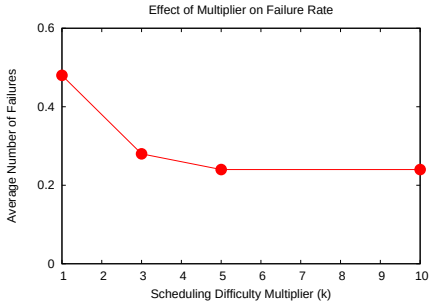


Fig. 6. Effect of increasing value of scheduling difficulty multiplier on scheduling performance. The average number of meetings that failed to be scheduled is shown.

SD strategy. We can conclude that the SD strategy significantly reduces the number of scheduling failures.

6 Conclusion

We have modeled the multiagent meeting scheduling problem as a form of distributed constraint reasoning in which agents must assign a set of values to a set of variables. We presented a novel approach to the problem in which agents use given or learned "scheduling difficulty" models of other agents in order to decide when to change their existing assignments in order to accept proposals from others. We have shown that this approach controls the amount of bumping so that the negotiation is able to terminate in a given amount of time, while also reducing the scheduling failure rate over an alternative approach that does not take into account such models. In future work, we are interested in how an agent can automatically learn these models from past negotiation history.

References

1. C. Bessire, A. Maestre, and P. Meseguer. Distributed dynamic backtracking. In *International Joint Conference on AI Workshop on Distributed Constraint Reasoning*, 2001.

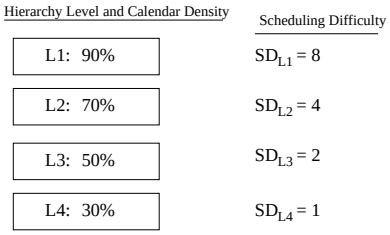


Fig. 7. Agent hierarchy where higher ranked agents have higher calendar densities.

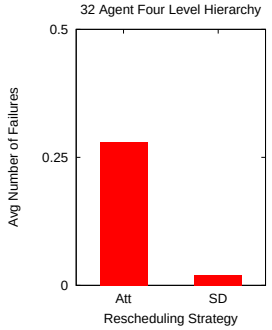


Fig. 8. Comparison of two rescheduling strategies (Att, SD) in a four level organization hierarchy. The number of meetings that failed to be scheduled (average over 500 run) is shown.

2. Eugene C. Freuder, Marius Minca, and Richard J. Wallace. Privacy/efficiency tradeoffs in distributed meeting scheduling by constraint-based agents. In *IJCAI-2001 Workshop on Distributed Constraint Reasoning*, 2001.
3. Leonardo Garrido and Katia Sycara. Multi-agent meeting scheduling: Preliminary experimental results. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)*. The MIT Press: Cambridge, MA, USA.
4. R. Mailler and V. Lesser. A mediation based protocol for distributed constraint satisfaction. In *The Fourth International Workshop on Distributed Constraint Reasoning*, 2003.
5. P. J. Modi, W. Shen, M. Tambe, and M. Yokoo. Adopt: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 2004.
6. P. J. Modi, M. Veloso, S. Smith, and J. Oh. Cmradar: A personal assistant agent for calendar management. In *Agent Oriented Information Systems, (AOIS) 2004*, 2004.
7. Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Prentice-Hall, Inc., 1982.
8. Sandip Sen and Edmund Durfee. A Contracting Model for Flexible Distributed Scheduling. *Annals of Operations Research*, 65:195–222, 1996.
9. Sandip Sen and Edmund H. Durfee. A formal study of distributed meeting scheduling. In *Group Decision and Negotiation*, volume 7, pages 265–289, 1998.
10. M.C. Silaghi, D. Sam-Haroud, and Boi Faltings. Asynchronous search with aggregations. In *Proceedings of National Conference on Artificial Intelligence*, 2000.
11. M. Yokoo. *Distributed Constraint Satisfaction:Foundation of Cooperation in Multi-agent Systems*. Springer, 2001.

On the Evaluation of DisCSP Algorithms*

Ismel Brito, Fernando Herrero, and Pedro Meseguer

Institut d'Investigació en Intel.ligència Artificial
Consejo Superior de Investigaciones Científicas
Campus UAB, 08193 Bellaterra, Spain.
{ismel|fhcarron|pedro}@iiia.csic.es

Abstract. Not every research paper in DisCSP evaluates algorithms in the same way. Motivated by this fact, we revise some elements of the area of distributed algorithms as well as distributed constraints, which can help to develop a well-founded methodology for evaluation of DisCSP algorithms. Although preliminary, we suggest a number of points which should be considered in such methodology.

1 Introduction

In this paper we aim at collecting a number of thoughts and ideas about the task of how evaluate algorithms for solving DisCSP. As researchers on distributed constraint satisfaction, we often develop new versions of existing procedures, we devise new heuristics and we produce new solving algorithms. To assess the practical importance of these new developments, their evaluation is a crucial point. Facing this issue, we often consider questions like,

- what is the most adequate environment to test our algorithms?
- on which benchmarks should they been evaluated?
- which are the most adequate parameters to measure algorithmic efficiency?

Often, different research groups have different answers to these questions. Our goal is to achieve a consensus in the community of distributed constraint satisfaction, in order to establish a common accepted *methodology* on the way algorithms should be evaluated. Obviously, this methodology should follow standard methods in the area of *distributed algorithms* (see [4] for a comprehensive review of this area). In addition, since constraint solving is NP-complete, many solving algorithms have the same worst-case complexity. To really evaluate these algorithms in practice, we have to identify some parameters whose measure could give an idea of the amount of resources used in the algorithm execution. The methodology has to answer two types of questions. First, to define *what* parameters should be measured (total CPU time, number of cycles, concurrent constraint checks, number of messages exchanged, etc.). Second, to define *how* this can be measured, in a double sense: *on which environment* evaluation is performed (reality vs. simulation, several computers vs. one computer), and *on which benchmarks*

* This research is supported by the REPLI project TIC-2002-04470-C03-03.

(distributed random, distributed versions of existing CSP, specific DisCSP appplications, etc.). As a consequence, we expect that comparison among different approaches would be facilitated, and the value of scientific communication would be promoted.

In the following, we discuss some of these issues (the question of benchmarks is not considered) based on our experience. We strongly believe that other research groups can provide valuable ideas and suggestions, and we urge them to do so.

2 Preliminaries

There are several definitions of distributed constraint satisfaction problems. Without trying to be exhaustive, we think that all of them share the following idea. A *distributed constraint satisfaction problem* (DisCSP) is a CSP which is distributed among several agents. Each agent contains a part of the problem, but no agent contains the whole problem. Some overlapping may exist among agents, although no two of them can contain exactly the same initial information. Because some reasons (privacy, size, etc.), the information of each agent cannot be transferred into a central server, where the whole problem could be solved by classical, centralized CSP solving methods. In the distributed setting, the task is to find a solution of the problem (an assignment of all the variables satisfying all constraints), by exchanging messages among agents.

Depending on the model that we assume about the timing of events in the distributed system, we obtain different types of algorithms. In [4], three timing models are considered, which are informally described as follows:

1. *The synchronous model*. "This is the simplest model to describe, to program and to reason about. We assume that components (agents) take steps simultaneously, that is, that execution proceeds in synchronous rounds."
2. *The asynchronous model*. "We assume that separate components (agents) take steps in arbitrary order, at arbitrary relative speeds."
3. *The partially synchronous model*. "We assume some restrictions on the relative timing of events, but execution is not completely lock-step as it is in the synchronous model."

These three timing models generate three types of algorithms for DisCSP solving. Broadly speaking, a synchronous algorithm is based on the notion of *privilege*, a token that is passed among agents. Only one agent is active at any time, the one having the privilege, while the rest of agents are waiting [1]. When the process in the active agent terminates, it passes the privilege to another agent, which now becomes the active one. In an asynchronous algorithm every agent is active at any time, and it does not have to wait for any event. A partially synchronous algorithm in in between of these two types. An agent running a partially syncronous algorithm may require to wait for some special event, but not for every event.

To solve a DisCSP instance, the three types of algorithms differ in their functionality and efficiency. Considering functionality, asynchronous algorithms are the most general

[1] Except for special topological arrangements of the constraint graph. See [2] for a synchronous algorithm where several agents are active concurrently.

and portable, because they impose no assumptions on the timing of computation steps. Usually, they are more robust and offer more privacy than the other two types. Regarding efficiency, as the amount of resources required to compute a solution, there is some debate on which type of algorithm is more efficient. We come back on this issue in the Section 5.

3 Evaluation

Two complexity measures, on time and on communication, are proposed in [4] for distributed algorithms that exchange messages. *Time complexity* aims at bounding the time required to compute a global solution by the whole system. *Communication complexity* considers the amoung of network resources needed to achieve a solution.

3.1 Time Complexity

For synchronous algorithms, [4] proposes using the number of rounds required to find a solution as the time complexity measure. For asynchronous algorithms, [4] requires to have an upper bound on the time between succesive chances of a task to perform a step. This is called a timed execution. The time of the event is the supremum of the times that can be assigned to such an event in all timed executions. Since CSP solving is NP-complete, this worst-case expression is exponential and does not help in clarifying the relative efficiency of different algorithms.

Alternatively, [3] proposes a new measure of time complexity as counting the number of constraint checks that cannot be performed concurrently when solving a DisCSP. A constraint check occurs when a value tuple is checked against a constraint. In classical CSP it is considered an atomic operation, which has to be performed for (almost) all constraint algorithms, so the number of constraint checks is a good estimation of the search effort. Inspired in the logical clocks of Lamport [5], in [3] the number of concurrent constraint ckecks is computed as follows. Each agent keeps a counter of its own performed constraint checks, and every message that it sends contains the value of that counter (when it was sent). When the receiver gets that message, it updates its own counter to the maximum between its counter and the counter contained in the message. When the algorithm stops, the maximum of the counters is the total concurrent constraint checks, and approximates the size of the longest sequence of checks that cannot be done concurrently.

At the end of the search, the number of concurrent constraint checks performed approximates the runtime of the algorithm if it is assumed that the elapsed time between two constraint checks not performed concurrently is approximately the same. However, this assumption does not hold in presence of random delays or for partially synchronous algorithms with unbounded waiting episodes. In this last case, waiting episodes can be counted at agent level. Following a similar approach to concurrent constraint checks, we can assess the longest sequence of waiting episodes which cannot be performed concurrently.

Other measures can provide complementary information. For instance, the distribution of constraint checks really performed by agents in the network gives some idea of how balanced is search effort among agents.

3.2 Communication Complexity

For the three timing models considered, [4] considers the total number of messages exchanged as the measure of communication complexity. How messages are counted depends on the communication model used, described in Section 3.3. This is also the common position of the distributed constraints community.

The size of messages can also be taken into account as secondary measure, following [4]. The cost of sending a message is the cost of setting the communication link plus the cost of properly sending the message. The cost of setting the communication link is paid when the first message is sent through that link. The cost of properly sending a message depends on its length (the message size plus the header added by the communication software). So message size has to be considered, especially when comparing algorithms exchanging messages whose sizes differ in more than a constant.

Assuming the Unicast communication model (see Section 3.3), the idea of concurrent constraint checks can be applied to messages. Each agent keeps a counter of the sent messages, and every message contains the value of that counter when it was sent. When the receiver gets that message, it updates its own counter to the maximum between its counter and the counter contained in the message. We call this value concurrent messages, and gives an idea of the length of the longest sequence of messages that cannot be done concurrently.

Other measures can provide complementary information. For instance, the distribution of the number of messages sent/received by agents in the network gives some idea of how balanced is the communication among agents.

3.3 Communication Model

It is often the case that algorithm description and analysis do not consider the underlying communication model. However, a real-case study should take this into account, as the communication costs may vary depending on which model is used. We analyze two communication models:

1. *Unicast* (also called *send/receive* or *point-to-point* communication). On a unicast network, messages are sent one by one to each of the recepients, thus requiring linear resources on the number of agents. This is the common model used in experiments and simulations.
2. *Multicast*. On the other hand, advantage could be taken from multicast networks, such as IP networks, on which agents can subscribe to a group and messages sent to that group do not imply any additional cost per agent. This model provides constant time and resources, irrespective to the number of recepients.

Since this is an implementation issue, it makes sense to reflect which of these models was actually used when presenting experimental results. It is not uncommon to consider “broadcast” communication as a single process, when in fact the implementation means sending one message to each receiver.

4 Simulator

Ideally, to evaluate a new algorithm one should have n dedicated processors connected to a common network on which tests would be done. However, this setting is often not available in most of our labs. Even if there is a number of computers available, the workload of each computer and the load of the communication network are out of the control of the experimenter, and these aspects have a significant impact on the efficiency of the algorithms. Because of that, we consider that simulation into a single computer is a suitable alternative to make the tunning and most of the experimentation for DisCSP algorithms. After that, some algorithms can be tested on a real setting, assuming the resources needed to perform a field test. In the following, we consider the different options for DisCSP algorithms when are evaluated by simulation on a single computer.

Usually, DisCSP algorithms are described in terms of agents. An agent is an autonomous entity that contains a part of the problem, it is able to perform its own reasoning process and to communicate with other agents. In a multi-task computer (for instance, a desktop with Linux operating system (OS)), a direct option is to implement each agent as a different task, all having the same priority. The OS scheduler is in charge of activating / desactivating the agents, that take control of the CPU as any other task in the system. Communication among agents is performed using standard task communication facilities (usually implemented using disk storage). This approach is relatively simple to implement but present some drawbacks. First, it depends on the OS, so results obtained in computers with different OS could not be directly comparable. Second, even using the same computer and the same implementation, it is difficult to reproduce exactly the same results when repeating the same experiments. There are some sublte factors (such as the mail server, the network load, the disk storage) which change between executions and are out of the control of the experimenter. Because of that, exact reproduction of previous results is almost impossible with this approach.

To overcome this fact, an alternative is to use a simulator that offers the same facilities as the OS, but allows one complete control. This simulator allows agents to execute, performs the scheduling among agents and provides communication facilities. With this approach, results are reproducible, the same experiment generates the same results (providing random elements are initialized with the same seed).

The first simulator of this kind appears in the seminal work of Yokoo [6, 7]. Each agent keeps its own clock, which is incremented at each cycle of computation. One cycle for an agent consists of reading all its incoming messages, processing them and writing all messages generated as answers. It is assumed that a message sent at time t is available to the receiver at time $t + 1$. This means a kind of synchronicity in the activation of agents, which is somehow contradictory with the evaluation of asynchronous procedures. We come back on this point in Section 5.

Another scheduling policy is to activate agents randomly: a random number between 1 and n determines the identifier of the agent to activate. When this agent terminates, the same process selects the next agent to activate. This approach seems to be more adequate to evaluate asynchronous procedures. Other scheduling policies could offer some interesting alternatives.

5 Discussion

In this Section we contrast some of the criteria presented above with current practices in the evaluation of DisCSP algorithms. With this exercise we identify some aspects which could be improved in distributed algorithm evaluation.

5.1 Evaluation Parameters

Time and communication. Often we see DisCSP algorithms which are evaluated considering time or communication, but not both aspects. In general, we think that this approach provides incomplete information and does not allow one to assess globally the amount of resources needed for an algorithm. Following standard practice in distributed algorithms, we propose to use these two measures when evaluating DisCSP algorithms. Some researchers have suggested to aggregate both measures in one (or translate one measure into another). When possible, this approach is attractive because it allows us to deal with a single number. However, in many cases it cannot be done without making arbitrary assumptions, difficult to justify. In such cases, we suggest to keep both measures separated.

Timing model. Evaluating an algorithm should follow methods which are adequate for the timing model assumed by the algorithm. Synchronous algorithms can be evaluated using the number of rounds as time complexity measure. However, asynchronous algorithms should not be evaluated using methods that assume a synchronous model (such as the number of rounds).

An interesting question is the evaluation of partially synchronous algorithms, especially on those parts which require waiting for some event caused by other agents. During a waiting episode, an agent may not use its own resources but it is causing some delay to agents which require its input. Waiting episodes can be counted at agent level. In addition, following a similar approach to concurrent constraint checks, we can assess the longest sequence of waiting episodes which cannot be concurrent.

Communication model. Most of DisCSP papers does not deal explicitely with the communication model. It is usually assumed that when an agent sends a message to p other agents, this causes p physical messages in the network. In other words, the unicast model is implicitely assumed. This is fine, the only concern here is that the communication model should be made explicit, so algorithms could be evaluated using different models. This will bring closer the DisCSP paradigm to real communication networks, which finally could promote the use of DisCSP algorithms for practical applications.

Message size. When messages of different sizes are present in DisCSP algorithms, usually size differences are neglected and the number of messages is the only evaluation parameter considered. We believe that this is not a fair approach and the message size cannot be ignored, especially when message sizes differ in more than a constant (for instance, in a function that depends on problem dimensions). We suggest to take message size differences into account, as suggested in the area of distributed algorithms [4].

5.2 Processing Messages: One by One vs Packets

Asynchronous DisCSP algorithms are often described assuming that agents react inmediately after receiving a message: they process messages one by one. However, some algorithms are evaluated processing messages by packets: an agent reads all messages that are waiting in the input buffer and processes them as a whole. It is worth noting that these two strategies may produce quite different results considering the evaluation parameters described above.

The motivation of asynchronous algorithms for processing messages by packets, instead of one by one, is to prevent useless work. A simple example occurs when two consecutive messages arrive from the same agent, informing that it has taken two different values. Obviously, the first message becomes obsolete as soon as the second arrives. All the work generated by processing the first message and extra messages that this processing might be caused, could be saved if the agent would have known the second message. Somehow, this idea was mentioned in [7] and [8]. Recently, in [1] a formal protocol for processing messages by packets is proposed.

Informally, when any agent processes messages by packets, it first reads all messages that are in its input buffer. Then, it processes all read messages as a whole, ignoring those messages that become obsolete by the presence of another message. The agent looks for any consistent value after its agent view and its nogood store are updated with these incoming messages.

Thus, every outgoing message that an agent will send is consequence of the previous incoming messages because all of them update the agent view before agent checks consistency. Therefore, before agent looks for a consistent value, the agent's concurrent counter has to be updated to the maximum value between its own counter before starting to process the packet and the maximum of all concurrent counter of all messages contained in the processed packet.

Empirically, we have tested both types of message processing on distributed binary random problems using two algorithms: one asynchronous and one partial synchronous. The former is the well-known *ABT* algorithm [6, 7]. The latter is *ABT-Hyb*[1], an novel *ABT*-like algorithm which introduces some synchronization points to avoid sending redundant messages. It can be seen as a partially synchronous algorithm.

In our experiments, we have 16 variables/agents ($n = 16$) and 8 values per variable ($m = 8$). The connectivity of the network is set to 0.5 (p_1=0.5). On Table 1 and Table2 we report results averaged over 100 executions in terms of the following parameters:

- the sum of all constraint checks performed by all agents (cc)
- the number of concurrent constraint checks (ccc)
- the total number of messages exchanged ($mess$)
- the number of concurrent messages, computed in the same way as ccc ($cmess$)
- the total number of *Info* messages exchanged ($info$)
- the total number of *Back* messages exchanged ($back$)
- the total number of *Add-Link* messages exchanged ($link$)
- the number of *Back* messages that are obsolete when are received ($obso$)

Regarding the communication cost, the number of messages exchanged in both algorithms processing messages by packets is lower than processing messages one by

messages processing	cc	ccc	mess	cmess	info	back	link	obso
one by one	92,860	23,148	33,184	3,635	25,413	7,733	38	4,824
by packets	77,550	35,408	31,986	5,558	24,877	7,770	39	2,339

Table 1. Results in the pick of difficulty for *ABT* with both types of messages processing

messages processing	cc	ccc	mess	cmess	info	back	link	obso
one by one	57,364	22,720	24,107	4,250	19,720	4,437	37	1,567
by packets	56,680	22,603	23,963	4,229	19,660	4,303	67	1,525

Table 2. Results in the pick of difficulty for *ABT-Hyb* with both types of messages processing

one. Considering the number of concurrent constraint checks, processing messages by packets increases the number of concurrent constraint checks with respect to processing messages one by one. However, the number of obsolete messages decreases when agents process messages by packets. This phenomenon can be seen better if we compute the following ratios:

– ratio of concurrency of constraint checks,

$$rccc = 1 - \frac{ccc}{cc} \tag{1}$$

– ratio of concurrency of messages,

$$rcm = 1 - \frac{cmess}{mess} \tag{2}$$

– ratio of information quality of $Back$ messages,

$$riq = 1 - \frac{obso}{back} \tag{3}$$

The ratio $rccc$ can give us an idea of how concurrent is our algorithm. On contrast, ratio $rccc$ and ratio riq can help us to measure the use of the resources of the network. These parameters are easily extended to synchronous algorithms. In them, $rccc = 0$, $rcmess = 0$ and $riq = 1$ [2].

On Table 3 and Table 4 we show the results of computing these ratios to the experimental results reported on Table 1 and Table 2. Regarding *ABT*, we can see that it becomes less concurrent when messages are processed by packets, although the quality of the information is higher. Regarding *ABT-Hyb* when messages are processing by packets, the concurrency of the algorithm and the quality of the information remains approximately the same as processing messages one by one. This happens because an *ABT-Hyb* agent can be in a *waiting state* without sending any outgoing message. In that state, the agent receives all $Info$ messages updating its agent view accordingly.

[2] Except for special arrangements of the constraint graph, as described in [2]

messages processing	rccc	rcmess	riq
one by one	0.7507	0.8905	0.3757
by packets	0.5434	0.8262	0.6692

Table 3. Ratios for *ABT* algorithm with both types of messages processing.

messages processing	rccc	rcmess	riq
one by one	0.6039	0.8237	0.6395
by packets	0.6013	0.8235	0.6456

Table 4. Ratios for *ABT-Hyb* algorithm with both types of messages processing.

Then, when an agent leaves the *waiting state* it will have a better idea of the current assignments of the other agents.

Finally, it is worth noting that although concurrency decreases when processing messages by packets, this does not necessarity means that the process is less efficient. In fact, it saves some useless work. This is reflected in the increment of riq (ratio of information quality) of the *Back* messages and in the decrement of $rccc$ (ratio of concurrent constraint checks) and $rcmsg$ (concurrent messages).

6 Summary

We believe that the evaluation of current DisCSP algorithms is not completely established, and a common methodology is badly needed. Such methodology should follow standard evaluation methods in distributed algorithms. We have reviewed some basic elements of this area, such as the timing model, the communication model, time and communication complexities. We have also considered evaluation procedures suggested from the distributed constraint community. We have tried to apply them to the evaluation of DisCSP algorithms. Doing this exercise, we have identified some points which should be followed in the evaluation of DisCSP algorithms. These results can be seen as preliminary. More work is needed to achieve a global and coherent methodology for the evaluation of DisCSP algorithms.

References

1. Brito I., Meseguer P. Synchronous, asynchronnous and hybrids algorithms for DisCSP. Submitted to *CP-04 Workshop on Distributed Constraint Reasoning*.
2. Collin Z., Dechter R., Shmuel K. On the Feasibility of Distributed Constraint Satisfaction. *In Proc. of the 12th International Joint Conference on Artificial Intelligence, IJCAI-91*, 318–324, 1991.
3. Meisels A., Kaplansky E., Razgon I., Zivan R. Comparing Performance of Distributed Constraint Processing Algorithms. *AAMAS-02 Workshop on Distributed Constraint Reasoning*, 86–93, Bologna, Italy, 2002.
4. Lynch N. *Distributed Algorithms*, Morgan–Kaufmann, 1996.

5. Lamport L. Time, Clock, and the Ordering of Evens in a Distributed System. *Communications of the ACM*, **21(7)**, 558–565, 1978.
6. Yokoo M., Durfee E.H., Ishida T., Kuwabara K. Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. *In Proc. of the 12th International Conference on Distributed Computing System*, 614–621, 1992.
7. Yokoo M., Durfee E.H., Ishida T., Kuwabara K. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Trans. Knowledge and Data Engineering* **10**, 673–685, 1998.
8. Zivan, R. and Meisels, A. *Synchronous and Asynchronous Search on DisCSPs*. In Proc. of EUMAS-2003, Oxford, UK, 2003

Message delay and DisCSP search algorithms

Roie Zivan and Amnon Meisels
{zivanr,am}@cs.bgu.ac.il

Department of Computer Science,
Ben-Gurion University of the Negev,
Beer-Sheva, 84-105, Israel

Abstract. Due to the distributed nature of the problem, message delay can have unexpected effects on the behavior of distributed search algorithms on *Distributed constraint satisfaction problems* ($DisCSPs$). This has been recently shown in an experimental study of two asynchronous DisCSP algorithms [Fernandez et. al.2002]. To evaluate the impact of message delay on the run of DisCSP search algorithms, an *Asynchronous Message Delay Simulator* ($AMDS$) for $DisCSP$s which includes the cost of message delays is introduced. The number of steps of computation calculated by the $AMDS$ (or number of concurrent constraints checks) can serve as good performance measures, when messages are delayed.
The performance of three representative algorithms is measured on randomly generated instances of DisCSPs with several types of delays for messages.
Two measures of performance are used - concurrent computation time and network load. The performance of asynchronous backtracking deteriorates on systems with random message delays, for both measures. For synchronous algorithms, with delayed messages, time performance becomes worse then asynchronous backtracking, but the network load is not affected. Concurrent search algorithms, are affected very lightly by message delay with respect to both measures.

Acknowledgment: Supported by the Lynn and William Frankel center for Computer Sciences.

Key words: Distributed Constraint Satisfaction, Search, Distributed AI.

1 Introduction

Distributed constraints satisfaction problems (*DisCSP*s) are composed of agents, each holding its local constraints network, that are connected by constraints among variables of different agents. Agents assign values to their variables, attempting to generate a locally consistent assignment that is also consistent with all constraints between agents (cf. [Yokoo2000,Solotorevsky et. al.1996]). To achieve this goal, agents check the value assignments to their variables for local consistency and exchange messages among them, to check consistency of their proposed assignments against constraints with variables that belong to different agents [Yokoo2000,Bessiere et. al.2001].

Search algorithms on DisCSPs are run concurrently by all agents and their performance must be measured in terms of distributed computation. Two measures are commonly used to evaluate distributed algorithms - time, which is measured in terms of computational effort and network load [Lynch1997]. The time performance of search algorithms on DisCSPs has traditionally been measured by the number of computation cycles or steps (cf. [Yokoo2000]). In order to take into account the effort an agent makes during its local assignment the computational effort can be measured by the number of concurrent constraints checks that agents perform ([Meisels et. al.2002,Silaghi2002]). Measuring the network load poses a much simpler problem. Network load is generally measured by counting the total number of messages sent during search [Lynch1997].

When instantaneous message arrival is assumed, steps of computation in a standard simulator can serve to measure the concurrent run-time of a DisCSP algorithm [Yokoo2000]. For an optimal communication network, in which messages arrive with no delay, one can also use the number of concurrent constraints checks (CCCs), for an implementation independent measure of concurrent run time [Meisels et. al.2002]. On realistic networks, in which there are variant message delays, the time of run cannot be measured simply by the steps of computation. Take for example Synchronous Backtracking [Yokoo2000]. Since all agents are completely synchronized and no two agents compute concurrently, the number of computational steps is not affected by message delays. However, the effect on the run time of the algorithm is completely cumulative (delaying each and every step) and is thus large (see section 6 for details).

In order to evaluate the impact of message delays on DisCSP search algorithms, we present an *Asynchronous Message Delay Simulator* ($AMDS$) which measures the logical time of the algorithm run in steps of computation or concurrent constraints checks, and simulates message delays accordingly. The $AMDS$ is described in detail in section 3. It can simulate systems with different types of message delays from fixed message delays, through random message delays, to systems in which the length of the delay of each message is dependent on the current load of the network. Since the delay is measured in concurrent computation steps (or concurrent constraints checks), the final logical time that is reported as the cost of the algorithm run, includes steps of computation which were actually performed by some agent, and computational steps which were added as message delay simulation while no computation step was performed concurrently (see section 3).

To demonstrate the behavior of DisCSP search algorithms in the presence of message delay, three algorithms are compared. Although the three chosen algorithms are similar in their run-time results on systems with no message delay they are very different from one another. The first, *Conflict based Back Jumping* (CBJ) [Zivan and Meisels2003] is a synchronous algorithm which performs pruning of its search space according to *Dynamic Backtracking* (DB) methods [Ginsberg1993,Zivan and Meisels2003]. The second is the *Asynchronous Backtracking* (ABT) algorithm in which agents perform assignments concurrently and asynchronously [Yokoo2000,Bessiere et. al.2001]. The third, *Concurrent Backtracking* [Zivan and Meisels2004] is a concurrent algorithm in which a dynamic number of independent search processes explore concurrently and asynchronously, non intersecting parts of the $DisCSP$ search space. The results presented in section 6 show the different impact of message delays on these three algorithms.

Distributed constraints satisfaction problems ($DisCSPs$) are presented briefly in section 2. A detailed introduction of the algorithm and method for simulating message delays in $DisCSP$ search, and of the method of evaluating the run time of an algorithm, is presented in section 3. A proof of the validity of the simulating algorithm is presented in section 4. A description of the compared algorithms - synchronous Conflict based Backjumping (CBJ), Asynchronous Backtracking (ABT), and Concurrent Backtracking ($ConcBT$), is presented in section 5. Section 6 presents extensive experimental results, comparing all three algorithms on randomly generated $DisCSP$s with different types of message delays. A discussion of the new insights of the performance and on the advantages of these three algorithms, on different $DisCSP$ instances and communication networks, is presented in section 7.

2 Distributed Constraint Satisfaction

A distributed constraints network (or a distributed constraints satisfaction problem - *DisCSP*) is composed of a set of k agents $A_1, A_2, ..., A_k$. Each agent A_i contains a set of constrained variables $X_{i_1}, X_{i_2}, ..., X_{i_{n_i}}$. Constraints or **relations** R are subsets of the Cartesian product of the domains of the constrained variables. For a set of constrained variables $X_{i_k}, X_{j_l}, ..., X_{m_n}$, with domains of values for each variable $D_{i_k}, D_{j_l}, ..., D_{m_n}$, the constraint is defined as $R \subseteq D_{i_k} \times D_{j_l} \times ... \times D_{m_n}$. A **binary constraint** R_{ij} between any two variables X_j and X_i is a subset of the Cartesian product of their domains; $R_{ij} \subseteq D_j \times D_i$. In a distributed constraint satisfaction problem *DisCSP*, the agents are connected by constraints between variables that belong to different agents (cf. [Yokoo et. al.1998,Solotorevsky et. al.1996]). In addition, each agent has a set of constrained variables, i.e. a *local constraint network*.

An assignment (or a label) is a pair $< var, val >$, where var is a variable of some agent and val is a value from var's domain that is assigned to it. A *partial assignment* (or a compound label) is a set of assignments of values to a set of variables. A **solution** to a *DisCSP* is a partial assignment that includes all variables of all agents, that satisfies all the constraints. Following all former work on *DisCSP*s, agents check assignments of values against non-local constraints by communicating with other agents through sending and receiving messages. An agent can send messages to any one of the other agents.

The delay in delivering a message is assumed to be finite [Yokoo2000]. One simple protocol for checking constraints, that appears in many distributed search algorithms, is to send a proposed assignment $< var, val >$, of one agent to another agent. The receiving agent checks the compatibility of the proposed assignment with its own assignments and with the domains of its variables and returns a message that either acknowledges or rejects the proposed assignment. The following assumptions are routinely made in studies of Distributed CSPs and are assumed to hold in the present study [Yokoo2000,Bessiere et. al.2001].

1. All agents hold exactly one variable.
2. The amount of time that passes between the sending of a message to its reception is finite.

3. Messages sent by agent A_i to agent A_j are received by A_j in the order they were sent.
4. Every agent can access the constraints in which it is involved and check consistency against assignments of other agents.

3 Simulating search on DisCSPs

The standard model of Distributed Constraints Satisfaction Problems has agents that are autonomous asynchronous entities. The actions of agents are triggered by messages that are passed among them. In real world systems, messages do not arrive instantaneously but are delayed due to networks properties. Delays can vary from network to network (or with time, in a single network) due to networks topologies, different hardware and different protocols used. To simulate asynchronous agents, the simulator implements agents as *Java Threads*. Threads (agents) run asynchronously, exchanging messages by using a common mailer. After the algorithm is initiated, agents block on incoming message queues and become active when messages are received.

Concurrent steps of computation, in systems with no message delay, are counted by a method similar to that of [Lamport1978,Meisels et. al.2002]. Every agent holds a counter of computation steps. Every message carries the value of the sending agent's counter. When an agent receives a message it updates its counter to the largest value between its own counter and the counter value carried by the message. By reporting the cost of the search as the largest counter held by some agent at the end of the search, we achieve a measure of concurrent search effort that is similar to Lamport's logical time [Lamport1978].

On systems with message delays, the situation is more complex. For the simplest possible algorithm, Synchronous Backtrack (SBT) [Yokoo2000], the effect of message delay is very clear. The number of computation steps is not affected by message delay and the delay in every step of computation is the delay on the message that triggered it. Therefore, the total time of the algorithm run can be calculated as the total computation time, plus the total delay time of messages. In the presence of concurrent computation, the time of message delays must be added to the total algorithm time *only if no computation was performed concurrently*. To achieve this goal, the algorithm of the *Asynchronous Message-Delay Simulator* ($AMDS$) counts message delays in terms of computation steps and adds them to the accumulated run-time when no computation is performed concurrently.

In order to simulate message delays, all messages are passed by a dedicated $Mailer$ thread. The mailer holds a counter of concurrent computation steps performed by agents in the system. This counter represents the logical time of the system and we refer to it as the *Logical Time Counter* (LTC). Every message delivered by the mailer to an agent, carries the LTC value of its delivery to the receiving agent. To compute the logical time that includes message delays, agents perform a similar computation to the one used when there are no message delays [Meisels et. al.2002]. An agent that receives a message updates its own LTC to the largest value between its own and the LTC on the message received. Then the agent performs the computation step, and sends its outgoing messages with the value of its LTC incremented by 1.

- **upon receiving message** msg:
  1. LTC ← max(LTC, msg.LTC)
  2. delay ← $choose\_delay$
  3. msg.$delivery\_time$ ← LTC + delay
  4. $outgoing\_queue$.add(msg)
  5. $deliver\_messages$
- **when there are no incoming messages and all agents are idle**
  1. LTC ← $outgoing\_queue$.first_msg.LTC
  2. $deliver\_messages$
- **deliver_messages**
  1. **foreach** (message m in outgoing queue)
  2. **if** ($m.delivery\_time \leq$ LTC)
  3. m.LTC ← LTC
  4. deliver(m)

Fig. 1. The Mailer algorithm

The mailer simulates message delays in terms of concurrent computation steps. To do so it uses its own (global) LTC, according to the algorithm presented in figure 1. Let us go over the details of the $Mailer$ algorithm, in order to understand the measurements performed by the $AMDS$ during run time.

When the mailer receives a message, it first checks if the LTC value that is carried by the message is larger than its own value. If so, it increments the value of the LTC (line 1). This generates the value of the global clock (of the Mailer) which is the largest of all logical times of all agents. In line 2 a delay for the message (in number of steps) is selected. Here, different types of selection mechanisms can be used, from fixed delays, through random delays, to delays that depend on the actual load of the communication network. To achieve delays that simulate dependency on network load, for example, one can assign message delays that are proportional to the size of the outgoing message queue. Each message is assigned a $delivery\_time$ which is the sum of the current value of the Mailer's LTC and the selected delay (in steps), and placed in the $outgoing\_queue$ (lines 3,4). The $outgoing\_queue$ is a priority queue in which the messages are sorted by $delivery\_time$, so that the first message is the message with the lowest $delivery\_time$. In order to preserve the third assumption of section 2, messages from agent A_i to agent A_j cannot be placed in the outgoing queue before messages which are already in the outgoing queue, which were sent from A_i to A_j. This property is essential to asynchronous algorithms which are not correct without it (cf. [Bessiere et. al.2001]). The last line of the $Mailer$'s code calls method $deliver\_messages$, which delivers all messages with $delivery\_time$ less or equal to the mailer's current LTC value, to their destination agents.

When there are no incoming messages, and all agents are idle, if the $outgoing\_queue$ is not empty (otherwise the system is idle and a solution has been found) the $Mailer$ increases the value of the LTC to the value of the $delivery\_time$ of the first message in the outgoing queue and calls $deliver\_messages$. This is a crucial step of the simulation algorithm. Consider the run of a synchronous search algorithm. For *Synchronous*

Backtracking (SBT) [Yokoo2000], every delay needs the mechanism of updating the Mailer's LTC (line 1 of the second function of the code in figure 1). This is because only one agent is computing at any given instance, in synchronous backtrack search.

The concurrent run time reported by the algorithm, is the largest LTC held by some agent at the end of the algorithm run. By incrementing the LTC only when messages carry LTCs with values larger than the mailer's LTC value, steps that were performed concurrently are not counted twice. This is an extension of Lamport's logical clocks algorithm [Lamport1978], as proposed for DisCSPs by [Meisels et. al.2002], and extended here for message delays.

A similar description holds for evaluating the algorithm run in logical concurrent constraints checks. In this case the agents would extend the value of their LTCs in each step, not by one, but by the number of constraints checks they actually performed. This enables a concurrent performance measure that incorporates the computational cost of the local step, which might be different in different algorithms. Furthermore, it also enables to evaluate algorithms in which agents perform computation which is not triggered or followed by a message.

4 Correctness of the $AMDS$

In order to prove the validity of the proposed measure simulation, its correspondence to runs of a *Synchronous Cycles Simulator* is presented. In a *Synchronous Cycle Simulator* [Yokoo2000], in every cycle each agent can read all messages that were sent to it in the previous cycle and perform a single computation step which can be followed by the sending of messages (which will be received in the next cycle). Agents can be idle in some cycles, if they do not receive a message which triggers a computation step. The cost of the algorithm run, is the number of synchronous cycles performed until a solution is found or a non solution is declared (see [Yokoo2000]). Message delay can be simulated in such a synchronous simulator by delivering messages to agents some cycles after they were sent.

Theorem 1. *Any run of $AMDS$ can be simulated by a* Synchronous Cycle Simulator *(SCS), in which cycle c_i of the SCS corresponds to an LTC value of $AMDS$.*

The proof of the theorem is immediate. Every message m sent by an agent A_i to agent A_j can be assigned a value d which is the largest value between the LTC carried by m in the $AMDS$ run and the value of the LTC held by A_j when it received m. Running a *Synchronous Cycle Simulator* (SCS) and assigning each message m with the value d calculated as described above, the message can be delivered to A_j in cycle d. The outcome of the special SCS is that every agent in every cycle c_i will have the same knowledge about the other agents as the agents performing the matching steps in the $AMDS$ run. Assuming the algorithm is deterministic, the agent will perform the same computation and send the same messages. If the algorithm includes random choices the run can be simulated by recording $AMDS$ choices and forcing the same choice in the synchronous simulator run. To complete the proof of the theorem one needs to show the following Lemma.

Lemma 1. *At any cycle c_i of the synchronous simulator, the LTC values of all agents performing the matching steps in the $AMDS$ are equal to i.*

Proof: We prove Lemma 1 by induction. After performing step number one, all agents in $AMDS$ advance their LTC to one. Assuming the Lemma holds for $N-1$ cycles, all agents that are about to perform the Nth step, hold counters with values less or equal to $N-1$. The messages they will receive will carry the $delivery\_time$ LTC which is $N-1$. Since the agent's LTCs are updated to the largest between the received LTC and their own, after receiving the message and performing the next step of computation, their LTC value will be equal to N. □

The theorem demonstrates that for computing steps of computation, the asynchronous simulator is equivalent to a standard SCS *that does not wait for all agents* to complete their computation in a given cycle, in order to move to the next cycle.

The most important advantage of the asynchronous simulator can now be described. When computational effort is computed, in terms of constraints checks for example, the SCS becomes useless. This is because at each cycle agents perform different amounts of computation, depending on the algorithm, on arrival of messages, etc. The simulator does not "know" the amount of computation performed by each agent and, therefore, cannot move the resulting message in the correct cycle (one that matches the correct amount of computation and waiting). The natural way to compute concurrent CCs is by using an asynchronous simulator, the $AMDS$ as proposed in section 3

5 The tested algorithms

In order to check the behavior of distributed search algorithms under message delays, the $AMDS$ is used to compare the run of three algorithms for solving $DisCSP$s. These algorithms represent three different families of algorithms:

- Synchronous algorithms represented by synchronous Conflict based Backjumping (CBJ) [Zivan and Meisels2003].
- Asynchronous Backtracking algorithms represented by ABT [Bessiere et. al.2001].
- Concurrent search algorithms represented by Concurrent Backtracking ($ConcBT$) [Zivan and Meisels2004].

In the following subsections the three representative algorithms are described. The performance of the algorithms is evaluated in section 6 and the impact of delayed messages on their performance is described. The relation of the impact of delayed messages on each of the algorithms and the properties of the algorithm's family, is discussed in section 7.

5.1 Conflict based Backjumping

The Synchronous Backtrack algorithm (SBT) [Yokoo2000], is a distributed version of chronological backtrack [Prosser1993]. SBT has a total order among all agents. Agents

exchange a partial solution that we term *Current Partial Assignment* (CPA) which carries a consistent tuple of the assignments of the agents it passed so far. The first agent initializes the search by creating a CPA and assigning its variables on it. Every agent that receives the CPA tries to assign its variable without violating constraints with the assignments on the CPA. If the agent succeeds to find such an assignment to its variable, it appends the assignment to the tuple on the CPA and sends it to the next agent. Agents that cannot extend the consistent assignment on the CPA, send the CPA back to the previous agent to change its assignment, thus perform a chronological backtrack. An agent that receives a CPA in a backtrack message removes the assignment of its variable and tries to reassign it with a consistent value. The algorithm ends successfully if the last agent manages to find a consistent assignment for its variable. The algorithm ends unsuccessfully if the first agent encounters an empty domain.

The version of Conflict based Backjumping (CBJ) [Prosser1993] improves on simple synchronous backtrack (SBT) by using a method based on dynamic backtracking [Ginsberg1993,Bessiere et. al.2001]. In the distributed CBJ, when an agent removes a value from its variable's domain, it stores the eliminating explanation ($Nogood$), i.e. the subset of the CPA that caused the removal. As in the corresponding version of asynchronous backtrack [Bessiere et. al.2001], when a backtrack operation is performed the agent resolves its $Nogoods$ creating a conflict set which is used to determine the culprit agent to which the backtrack message will be sent. The resulting synchronous algorithm has the backjumping property (i.e. CBJ) [Ginsberg1993]. When the CPA is received again, values whose eliminating $Nogoods$ are no longer consistent with the partial assignment on the CPA are returned to the agents' domain.

5.2 Asynchronous Backtracking

The $Asynchronous\ Backtrack\ algorithm$ (ABT) was presented in several versions over the last decade and is described here in accordance with the more recent papers [Yokoo2000,Bessiere et. al.2001]. In the ABT algorithm, agents hold an assignment for their variables at all times, which is consistent with their view of the state of the system (i.e. their $Agent\_view$). When the agent cannot find an assignment consistent with its $Agent\_view$, it changes its view by eliminating a conflicting assignment from its $Agent\_view$ data structure and sends back a $Nogood$.

The Asynchronous Backtrack algorithm ABT [Yokoo2000], has a total order of priorities among agents. Agents hold a data structure called $Agent\_view$ which contains the most recent assignments received from agents with higher priority. The algorithm starts by each agent assigning its variable, and sending the assignment to neighboring agents with lower priority. When an agent receives a message containing an assignment (an $ok?$ message [Yokoo2000]), it updates its $Agent\_view$ with the received assignment and if needed replaces its own assignment, to achieve consistency. Agents that reassign their variable, inform their lower priority neighbors by sending them $ok?$ messages. Agents that cannot find a consistent assignment, send the inconsistent tuple in their $Agent\_view$ in a backtrack message (a $Nogood$ message [Yokoo2000]). The $Nogood$ is sent to the lowest priority agent in the inconsistent tuple, and its assignment is removed from their $Agent\_view$. Every agent that sends a $Nogood$ message, makes

another attempt to assign its variable with an assignment consistent with its updated $Agent\_view$.

Agents that receive a $Nogood$, check its relevance against the content of their $Agent\_view$. If the $Nogood$ is relevant, the agent stores it and tries to find a consistent assignment. In any case, if the agent receiving the $Nogood$ keeps its assignment, it informs the $Nogood$ sender by re-sending it an $ok?$ message with its assignment [Bessiere et. al.2001]. An agent A_i which receives a $Nogood$ containing an assignment of agent A_j which is not included in its $Agent\_view$, adds the assignment of A_j to it's $Agent\_view$ and sends a message to A_j asking it to add a link between them. In other words, A_j is requested to inform A_i about all assignment changes it performs in the future [Yokoo2000].

The performance of ABT can be strongly improved by requiring agents to read all messages they receive before performing computation [Yokoo2000]. A formal protocol for such an algorithm was not published. The idea is not to reassign the variable until all the messages in the agent's 'mailbox' are read and the $Agent\_view$ is updated. This technique was found to improve the performance of ABT on the harder instances of randomly generated DisCSPs by a factor of 4 [Zivan and Meisels2003]. However, this property makes the efficiency of ABT dependent on the contents of the agent's mailbox in each step, i.e. on message delays (see section 6). The consistency of the $Agent\_view$ held by an agent, with the actual state of the system before it begins the assignment attempt is affected directly by the number and relevance of the messages it received up to this step.

Another improvement to the performance of ABT can be achieved by using the method for resolving inconsistent subsets of the $Agent\_view$, based on methods of dynamic backtracking [Ginsberg1993]. A version of ABT that uses this method was presented in [Bessiere et. al.2001]. In [Zivan and Meisels2003] the improvement of ABT using this method over ABT sending its full $Agent\_view$ as a $Nogood$ was found to be minor. In all the experiments in this paper a version of ABT which includes both of the above improvements is used. Agents read all incoming messages that were received before performing computation and $Nogoods$ are resolved, using the dynamic backtracking method.

5.3 Concurrent Backtracking

The $ConcBT$ algorithm [Zivan and Meisels2004] performs multiple concurrent backtrack searches on disjoint parts of the *DisCSP* search-space. Each agent holds the data relevant to its state on each sub-search-space in a separate data structure which is termed *Search Process (SP)*. Agents in the $ConcBT$ algorithm pass their assignments to other agents on a *CPA* (Current Partial Assignment) data structure. Each *CPA* represents one search process, and holds the agents current assignments in the corresponding search process. An agent that receives a *CPA* tries to assign its local variable with values that are not conflicting with the assignments on the *CPA*, using the current domain in the *SP* related to the received *CPA*. The uniqueness of the *CPA* for every search space ensures that assignments are not done concurrently in a single sub-search-space.

Exhaustive search processes which scan heavily backtracked search-spaces, can be split dynamically. Each agent can generate a set of CPAs that split the search space of a

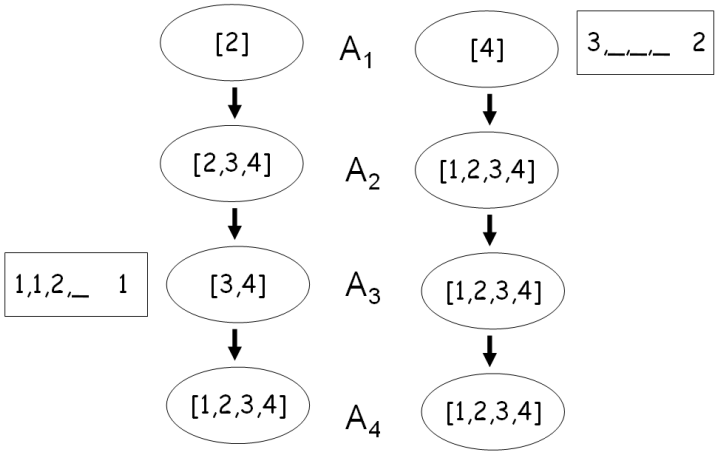


Fig. 2. ConcBT with two CPAs

CPA that passed through that agent, by splitting the domain of its variable. Agents can perform splits independently and keep the resulting data structures (*SP*s) privately. All other agents need not be aware of the split, they process all $CPAs$ in exactly the same manner (see [Zivan and Meisels2004] for a detailed explanation). *CPA*s are created either by the Initializing Agent (*IA*) at the beginning of the algorithm run, or dynamically by any agent that splits an active search-space during the algorithm run. A heuristic of counting the number of times agents pass the *CPA* in a sub-search-space (without finding a solution), is used to determine the need for re-splitting of that sub-search-space. This generates a mechanism of load balancing, creating more search processes on heavily backtracked search spaces.

A backtrack operation is performed by an agent which fails to find a consistent assignment in the search-space corresponding to the partial assignment on the *CPA*. Agents that have performed dynamic splitting, have to collect all of the returning CPAs, of the relevant SP, before performing a backtrack operation.

Figure 2 presents an example of a DisCSP, searched concurrently by two synchronous processes represented by two CPAs, CPA_1 and CPA_2. Each of the four agents A_1 to A_4, holds two SPs. Only the current domains of the SPs are shown in Figure 2. The domains on the left represent the state after 3 assignments to CPA_1. The domains on the right of figure 2 represent the state after the first assignment to CPA_2.

Agent A_1 has assigned the value 1 on CPA_1 and the value 3 on CPA_2. The values that are left in each of its domains are 2 in SP_1 and 4 in SP_2. The two CPAs are traversing non intersecting sub search spaces in which CPA_1 is exploring all tuples beginning with 1 or 2 for agent A_1, and CPA_2 all tuples beginning with 3 or 4. CPA_1 is depicted on the LHS of figure 2 and CPA_2 is on the top RHS. Each CPA has its ID on its right.

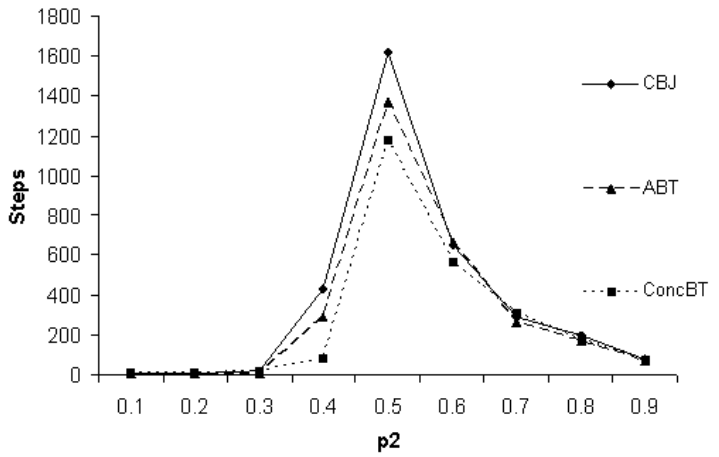


Fig. 3. Concurrent steps of computation with no message delays

6 Experimental evaluation

The network of constraints, in each of the experiments, is generated randomly by selecting the probability p_1 of a constraint among any pair of variables and the probability p_2, for the occurrence of a violation among two assignments of values to a constrained pair of variables. Such uniform random constraints networks of n variables, k values in each domain, a constraints density of p_1 and tightness p_2, are commonly used in experimental evaluations of CSP algorithms (cf. [Prosser1996]). Experiments were conducted on networks with 10 variables ($n = 10$) and 10 values ($k = 10$). All instances were created with density parameter $p_1 = 0.7$. The value of p_2 was varied between 0.1 to 0.9. This creates problems that cover a wide range of difficulty, from easy problem instances to instances that take several CPU minutes to solve.

In order to evaluate the algorithms, two measures of search effort are used. One counts the number of concurrent steps of computation [Lynch1997,Yokoo2000], to measure computational cost. The other measures communication load in the form of the total number of messages sent [Lynch1997]. Concurrent steps of computation are counted by a method similar to that of [Lamport1978,Meisels et. al.2002]. In order to evaluate the logical time (including message delays) of the algorithm, in steps of computation, we use the simulator as described in section 3.

In the first set of experiments the three algorithms are compared without any message delay. The results presented in figure 3 show that the numbers of steps of computation that the three algorithms perform are very similar, on systems with no message delays. ABT performs slightly less steps than CBJ and $ConcBT$ performs slightly better than ABT. However, when it comes to network load, the results in figure 4 show that for the harder problem instances, agents in ABT send *six times more messages* than sent by agents in CBJ and more than twice the number of messages sent by agents in $ConcBT$.

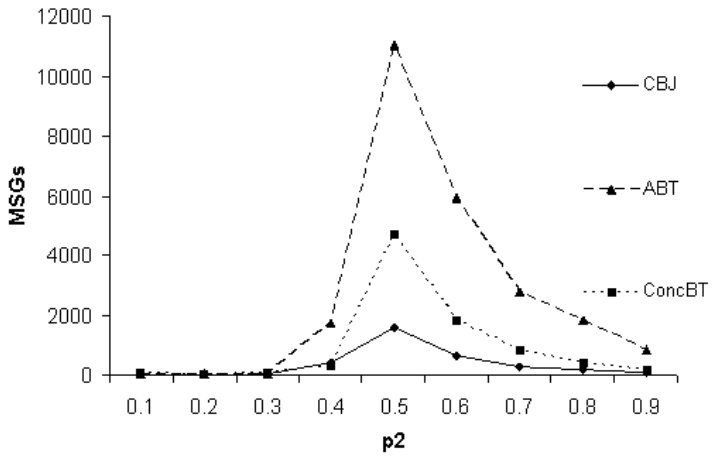


Fig. 4. Total number of messages with no message delays

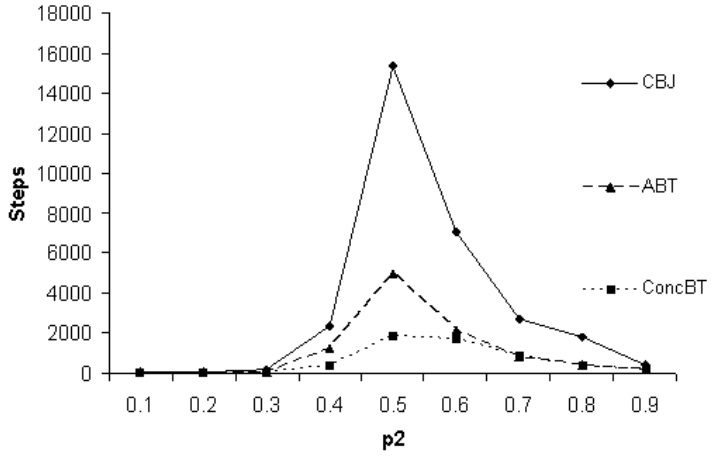


Fig. 5. Logical number of concurrent steps with random message delays

In the second set of experiments, the simulator's $Mailer$ delayed messages randomly for 5-10 steps (as described in section 3).

Figure 5 presents the results of logical time, counted in concurrent steps, for random message delays. It is clear in figure 5 that even though message delays do not affect the number of concurrent steps performed by agents in CBJ, when message delay is correctly counted, CBJ is affected the most. The number of steps performed by ABT in the presence of delays, grows by a large factor. This is expected, since agents are more likely to respond to a single message, instead of all the messages sent in the former (ideal) cycle of computation. Messages in asynchronous backtracking are many times conflicting. As a result, agents perform more unnecessary computation steps when re-

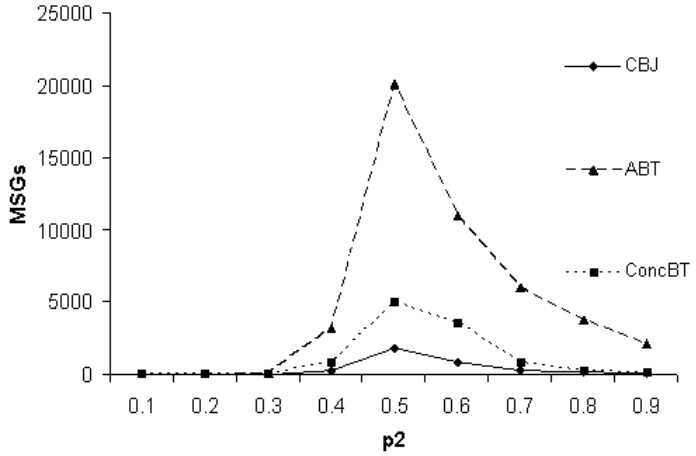


Fig. 6. Number of messages with random message delays

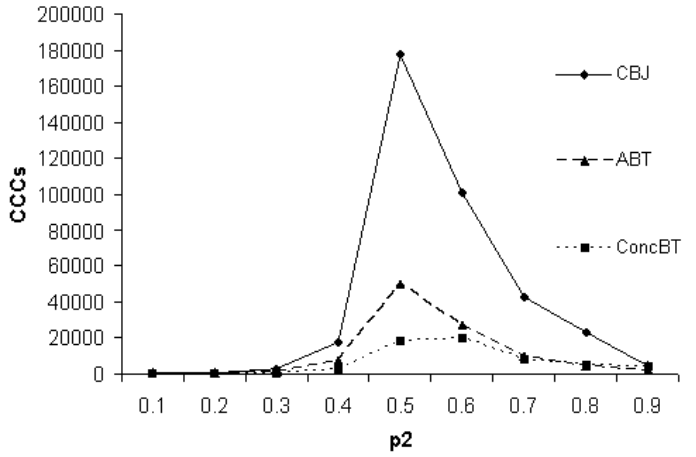


Fig. 7. Logical number of concurrent constraints checks with random message delays

sponding to fewer messages in each cycle.This can explain a similar result for ABT, on a different set of problems [Fernandez et. al.2002]. The logical time performance of concurrent search process algorithms, is not strongly affected by message delay. For the harder problems $ConcBT$ performs less than half the steps of computation of ABT (see figure 5). Network load, for the same (delayed messages) experiments is presented in Figure 6. Both CBJ and $ConcBT$ send the same number of messages as in the case of no message delays. The number of messages sent by asynchronous backtracking increases dramatically. ABT sends almost twice as much messages in the presence of random message delays, than it sends in the case of no message delays (figure 6). Figure 7

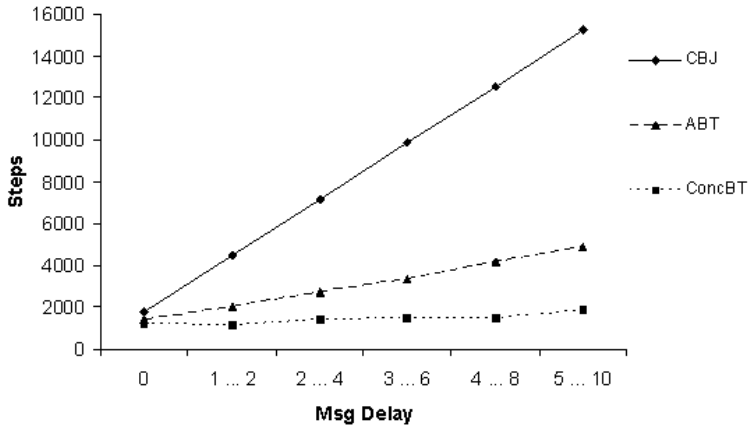


Fig. 8. Logical number of steps with different random message delays

presents the results for logical time that is counted in units of concurrent constraints checks. In this experiment the local computation is taken in to account. The delay for every message is chosen a random value between 50 to 150 constraints checks.

The last set of experiments tests the dependence of algorithm performance on the *amount of delay of messages*. All algorithms are run on the hardest problem instances (p_2 = 0.5) with an increasing amount of message delay. The different impact of random delays on the different algorithms is presented in figure 8. The number of steps of synchronous and of asynchronous backtracking grows with the size of message delay. In contrast, larger delays do not have an impact on the number of steps of concurrent search (Figure 8).

7 Discussion

A study of the impact of message delays on the behavior of $DisCSP$ search algorithms has been presented. Use was made of an asynchronous simulator that runs the $DisCSP$ algorithms with different types of message delays and measures performance in concurrent steps of computation. The logical number of steps/constraints-checks takes into account the impact of message delays on the actual runtime of $DisCSP$ algorithms. Three different algorithms for solving $DisCSPs$ were investigated.

In asynchronous backtracking, agents perform assignments asynchronously. As a result of message delay, some of their computation can be irrelevant (due to inconsistent $Agent\_views$ while the updating message is delayed). This can explain the large impact of message delays on the computation performed by ABT (cf. [Fernandez et. al.2002]). The results presented in section 6 strengthen the results reported by Fernandez et. al. [Fernandez et. al.2002], and do so for a larger family of random problems.

The impact of message delays on concurrent search algorithms is minor. This is very apparent in Figure 8, where the number of steps of computation is independent of the size of message delay for $ConcBT$.

To understand the robustness of $ConcBT$ to message delay imagine the following example. Consider the case where $ConcBT$ splits the search space multiple times and the number of CPAs is larger than the number of agents. In systems with no message delays this would mean that some of the CPAs are waiting in incoming queues, to be processed by the agents. This delays the search on the sub-search-spaces they represent. In systems with message delays, these queues are shortened due to later arrivals of $CPAs$. The net result is that agents are kept busy at all times, performing computation against consistent partial assignments. The results in section 6 demonstrate that the above possible description can be achieved.

In terms of network load, the results of the experimental investigation show that asynchronous backtrack puts a heavy load on the network, which doubles in the case of message delays. The number of messages sent, in both synchronous and concurrent algorithms, is much smaller than the load of asynchronous backtracking and is not affected by message delays.

References

[Bessiere et. al.2001] C. Bessiere, A. Maestre and P. Messeguer. Distributed Dynamic Backtracking. *Proc. Workshop on Distributed Constraints, IJCAI-01*, Seattle, 2001.

[Fernandez et. al.2002] C. Fernandez, R. Bejar, B. Krishnamachari, K. Gomes Communication and Computation in Distributed CSP Algorithms. *Proc. Principles and Practice of Constraint Programming, CP-2002*, pages 664-679, Ithaca NY USA, July, 2002.

[Ginsberg1993] M. L. Ginsberg Dynamic Backtracking. *Artificial Intelligence Research*, vol.1, pp. 25-46, 1993

[Lamport1978] L. Lamport Time, clocks and the ordering of events in a distributed system. *Comm. of ACM*, vol. 21, pp.558-565, 1978.

[Lynch1997] N. A. Lynch. Distributed Algorithms. Morgan Kaufmann Series, 1997.

[Meisels et. al.2002] A. Meisels et. al. Comparing performance of Distributed Constraints Processing Algorithms. *Proc. DCR Workshop, AAMAS-2002* , pp. 86-93, Bologna, July, 2002.

[Prosser1993] P. Prosser. Hybrid Algorithm for the Constraint Satisfaction Problem, *Computational Intelligence*, vol. 9, pp. 268-299, 1993.

[Prosser1996] P. Prosser An empirical study of phase transition in binary constraint satisfaction problems *Artificial Intelegence*, vol. 81, pp. 81-109, 1996.

[Silaghi2002] M.C. Silaghi Asynchronously Solving Problems with Privacy Requirements. *PhD Thesis*,Swiss Federal Institute of Technology (EPFL), 2002.

[Solotorevsky et. al.1996] G. Solotorevsky, E. Gudes and A. Meisels. Modeling and Solving Distributed Constraint Satisfaction Problems (DCSPs). *Constraint Processing-96*, New Hamphshire, October 1996.

[Yokoo et. al.1998] M. Yokoo, E. H. Durfee, T. Ishida, K. Kuwabara. Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Trans. on Data and Kn. Eng.*, vol. 10(5), pp. 673-685, 1998.

[Yokoo2000] M. Yokoo. Algorithms for Distributed Constraint Satisfaction: A Review. *Autons Agents Multi-Agent Sys 2000*, vol. 3(2), pp. 198-212, 2000.

[Zivan and Meisels2003] R. Zivan and A. Meisels. Synchronous vs. Asynchronous search on DisCSPs. *Proc. EUMAS-03 1st European Workshop on Multi-agent Systems*, pp. 202-208, Oxford, December, 2003.

[Zivan and Meisels2004] R. Zivan and A. Meisels. Concurrent Backtrack Search on DisCSPs. *Proc. FLAIRS-04*, Miami Beach, May, 2004. (full version can be loaded from http://www.cs.bgu.ac.il/ zivanr)