Distributed Constraint Reasoning (DCR'07)

Distributed Constraint Reasoning (DCR) problems arise when pieces of information about variables, constraints or both are relevant to independent but communicating agents. They provide a promising framework to deal with the increasingly diverse range of distributed real world problems emerging from the evolution of computation and communication technologies.

Distributed Constraint Satisfaction Problems (DisCSPs) have been studied for almost two decades. Recent years have seen increasing interest in this topic, and algorithms have recently become efficient enough so that practical applications can be considered. The current challenges posed by solving Distributed Constraint Reasoning Problems include dealing with resource restrictions (such as limited time and/or communication), privacy requirements, identifying and exploiting opportunities for cooperation, and designing conflict resolution strategies.

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, 2004), IJCAI (2001, 2003,2005) and AAMAS (2002, 2006). Building upon these previous successful workshops, we continue in 2007, with the Seventh International DCR workshop held in conjunction with IJCAI-2007 in Hyderabad, India.

These proceedings include 9 papers that span several areas within DCR, from theoretical analyses, to new algorithms, to applications, to implementations of DCOP platforms.

I am happy to announce that this year's workshop will also feature two invited talks. Bringing our research into solutions for real world problems should always be a high priority for the community. Thus, I am happy to announce an invited talk by Toby Walsh on a large distributed scheduling application. In an effort to reach out to other communities, we will host a second invited talk, by Gianni Di Caro, who will introduce us to Ant Colony Optimization (ACO) techniques.

Finally, I would like to thank all the authors and especially the organizing committee, for working hard to make this an interesting and fruitful workshop.

Adrian Petcu,
workshop chair

January 2007

Organization

Program Chair

Adrian Petcu
LIA(I&C)
Ecole Polytechnique Federale de Lausanne (EPFL)
Station 14
IN-Ecublens
1015 Lausanne
Switzerland
Tel: +41-21-693-6711
Fax: +41-21-693-5225
adrian.petcu@epfl.ch

Program Committee

Christian Bessiere, LIRMM, France
Boi Faltings, EPFL, Switzerland
Youssef Hamadi, Microsoft Research Cambridge, UK
Katsutoshi Hirayama, Kobe University, Japan
Amnon Meisels, Ben-Gurion University, Israel
Pedro Meseguer, IIIA, Spain
Pragnesh Jay Modi, Drexel University, USA
Adrian Petcu (workshop chair) EPFL, Switzerland
Marius Silaghi, Florida Institute of Technology, USA
Milind Tambe, University of Southern California, USA
Makoto Yokoo, Kyushu University, Japan
Weixiong Zhang, Washington University, USA

Table of Contents

The Planning of the Oil Derivatives Transportation by Pipelines as a Distributed Constraint Optimization Problem

Fernando J. Moura Marcellino[1], Nizam Omar[2], and Arnaldo Vieira Moura[3]

[1] PETROBRAS Petróleo Brasileiro S/A, São Paulo, Brazil,
fmarcellino@petrobras.com.br
[2] FCI – Universidade Presbiteriana Mackenzie, São Paulo, Brazil,
omar@mackenzie.br
[3] Instituto de Computao – UNICAMP, Campinas, Brazil, arnaldo@ic.unicamp.br

This work models the Oil Derivatives Transportation by Pipelines as a Distributed Constraint Optimization Problem (DCOP), where the variables and constraints are distributed among multiple autonomous agents, which represent different terminals and refineries, and thus keep the privacy of the information associated to each of them. An adaptation of the Asynchronous Distributed Optimization algorithm (Adopt) is used to solve this DCOP and a comparison between the Adopt and the Synchronous Branch-and-Bound algorithm (SBB) is made. This performance evaluation is accomplished using the traditional metric of Number of Synchronous Cycles, and considers different heuristics for the Adopt and the SBB. In addition a preprocessing technique was developed to speed up Adopt. As it happened in the Adopts original work, the experimental results also prove its superiority over the SBB for this kind of problem.

1 Introduction

Distributed solution of problems by different agents, which behave in an autonomous way and collaborate with each other in order to reach a global solution, has been used to solve Constraint Satisfaction Problems (CSPs). The motivation is the existence of certain problems for which it is prohibitive or undesirable to concentrate all the knowledge in a single agent. In addition to the costs associated to information collection, there are the costs of communication and translation of the problem knowledge into a common format, and also the risk of keeping all the information in a single agent, which can be inconvenient for reasons of security or privacy [1].

The planning and scheduling of activities related to the distribution and transport of products have deserved increasing interest in the last 20 years. The annual costs of transport of products reach very high values due to the great volumes of raw materials and finished products that are moved [2]. This scenario worsens when one considers the distribution and transference of oil derivatives. When dealing with liquid and gaseous products, using pipelines represents the most economical and reliable transportation method, one that can operate continuously, a unique and distinguishing characteristic [3].

The main objective of this work is to model the Distributed Problem of Oil Derivatives Transportation Through Pipelines as a Distributed Constraint Satisfaction Problem (DisCSP). In order to control the complexity of the general problem, the Simplified Distributed Problem of Oil Derivatives Transportation by Pipelines (SDPOTP) is defined. In the SDPOTP, only the planning problem on a representative subgroup of the pipeline network in the region of São Paulo, Brazil, is considered. That is one of the most complex oil networks in operation in the world. Furthermore, among all the derivatives that travel in the pipelines, only the movements of the three most important ones are treated. Although less complex than the real problem, the SDPOTP is useful as it can serve as a reference, both for the model and the algorithms it used, when dealing with applications for the complete problem. No oversimplifying hypothesis were included that could jeopardize the main requirements of the real situation.

In this work, the DisCSP is modeled as a Distributed Constraint Optimization Problem (DCOP). The solution considered characteristics that were so far solved only in a centralized form. Among such characteristics is the total balance of products that may impose maritime import or export of derivatives.

The Asynchronous Distributed Optimization (Adopt) and the Synchronous Branch-and-Bound (SBB) algorithms were adapted to the peculiarities of the SDPOTP, using a number of different heuristics. Their efficiency was compared using the Number of Synchronous Cycles metric, the dominant metric for DCOP algorithms [4]. A preprocessing technique was also created, exploiting some peculiarities of the SDPOPT and of the Adopt algorithm, in order to improve its performance. The heuristics and the preprocessing technique are both described.

Finally, a generator was especially developed to create problem instances by dividing the problem into different categories, which are associated with different levels of abstraction and complexity. The results of the experiments showed that the Adopt algorithm was the most adequate one to treat the SDPOTP.

The text is organized as follows. Section 2 describes the problem and Section 3 discusses the DCOP and the Adopt algorithm. Section 4 formalizes the SDPOPT as a DCOP. Section 5 presents the test environment and the computational results. Some concluding remarks are found in Section 6.

2 Problem Description

The problem is the transportation of oil products through terrestrial pipelines, which interconnect refineries with both land and maritime terminals. The refineries are responsible for the production of oil derivatives. Both refineries and land terminals have storage capacity and must provide for local consumers. In turn, the maritime terminals, in addition to these characteristics, can also load products into and unload products from ships. All such terminals and refineries, henceforth named bases, are connected by a network of pipelines capable of conveying different types of derivatives they are multi-product pipelines.

In order to supply the local market at the various bases, derivatives must be produced in the refineries, and pumped through a certain sequence of pipelines, up to their destinations. The objective of the network operation is the proper

supply of products to all the consuming markets, taking into account the production of the refineries, the possibility of importation and exportation of derivatives by ships, the inventory in each of the bases as well as in the pipelines themselves, and the operational constraints of the pipelines. The results generated must indicate the levels of product inventory in each of the bases at the end of a certain time horizon, as well as the sequencing of movements of every product through each of the pipeline segments along this time.

Recently, the oil and natural gas industry in Brazil had to face new challenges. In this new scenario, all participants must have guaranteed access to the existing oil pipeline network, with the operation of the network becoming notably more difficult. In particular, the information belonging to different companies must be preserved. This new constraint renders inadequate the conventional centralized approach, since it sends all this information to a single process. This process executes a particular algorithm to construct a solution to the problem which, in turn, is distributed to all the different bases. The approach taken here is based on a distributed architecture, so that it can preserve the secrecy of the information in each base, can preserve the production information in the refineries, and can protect the information about imports and exports at maritime terminals. It also optimizes pipeline utilization by minimizing transportation costs.

2.1 Definition of the SDPOTP

The SDPOTP is a simplification of the real transportation problem through pipelines. The main simplifications are as follows:

- Only the planning problem is treated. In fact the transportation problem through pipelines consists of two interdependent problems which are the planning and the scheduling problem;
- A representative subgroup of the actual pipeline network of PETROBRAS in the area of São Paulo was considered;
- The three most prominent products were chosen, namely gasoline, diesel and naphtha. They represent more than 50
- Pipelines connecting the same bases are modeled as a single virtual pipeline;
- Net production includes production and importation; similarly, the net demand includes local demand, local consumption and local exportation;
- The initial inventories of the pipelines are not taken into account. This simplification is based on the assumption that the content of each pipeline at the end of the planning interval is exactly the same as it was at the beginning.

3 The Distributed Constraint Optimization Problem

The constraint satisfaction paradigm has proved naturally suitable to model many different types of complex combinatorial problems, and many successful applications have been developed, even in the oil industry area [5]. The SDPOTP is also a Constraint Satisfaction Problem (CSP). However, taking into account the inherent necessity of the problem to maintain the secrecy of the information

belonging to each of the bases, it was more adequate to model the SDPOTP as a DisCSP (Distributed Constraint Satisfaction Problem). Finally, since it is also an optimization problem whose the objective function is the total transportation cost of products through pipelines and ships, it was even more convenient to model it as a DCOP (Distributed Constraint Optimization Problem). As it will be shown, after analyzing the main constraints of this problem, namely the Conservation of Volume and the Occupation Limit of the Pipelines, it was clear that these are not hard constraints. In fact, they allow some flexibility, in such a way that both constraints can be represented by a value associated with their degree of satisfaction, and not simply by a predicate returning "True" or "False".

The minimization of transportation costs, in addition to providing a solution of greater quality, avoids an anomaly that might occur despite the satisfaction of all constraints. This anomaly would appear as circulating flows, i.e., closed cycles of product flows that leave and come back to the same original base.

3.1 DCOP Formalization

Modi et al. [6] formalize a DCOP as a tuple (A, V, D, C, F), were

- $A = \{a_1, a_2, \ldots, a_n\}$ is a set of n agents,
- $V = \{x_1, x_2, \ldots, x_n\}$ is a set of n variables, each one associated to an agent,
- $D = \{D_1, D_2, \ldots, D_n\}$ is a set of finite and discrete domains each one associated to the corresponding variable,
- $C = \{f_{ij} : D_i \times D_j \rightarrow N, \text{with } i, j = 1 \ldots n, i \neq j\}$ is a set of constraints, represented by a cost function f_{ij} for each pair of variables x_i and x_j,
- $F = \sum f_{ij}(d_i, d_j)$, with $x_i \leftarrow d_i, x_j \leftarrow d_j, x_i, x_j \in V$, is the objective function.

Only the agent has knowledge and control over values assigned to variables associated to it. Its goal is to find a valuation for all variables that minimizes F.

Once the DCOP framework surfaced as the most convenient one for the SDPOTP, the next step was to evaluate the best available algorithm to solve DCOP models. After eliminating algorithms that were not complete, only a few remained: the Synchronous Branch and Bound (SBB), until recently the only complete one [7], and 3 asynchronous algorithms. These were the Asynchronous Distributed Optimization (Adopt), the Optimal Asynchronous Partial Overlay (OptAPO) [8], and the Distributed Pseudotree Optimization Procedure (DPOP) [9]. Since the Adopt has a functionality of bounded-error approximation, which is used in this work, it was chosen as the representative of the asynchronous group.

3.2 The Adopt Algorithm

The Adopt algorithm was developed by Modi et al. [6]. It is an algorithm capable of finding excellent solutions to DCOP problems using only local asynchronous communication. Further, it shows a polynomial space complexity for each agent. The communication is local in the sense that an agent does not send messages

to all other agents, but only to the neighboring ones. Modi et al demonstrated that the Adopt algorithm is not only sound, but it is complete as well [6].

Before the search begins the agents must be sorted in a chained structure, just as for the SBB algorithm, or else in the form of a Depth-First Search (DFS) tree. The tree ordering is defined by a parent-child relationship that must form an acyclic graph. In this work, such as in Modi's, this sorting was obtained in a preprocessing phase. Given a constraint graph associated with a DCOP, several valid DFS orderings can represent the problem. An important aspect when comparing two different orderings is the depth of the trees, i.e., the distance of the longest path from the root to a leaf. According to Modi et al, it is intuitive to assume that trees with smaller depth must lead to better performances, since the information can flow up and down the tree more quickly [6]. This hypothesis, however, was not verified by Modi et al. In this work this comparison was done, using different heuristics for the ordering of the problem variables.

4 Modeling the SDPOTP as a DCOP

The problem was modeled (Fig. 1) as a directed graph $G = (V, E)$, where the bases form the set of vertices $V = \{b_1, b_2, \ldots, b_N\}$, and the pipelines form the set of edges $E = \{o_1, o_2, \ldots, o_N\} \subseteq V \times V$.

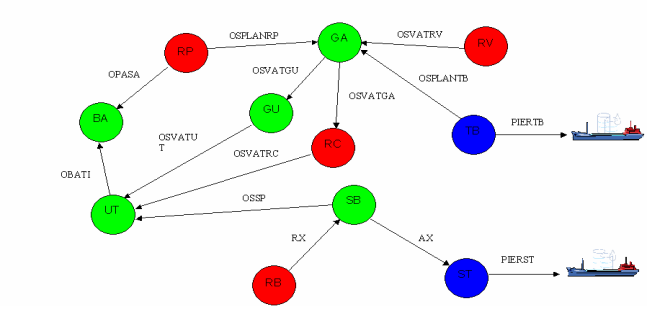


Fig. 1. Simplified Graph of the Pipeline Network of PETROBRAS in São Paulo

4.1 Instances of the SDPOTP

In order to define an instance of the problem the following is necessary:

- The time interval, T, called planning horizon, which was set equal to 1 week (168 hours);
- For each pipeline and product pair, the average normal flowrate, $FN_{p,r}$, indicating the flowrate in the conventional direction of operation for that product and in that pipeline;
- For each pipeline and product pair, the average reverse flowrate, $FR_{p,r}$, representing the flowrate in the opposite direction;
- The net production, $P_{b,r}$, for that base and that product, along the time T;
- The net demand, $D_{b,r}$, for that base and that product, along the time T;

– The storage capacity, $C_{b,r}$, that represents the total storage capacity for that product and in that base at the end of the time horizon T;
– The initial inventory, $IO_{b,r}$, that is the total inventory of that product and in that base at the beginning of the time horizon T;

4.2 Variables of the SDPOTP

For each pipeline and product pair:

– $Q_{p,r}$, is the volume of the product that is transported through the pipeline along the horizon T. The two directions of operation of the pipeline are represented by the sign of $Q_{p,r}$. The corresponding domains are:

$$DQ_{p,r} = \{Q_{p,r} \in Z | Q_{p,r} \geq (-FR_{p,r} \times T) \text{ and } Q_{p,r} \leq (FN_{p,r} \times T)\}$$

– $IF_{b,r}$ is the inventory of the product in the base at the end of period T. The corresponding domains are $DIF_{b,r} = \{IF_{b,r} \in N | IF_{b,r} \geq IMIN_{b,r}$ and $IF_{b,r} \leq IMAX_{b,r}\}$, where $IMIN_{b,r}$ and $IMAX_{b,r}$ are the limits of the inventory, namely, $IMIN_{b,r} = 0.4 \times C_{b,r}$ and $IMAX_{b,r} = 0.6 \times C_{b,r}$. Although the original problem has continuous domain values, the use of discrete values doesn't represent a restriction if one chooses the proper granularity for each variable.

4.3 The Agents

One agent was created for each pipeline and each base. The agent associated with a base is responsible for the variables of the final inventory of each product in that base, while the agent associated with a pipeline takes care of the volume of each product transported through this pipeline, during the planning horizon.

In contrast with the model in Modi et al [6], this model consists of a DCOP with several variables per agent, with each agent representing a distinct physical entity and having its own information and decision rules. The domains of the inventory and volume variables are discrete and are known only by the corresponding agent. Moreover, as will be seen, instead of only binary constraints, the model considered n-ary constraints.

In order to treat several variables per agent, the model uses a pseudo-agent for each variable. It behaves as if it were an agent with a single variable. Hence, each agent comprises several pseudo-agents, each one dealing with only one variable associated with one entity (base or pipeline), and one of the products.

4.4 Constraints of the SDPOTP

The constraints for the SDPOTP are as follows:

Final Inventory Not Null: this constraint is automatically satisfied by the definition of the domain of the $IF_{b,r}$ variables, whose lower bound is positive, that is, $IFMIN_{b,r} = 0.4 \times C_{b,r}$.

Storage Capacity: this constraint is also automatically satisfied, since the definition of the domain of the $IF_{b,r}$ sets its upper bound as $0.6 \times C_{b,r}$.

Volume Conservation: for each pair base and product we must have:

$$P_{b,r}D_{b,r} + IO_{b,r}IF_{b,r} + \sum_{p} Q_{p,r} = 0,$$

where the sum is over all pipelines p that are connected to base b. This constraint represents the volume conservation of each product with respect to each base. It involves not only the inventory variables for the base considered, but also the volume variables of all pipelines connected to that base. According to the DCOP framework, these constraints must be satisfied and, furthermore, an evaluation function must be minimized. The evaluation function for the volume conservation constraint was defined thus:

$$FVC = \sum_{b,r}(P_{b,r}D_{b,r} + IO_{b,r}IF_{b,r} + \sum_{p} Q_{p,r})^2$$

where the inner sum is over all pipelines p that are connected to base b.

Pipeline Occupation Limit: for any pipeline p we must have

$$\sum_{r}(|Q_{p,r}|/FZ_{p,r}) \leq T \times \delta_r,$$

where δ_r the pipeline availability, and $FZ_{p,r}$ is the flowrate of that product through that pipeline. Depending on the positive or negative sign of $Q_{p,r}$, the latter will be given by $FN_{p,r}$ or by $FR_{p,r}$, respectively. This constraint involves only the volume variables of each product that circulates through a pipeline. It represents a physical time limit for the operation of the corresponding pipeline during the time horizon T. This time is given by $(T \times \delta_r)$, where δ_r is the fraction of T during which the pipeline is operational. The estimation of value for δ_r takes into account the total maintenance time forecasted for the pipeline during the horizon T. For practical purposes, due to probable inaccuracies in the planning model, very high rates of pipeline occupation must be avoided. A tolerance limit of 90% was defined in this work, and occupation rates inferior to this value lead to a null value in the corresponding evaluation function. Thus, the evaluation function for the limit of occupation constraint was written in the form $FPOL = \sum_p FPOL_p$, where

$$FPOL_p = \begin{cases} 0, & \text{if } OR_p \leq 0.9 \\ OR_p - 0.9, & \text{if } OR_p > 0.9 \end{cases}$$

and where OR_p is the occupation rate of the pipeline, as follows:

$$OR_p = \big(1/(T \times \delta_r)\big) \sum_{r}(|Q_{p,r}|/VZ_{p,r}).$$

4.5 Optimization

In addition, a global evaluation function, $FCOST$, is formulated taking into account the optimization of the costs of transporting products through the

pipelines and the costs of exporting and importing by ships. Consulting costs established by TRANSPETRO, the company responsible for the operation of the pipeline network, it was clear that the three products considered in our simplified problem, i.e., gasoline, diesel and naphtha, all showed the same tariff, for each pipeline. Similarly, the average cost for ship freights and harbor expenses is basically the same for these three products. Thus, and recalling maritime transport was modeled as flows through virtual pipelines, called piers, we can represent the objective function of the problem by the following expression:

$$FCOST = \sum_{p,r} T_{p,r}|Q_{p,r}| + \sum_{i,r} T_{i,r}|Q_{i,r}|,$$

where $T_{p,r}$ and $T_{i,r}$ are the transportation tariffs per unit of volume of that product, through that pipeline or pier, respectively. Or, in a more compact way:

$$FCOST = \sum_{P,r} T_{P,r}|Q_{P,r}|.$$

where P is a generalized pipeline, encompassing the pipelines themselves and the piers, and $T_{P,r}$ is the transportation tariff per unit of volume of that product for that P. Therefore, we write the global evaluation function in the form:

$$F = w_{VC} \times FVC + w_{POL} \times FPOL + FCOST,$$

where w_{VC} is a weight that measures the importance of the volume conservation constraint, and w_{POL} a weight associated with the other constraints. The term $FCOST$ is not weighted, since it is already measured in actual cost units. As for the other components, they must be normalized with respect to that cost unit. In this work w_{VC} and w_{POL} were set to 1000. As we will see later, this choice proved to be an appropriate one, allowing for the satisfaction of the all constraints, while minimizing the $FCOST$ function. The criterion for choosing this value was basically to guarantee that $w_{VC} >> FCOST$ and $w_{POL} >> FCOST$.

4.6 Adaptations of the Algorithms for the DCOP

The original the SBB and Adopt algorithms of Modi et al [6] were adapted to the peculiarities of the SDPOTP. Their performance was confronted in this work.

The base and pipeline agents deal with more than one variable, i.e., $IF_{b,r}$ and Qpipeline product for each product, respectively. The pipeline agents are responsible for the FPOL term in the global evaluation function, and the base agents, in turn, take care of the FVC term. This situation imposes certain conditions on the priorization of the agents. Note that each base agent is responsible for the volume conservation constraint in that base. Hence, each agent responsible for a pipeline that is connected to that base must precede the base agent, in such a way that all involved values of the $Q_{p,r}$ variables have already been sent by the pipeline agents, and are available in the base agent at the moment of the constraint evaluation. The pipeline occupation limit constraints pose no precedence requirements to the agents, since this type of constraint involves only the volume variables of each product transported through the pipelines being, therefore, associated with variables controlled by the same pipeline agent.

4.7 Heuristics Employed

Heuristics used in Constraint Satisfaction Problems can be classified basically into two categories: Ordering of Domain Values and Ordering of the Variables. Two Ordering of Domain Values for the $IF_{b,r}$ and the $Q_{p,r}$ variables were used:

Increasing: the trivial ordering, with the values chosen from the smallest up to the largest one.

Alternating: a more elaborated ordering. It is guided by the satisfaction of the volume conservation and occupation limit constraints, while still trying to minimize the transportation costs. This goal suggests an ordering that seeks to minimize the flow of products through the pipelines while minimizing the difference between the production and the demand of each product in each base. For that, the domains were sorted in the following form:

- *Volume variables*: in order to minimize the flow of each product through each pipeline, the domains are scanned in increasing order of absolute values of the volumes. Thus, it starts always from the value 0, and then the next positive value is chosen, and then its negative symmetrical value, returning to the consecutive positive value, and so on, until the domain is exhausted. If the distribution between positive and negative values is asymmetric, when the smaller side finishes the other side is taken in increasing order of its absolute value, until the whole domain is exhausted.
- *Inventory variables*: aims at the minimum balance for each product in each base. It chooses first the inventory value that is closest to the product balance, where $BAL_{b,r} = P_{b,r} - D_{b,r} + IO_{b,r}$. After that, the consecutive value is chosen, and then its symmetrical with respect to the chosen initial value, returning to the consecutive value, and so on until the whole domain is scanned. If the distribution between the values around the initial value is asymmetric, when the smaller side finishes, the other side is taken in increasing order of absolute values until no value remains to be considered.

Example of the Increasing ordering:

$$DQ_{p,r} = \{-3, -2, -1, 0, 1, 2, 3, 4, 5\}$$
$$DIF_{b,r} = \{6, 7, 8, 9\}$$

For the Alternating ordering:

$$DQ_{p,r} = \{0, 1, -1, 2, -2, 3, -3, 4, 5\}$$
$$DIF_{b,r} = \{6, 7, 8, 9\} \text{ if balance is less than or equal to 6}$$
$$DIF_{b,r} = \{7, 8, 6, 9\} \text{ if balance is 7}$$
$$DIF_{b,r} = \{8, 9, 7, 6\} \text{ if balance is 8}$$
$$DIF_{b,r} = \{9, 8, 7, 6\} \text{ if balance is greater or equal to 9}$$

As to the *Variable Ordering*, this heuristic is related to the order in which the variables are prioritized in the construction of the algorithm search tree. As was seen, given the specificities of the SDPOTP, the $Q_{p,r}$ variables of each pipeline agent must have a priority higher than the $IF_{b,r}$ variables of base agents that

are connected to the pipeline. After this precedence is enforced, there remains a certain freedom for the prioritization of the volume and inventory variables. Two criteria for ordering these variables were used. One is based on the domain and the other relies on the degree of the node in the constraint graph. The *Domain* ordering respects the precedence mentioned above, and the variables are prioritized in increasing order with respect to the size of there domains. The *Degree* ordering prioritizes variables in decreasing order of the degree of its corresponding node in the constraint graph of problem. Thus, in the latter case, the variables that take part in a great number of constraints are assigned higher priorities. The *Degree* heuristic was employed in the original work on the Adopt algorithm by Modi et al [6]. Unlike the SBB algorithm, which sees only a chain structure, the Adopt algorithm admits branches with several children. As will be seen later, this characteristic leads to a superior performance. In this work, for the sake of comparison, the Adopt algorithm will be used with both structures, i.e., with a chain structure and with a tree structure.

When the Adopt algorithm was used with chain structure, both the Domain and Degree ordering heuristics were tested. The same was done for the SBB algorithm. On the other hand, a tree structure can only be used with the Adopt algorithm. In this work, when using a tree structure, a new heuristics, named Small Tree, was created to be used with the Adopt algorithm. Its main feature is the creation of small prioritization trees, in contrast to the chain structure, that, in fact, is a great depth tree with a single long branch. The heuristic Small Tree was obtained by the exploitation of the particularitities of the SDPOTP constraint graph, in order to generate a tree with the smallest possible number of levels, but still respecting the requirement of the Adopt algorithm that all variables related by a same constraint must be located in a same sub-tree, in the complete prioritization tree. The motivation for the study of this heuristic was the conclusion in Modi et al [6], which suggested that the use of a tree with smaller depth would speed up the Adopt algorithm.

4.8 The Preprocessing Method

Since DCOP is a NP-complete problem, the application of the corresponding algorithms is not efficient, in some cases. Here, the Adopt algorithm has a great advantage when compared to other distributed algorithms since its feature of bounded-error approximation makes it possible to generate solutions within a specified errror bound of the global optimal solution, even without knowing it a priori. This allows the quicker attainment of solutions that, even if not optimal, can guarantee quality within a certain tolerance interval. Taking advantage of this peculiarity, it is possible to create a simple preprocessing technique, which consists of determining a non null lower bound for the cost associated with the quality of the solution, and submitting it as a bounded-error for the Adopt algorithm in the following phase. In this way it is possible to get a global optimal solution with less effort, since the generation solutions with cost less than the stipulated lower bound can be rejected a priori.

The preprocessing phase proposed in this work consists of calculating the lower bound for the cost function of the SDPOTP, thus obtaining a non null

value which is assuredly smaller than or equal to the minimum possible cost for the problem. Taking into account the specific characteristics of the SDPOTP, the following lower bound was so defined:

$$LB = 0,5 \times \sum_{b} \sum_{r} \min(|\Delta_{b,r}|),$$

where $\Delta_{b,r} = D_{b,r}P_{b,r} + IF_{b,r}IO_{b,r}$.

5 Computational Experiments

5.1 The Test Environment

In order to evaluate a distributed algorithm it would be necessary to make use of N interconnected dedicated processors, an apparatus that, usually, is not available in most laboratories. Even if such facilities were present, the work load of each computer and the load of the communication network would normally be out of the experimenter control, these being two aspects that have a significant impact on the efficiency of the algorithms. As an alternative, the simulation in a single computer might be a convenient way to evaluate distributed algorithms.

This work used a simulator similar to that in Modi [10], where the algorithms Adopt and SBB were compared over the graph-coloring problem. The simulator was coded in Java, and each agent was implemented as a thread. Although each algorithm executes as a unique process, the use of threads for agents may cause subtle effects that can lead to different results when the same experiment is repeated. These differences, however, are not as important as the ones that would rise when using a different process for each agent. In fact, their effects appear as small fluctuations that hardly affect the reproducibility of the results when the same computer and operating system are repeatedly used. The advantage of this kind of simulator becomes evident with asynchronous algorithm, such as Adopt, since in this case the implementation can take advantage of parallel processing.

5.2 Instance Generator for the Problem

An instance generator for the SDPOTP was developed, based on actual historical data. An instance of the problem is defined by configuration parameters which are generated randomly, while still respecting the bounds imposed by the data. These parameters are, first, the storage capacities of the bases and the flowrates of the pipelines for each product, which were specified as in the real data. Secondly, there are the initial inventories of each product in each base, which were generated randomly from a uniform distribution, whose minimum and maximum limits were determined from the available data. That uniform distribution was implemented using the pseudorandom generator of the Java language.

Before coding the instance generator for the SDPOTP, a gradual approach was taken leading to increasingly more complex problems. The topology of the SDPOTP was fixed, including the number of bases, pipelines, piers and products,

as well as the bases of origin and destination of each pipeline. This configuration was the most complex setup tested. To get less complex instances, virtual pipelines and virtual bases were created by fusing bases and pipelines present in the complete problem. Thus several categories of the SDPOTP were created, at different levels of abstraction with respect to the complete problem. In the experiments, a category was represented by its number of agents and variables .

Here, 5 instances of the problem were generated, each containing 7 categories, from the simplest one with 7 agents and 7 variables, up to instances with 15 agents and 45 variables. With these categories, it was possible to compare the performance of the distributed solutions, over problems of different complexities.

5.3 Results of the Experiments

The experiments were divided into 4 phases. In each one, a graph for the average over the 5 instances of the corresponding category was drawn. The measures represent rates of the same quantity for the two distinct cases which are being compared. Since they are rates, the geometric average was used throughout.

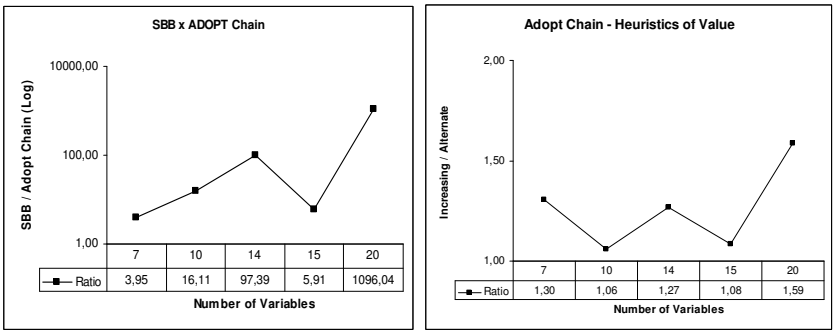


Fig. 2. (left) SBB and Adopt; (right)Value Ordering for the Adopt Chain

In the first phase, the SBB and Adopt algorithms were compared using the heuristics which were described previously. Since the SBB algorithm deals only with the chain structure, only this kind of prioritization was used for the Adopt algorithm. In addition, for each of the algorithms the best results were chosen, considering all the heuristics combinations that were tried. Fig. 2(left) shows the graph representing the average results for all instances. It shows the superiority of the Adopt algorithm over the SBB algorithm when the chain structure was used in the SDPOTP, in all the categories of the problem which were tested.

After confirming the superiority of the Adopt algorithm, the second phase of the experiments compared the results produced by this algorithm, in order to identify the best heuristics for both value ordering and variable ordering, still using the chain structure for the SDPOTP. First, the heuristics of value ordering was evaluated. The results are shown in Fig. 2(right). As can be seen, the most efficient heuristic for this case was the Alternate strategy, which was superior to the Increasing heuristics in all categories of the problem. Next, a

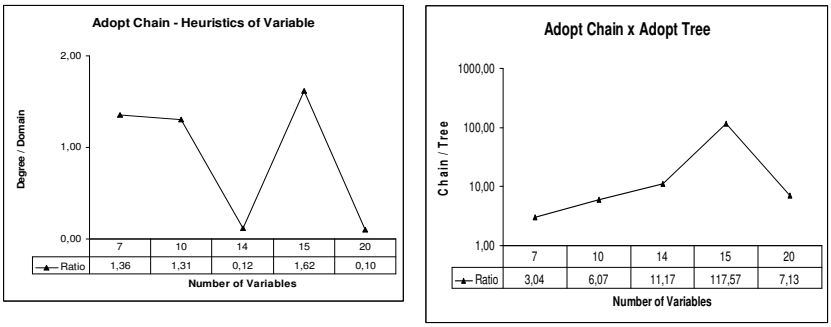


Fig. 3. (left)Variable Ordering for the Adopt Chain; (right)The Adopt Chain and Tree

similar evaluation was done, now considering the variable ordering heuristics. Fig. 3(left) depicts the results. It can be observed that although the heuristics on the domain size turned out to be the best, its superiority did not appear in all the categories, but only in three out of the five categories considered for the SDPOTP. We may conclude that the suitability of the variable ordering heuristics is more dependent on the particular structure of each SDPOTP that was considered.

In the next phase, both the chain and tree structures were tested with the Adopt algorithm. In each case, the prioritizations heuristics that presented the best results in the previous phase was used, i.e., the Alternate heuristics for value ordering and the Domain heuristics for variable ordering. This test led to the graph of Fig. 3(right). It shows the unquestionable superiority of the tree structure over the chain structure, when used with the SDPOTP. The next step analyzed the tree structure in more detail, seeking for even better efficiency. For this, the Small Tree heuristics was used with the same prioritization strategies. This heuristics was implemented, and its results were compared with the one based on domain size, which generated the best performance so far for the Adopt algorithm with a tree structure. The results are depicted in Fig. 4(left). For all categories, the small tree using the domain strategy yielded the best performance for the SDPOTP.

In the last phase, the preprocessing technique developed for the Adopt algorithm was evaluated. In order to verify if it speeds up the Adopt algorithm, the tests of the previous phase were repeated using the same categories. Knowing that the best heuristics for the Adopt algorithm with the tree structure was the Alternate strategy for value ordering and the Small Tree for variable ordering, these same heuristics were used in this experiment. The results proved clearly the advantage of the preprocessing mechanism with the Adopt algorithm, without compromising the quality of the solution. The results are in Fig. 4(right).

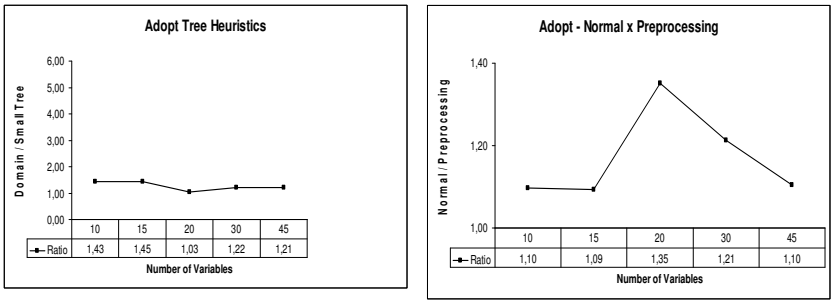


Fig. 4. (left)Domain and the Small Tree Heuristics; (right)Preprocessing Technique

6 Conclusions

This work showed the efficacy of modeling the SDPOTP as a DCOP, which, in contrast to a DisCSP model, does always have a solution with a certain level of quality. In other words, any DCOP has a solution that is optimal with respect to a certain quality criterion. In the DCOP approach, the advantages of using the Adopt algorithm for obtaining optimized solutions were clearly shown. Unlike the original work of [6], where only binary constraints were considered and where each agent dealt with only one variable, this work uniformly treated n-ary constraints and several variables per agent. Even with these extensions, the Adopt algorithm proved to be quite adequate to solve the SDPOTP. As also shown in the original work, the Adopt algorithm was also superior to the SBB algorithm, an admittedly complete algorithm for DCOPs.

In addition to its greater efficiency, Adopt algorithm clearly better fulfills the requirements for privacy when solving the SDPOTP since, unlike the SBB algorithm, it does not require that the agents have access to the information about the global upper bounds during the search process. Instead, it chooses values based only on the local information available for each agent.

Given the facility for generating bounded-error solutions by the Adopt algorithm, it was possible to take advantage of a preprocessing technique which was specially developed for the SDPOTP. This technique caused a significant increase in the search efficiency, without compromising the solution quality.

Specific heuristics were created for the SDPOTP. For value ordering, the Alternate heuristics was developed, and it proved to be superior when compared with the Increasing strategy. In the same vein, the heuristic Small Tree for variable ordering was created, as suggested in [6], where it was conjectured to be superior to the Degree heuristics. For the SDPOTP, this superiority was confirmed in this work, by showing that the Small Tree heuristics was superior to the Domain heuristics, which, in turn, was proved superior to the Degree heuristics.

It was shown that the distributed approach leads to a natural solution of global product balance in the bases, considering the importation and exportation of products by ships. It was demonstrated that, based only on local information

and rules, the base agents where harbor installations are present can determine the necessities of importation and exportation, in order to supply all other agents.

As in the original work of [6], the Adopt algorithm had its distributed characteristics implemented by simulation. This approach masks some of the problems of a predominantly asynchronous algorithm, such as, for example, the effect of the overhead caused by messages exchanges between agents. It would be important to repeat the tests presented here using really distributed independent interconnected processors. This would allow for a more effective comparison between synchronous and asynchronous algorithms by exploring the hardware parallelism. Further, a really distributed implementation would permit the test of communication complexity metrics, an issue that was not considered here. These tests could expose disadvantages of the asynchronous algorithms since, in general, they are more demanding in terms of communication between agents.

References

1. Bessire, C., Maestre, A., Meseguer, P.: Distributed dynamic backtracking. In: Proc. Workshop on Distributed Constraints, IJCAI-01, Seatle. (2001)
2. Rejowski, R., Pinto, J.M.: Scheduling of a multiproduct pipeline system. Computers and Chemical Engineering **27** (2003) 1229–1246
3. Sasikumar, M., Prakash, P., Patil, S., Ramani: Pipes: a heuristic search model for pipeline schedule generation. Knowl-Based System **10** (1997) 169
4. Yokoo, M., Durfee, E., Ishida, T., Kuwabara, K.: The distributed constraint satisfaction problem: Formalization and algorithms. Knowledge and Data Engineering **10** (1998) 673–685
5. Accioly, R., Marcellino, F.J.M., Kobayashi, H.: Uma aplicação da programação por restrições no escalonamento de atividades em poços de petróleo. In: Anais da Sociedade Brasileira de Pesquisa Operacional., Sociedade Brasileira de Pesquisa Operacional. (2002) 340 In Portuguese.
6. Modi, P.J., Shen, W., Tambe, M., Yokoo, M.: Adopt: Asynchronous distributed constraint optimization with quality guarantees. Artificial Intelligence Journal (2005)
7. Hirayama, K., Yokoo, M.: Distributed partial constraint satisfaction problem. In: Principles and Practice of Constraint Programming. (1997)
8. Mailler, R., Lesser, V.: Solving distributed constraint optimization problems using cooperative mediation. In: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems. (2004)
9. Petcu, A., Faltings, B.: A scalable method for multiagent constraint optimization. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI-05. (2005)
10. Modi, P.J.: Distributed Constraint Optimization for Multiagent Systems. PhD thesis, University of Southern California (2003)

DisChoco: A platform for distributed constraint programming

Redouane Ezzahir,[1] Christian Bessiere,[2] Mustapha Belaissaoui,[3] and El Houssine Bouyakhf[1]

[1] LIMIARF/FSR, U. of Mohammed V Agdal, Maroc
{bouyakhf,ezzahir}@fsr.ac.ma
[2] LIRMM (CNRS / U. of Montpellier, France
bessiere@lirmm.fr
[3] ENCG - U. of Hassan I, Maroc
m.belaissaoui@encg-settat.ma

Abstract. Open-source platforms are very useful for development and experimentation in constraint programming. However, until recently, no such platform existed for distributed constraint programming. This paper presents DisChoco, a platform for distributed constraint programming. DisChoco is a Java library implemented using the Choco solver and simple agent communication infrastructure (SACI). DisChoco can be used for simulation of a multi-agents environment on a single Java virtual machine, or performed in an environment physically distributed for a realistic use. DisChoco takes into account agent with a complex local problem, message loss, message corruption, and message delay. The implementation of DisChoco was made to offer a modular software architecture which accepts extensions easily. This paper presents the software architecture and illustrates how to implement a specific protocol.

1 Introduction

Constraint programming is a general framework that can formalize various problems in AI. Many theoretical and experimental studies have been performed, and various sophisticated centralized solvers have been developed (Ilog Solver, Chip, Choco, Gecode, etc.). Constraint solvers have the advantage that the user can concentrate her effort on the modelling of the problem, letting the solver run with the 'by default' solving characteristics. But if a researcher wants to implement and test her own techniques, black-box solvers are not the adequate tool. Open-source platforms permit to incorporate and test new ideas in constraint programming without the burden of re-implementing from scratch an ad-hoc solver. Some of the existing constraint solvers are open-source: Choco, Gecode, etc. In [9], the distributed constraint satisfaction problem (DisCSP) is formalized as a constraint satisfaction problem in which variables are distributed among multiple automated agents. The agents solve local constraint satisfaction sub-problems and a communication protocol between agents is performed, in order to allow the distributed system converge to a global solution.

Writing distributed applications is difficult because the programmer has to explicitly juggle with many quite different concerns including centralized programming, asynchronous and concurrent programming, distributed structure, fault tolerance security, open computing and others. The distributed constraint solvers known by the DCR community are MELY [6] , DiSolver [4] and Frodo [5]. We propose a new distributed solver namly DisChoco. With DisChoco, our goal is to propose an open-source platform in which it is possible to implement easily and simulate as much as possible the different concerns of distributed constraint programming.

DisChoco is a Java library built on top of the Choco Java open-source solver. Communication is performed via the simple agent communication infrastructure (SACI) if the agents are implemented on distant machines. Otherwise (simulation) the communication is performed via a local communication simulator. The implementation of DisChoco was made to offer a modular software architecture which accepts extensions easily. DisChoco can be used for simulation of a multi-agents environment on a single Java virtual machine, or performed in an environment physically distributed for a realistic use. Each agent in the environment is executed asynchronously in a separate execution thread, and communicates with its peers through message exchange. DisChoco takes into account agent with a complex local problem, message loss, message corruption, and message delay. In this paper, we present the software architecture of DisChoco and the structure of DisChoco agents. We illustrate how to implement a specific protocol. We give experimental results on an ABT implementation.

2 Software Architecture

The DisChoco platform is a Java library implemented using the Choco solver as agent local solver. DisChoco has a communication interface, ChocoCommunation, which makes it possible to implement any mean of communication between local or distant processes. To simulate a multi-agent environment in a single Java virtual machine, each agent must be executed asynchronously in a separate execution thread, and must communicate with its peers through message exchange. For real applications, DisChoco can be performed in an environment physically distributed using the SACI library (Simple Agents Communication Infrastructure [3]). ABT family [1] is the first distributed class of algorithms implemented in this framework.

2.1 User interface

DisChoco is based on Choco, a platform for research in centralized constraint programming and combinatorial optimization. This significant choice of design enabled us to keep the same philosophy of design as Choco and to benefit from the modules already implemented in it. We kept the same instructions notations of programming that were used in Choco. Figure 1 is an example of DisChoco code. This example illustrates the steps of declarations of the objects handled.

```
1. DisProblem DisCSP = new DisProblem ("Example");

2. List infeasePair = {(0,0),(0,3),(1,1)(1,3),(2,0),(2,1)};

3. ChocoAgent ag1 = DisCSP.makeAgent("ABT","Agent_1");
4. ChocoAgent ag2 = DisCSP.makeAgent("ABT","Agent_2");

5. Problem p1 = ag1.getLocalProblem();
6. IntVar X0 = p1.makeEnumIntVar("X0",0, 3);
7. IntVar X1 = p1.makeEnumIntVar("X1",0, 3);
8. P1.post(p1.neq(X0,X1));
9. P1.post(P1.infeasPairAC( X0, X1, infeasePair ));

10. Problem p2 = ag2.getLocalProblem();
11. DisVar Y0 = p.makeEnumIntVar("Y0",0, 3);
12. DisVar Y1 = p.makeEnumIntVar("Y1",0, 3);
13. P2.post(p2.neq(Y0,Y1));
14. P2.post(P2.infeasPairAC( Y0, Y1, infeasePair ));

15. DisCSP.post(DisCSP.infeasPairAC(ag1,X0,ag2,Y1,list));
16. DisCSP.post(DisCSP.neq(ag1,X0, ag2, Y1,));

17. DisCSP.solve();
```

Fig. 1. Example of DisChoco code

We start with distributed problem declaration (line 1) followed by the agent declaration which specifies the resolution algorithm to be used (lines 3–4). Next, the declaration of the variables and local constraints of each agent is made the same way as in Choco (lines 5–9 for Agent 1 and lines 10–14 for agent 2). Afterwards, we post the constraints that are external to agents, that is, the constraints that involve several agents (lines 15–16). Finally, we launch the resolution (line 17). We can point out that as in Choco, the constraints declaration and the resolution are clearly separate.

2.2 Object-Oriented model of agent systems

In traditional software development the need for modelling techniques and development methodologies has been recognized for a long time. Several modelling techniques and design methodologies have been developed and used extensively. Among the most successful techniques are the various object-oriented approaches [7]. An object is usually described as a holder of state information together with some behavior using operations upon the state information. Object-oriented methodologies provide support for identifying objects, and allow abstraction via object classes and inheritance via class hierarchies. We have used an object-oriented model for implementing agent systems in the DisChoco

platform. Distributed problems, agents, variables, constraints, and messages, can naturally be represented by objects:
Distributed Problem: All information is encapsulated in a Distributed Problem object (DisProbem) rather than in structures of global data. In the way of an environment physically distributed (several machines) DisProbem objects gather the necessary information for the set of agents performing on other machines.
Agents: Each agent is represented by an Agent object: ChocoAgent. The ChocoAgent object has an identifier that is unique in the system (AgentID), a name (AgentName), a local problem (Choco.Problem), a set of external constraints which connects this agent to the other agents in the system, and a set of properties used in the resolution.
Variables: For each agent, we have two classes of variables: local variables that are already defined in Choco and external variables which we define as a new class, **ExternalVar**, containing external variable knowledge. ExternalVar models a variable belonging to another agent, constrained with the agent. Each variable has a unique identifier VarID that is given by the pair (AgentID, index) where index is the variable index in the local problem and AgentID is the identifier of the agent that owns it. The ExternalVar class implements Event interface to perform propagation events: it propagates instantiations or value removal events when they occur.
Constraints: We have defined a new class of constraints: ExternalConstraint. The ExternalConstraint represents the external links (inter-agent constraints) between local variables and External variables. The ExternalConstraint class implements the Listener interface for receiving ExternalVar modification events and performing constraint propagation. Each ExternalConstraint is recorded on any ExternalVar involving it.
Agent view: The view of an agent is modeled with an object: AgentView. The AgentView class gathers the set of external variables and implements some methods that allow handling the view of the agent.
Communication: To define a communication system for distributed resolution, we have implemented an interface: **ChocoCommunication**. The ChocoCommunication interface defines the necessary methods for managing message communication: sendMessage, receiveMessage and broadcastMessage.
Message: All the types of messages sent in the system may be implemented by the class ChocoMessage and extended from SACI.Message. ChocoMessage defines the necessary methods to handle messages: getSenderID, getReceiverID, getLogicalTimeCounter, getMessageType (Value, Nogood, Heuristic,...).
MailerAMDS : The communication system is represented by MailerAMDS class. The MailerAMDS class implements ChocoCommunication interface, serves as a message relay in the system (see Figure 3) and implements an Asynchronous Message Delay Simulator.
DSolver : The resolution search is accomplish and controlled by DSolver object. This entity serves to solve distributed problem. DSolver is inherited from MailerAMDS.

2.3 Model of distributed solver

DisChoco provides two cases of usage: simulation and realistic use. Both use a communication system for managing message communication between agents. For the simulation use, the multi-agent system is implemented in a single Java virtual machine environment. Thus, each agent is a simple thread and the communication system is locally implemented. For solution detection of an asynchronous search, DisChoco uses Silaghi et al.'s solution detection that allows to detect solution before quiescence [8].

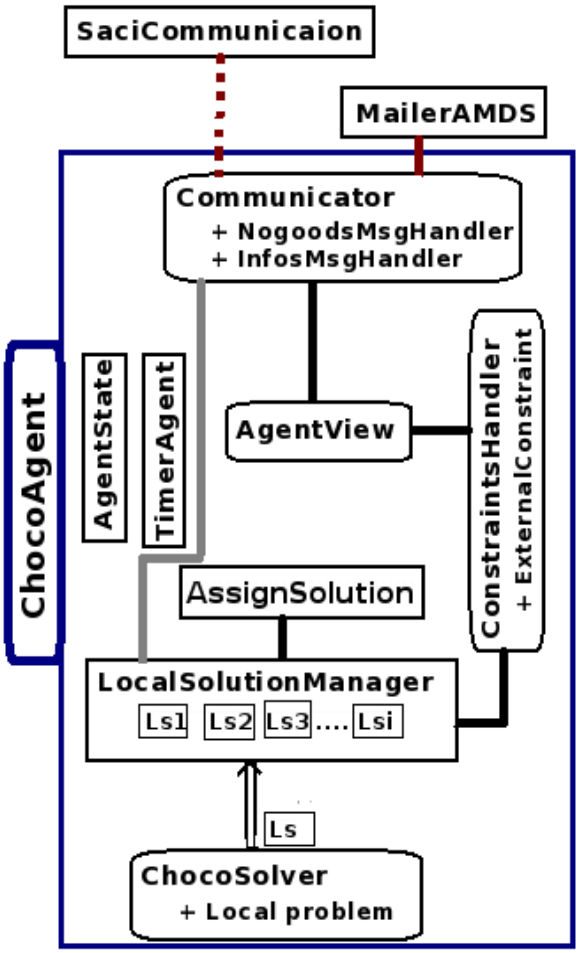


Fig. 2. The overall structure of a DisChoco agent.

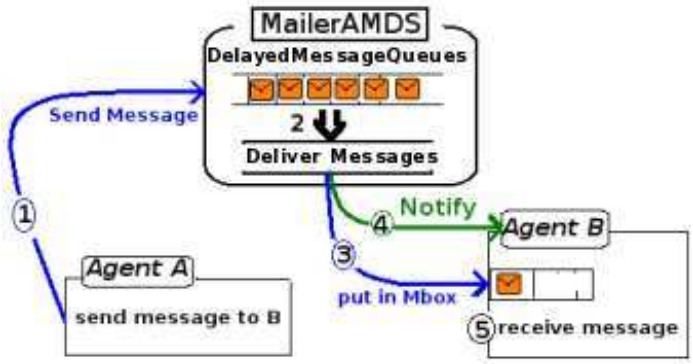


Fig. 3. The messages communication mechanism for simulation use of DisChoco.

Simulation use The user can simulate a distributed protocol within DisChoco. For this case, a communication system is implemented to simulate realistic internet network models. The communication system is represented by MailerAMDS class. The MailerAMDS class implements ChocoCommunication interface, serves as a message relay in the system (see Figure 3) and implements an Asynchronous Message Delay Simulator [11]. For each agent, the MailerAMDS maintains a queue of delayed messages (DelayedMessagesQueues). The delayed messages queue is a priority queue in which the messages are sorted by delivery time (in step). Each agent has a mailbox where messages for that agent are inserted. When a message is put in the mailbox the agent is notified. When an agent (A) wants to send a message to an agent (B), the message exchange between agents is performed as follows:

- Agent (A) is responsible for computing the content of the message. (A) builds a ChocoMessage object that contains a destination ID.
- Agent (A) sends this message over to the MailerAMDS.
- The MailerAMDS reads the message, finds the ID of the intended recipient, assigns to the message a delivery time which is the sum of the current value of the Mailers logical time counter (LTC) and the selected delay (in steps), and places it into the recipient's delayed messages queue that corresponds to intended recipient ID.
- At each step, the MailerAMDS delivers delayed messages with LTC less than or equal to its LTC and notifies concerned agents.

In addition, MailerAMDS is used to check termination condition and to simulate message loss and message corruption. The termination condition occurs when there are no incoming messages, all message queues are empty, and all agents are idle. The MailerAMDS checks if all messages queues are empty. This is a necessary condition for termination. If so, this information is used by the distributed solver (DSolver) to terminate execution of the resolution. To achieve

this goal, each agent has a Boolean variable (idle_state), indicating if the agent is in idle state. When the necessary termination condition is detected by MailerAMDS, the DSolver checks if all agents are idle (i.e., Idle_state=true). If so, DSolver stops all running threads (Agents) and prints the result. The simulation of messages loss is obtained by calling MailerAMDS.withMessageLoss() method, by which we mean that at any point it is possible that a message is not delivered and the recipient agent does not know if this message was sent. The MailerAMDS holds a variable (MsgLossProbabililty) to define message loss probability. When it is fixed to a specific value, the MailerAMDS generates a random value and checks if this value is smaller than message loss probability. If so, it deletes the message without delivering it. To generate corrupted messages we have defined for AllMessage object a function corrupt(): T→T for each message type T. It is used to modify the content of the message. When the mailer receives a message and its withMessageCorruption variable is true, it generates a random value and checks if this value is smaller than the value of message corruption probability (corruptprob). If so, the MailerAMDS converts the message to corrupted message by calling the message.corrupt() function. The impact of the messages corruption on resolution depends on the algorithm used. The resolution search is accomplish and controlled by DSolver object. This entity serves to solve distributed problem. DSolver is inherited from MailerAMDS. It:

- starts all agents of the system,
- finishes once all agents finished,
- announces solution, statistics.

```
AgentName = Square_5
Algorithm = ABT
DisProblemFactory = dischoco.samples.sudoku
#ProblemFileName = sudoku_5.txt
Comm=saci
#Comm=local
MasterAgent = false
SimpleAgent = true
MasterHostName = 211.44.34.11
NbAgents = 9
ConnectToMasterTimeOut = 300
ResolutionTimeOut = 7200
```

Fig. 4. Example of problem property file

3 Structure of DisChoco Agent

Realistic use DisChoco can be performed in an environment physically distributed using SACI library. The system is controlled by an agent Master that extends saci agent. Each agent has a **SaciCommunication** entity that allows him to communicate with other agents. A Main class has been implemented in DisChoco for loading problem properties file, instantiate agent and start resolution. Figure 4 shows an example of Sudoku problem property file. When Main class is started, it creates a new instance of SaciCommunication using file property. SaciCommunication tries to connect at master host. If it cannot connect in the duration specified by the ConnectToMasterTimeOut argument, a message error is displayed, otherwise the resolution starts. Distributed constraint programming and distributed combinatorial optimization are problems where each agent owns a local problem (variables and constraints) and where some constraints involve variables from several agents. Each agent executes an algorithm to solve the problem. Any algorithm run by an agent handles an AgentView and uses procedures with specific role: constraint handling, message handling, nogood handling, costs handling... These procedures differ from an algorithm to another. The use of virtual methods and dynamic selection of the right function implementing the polymorphic method has many impacts on the architecture of DisChoco. The overall structure of a DisChoco agent is shown in Figure 2. The agents are defined by ChocoAgent objects that gather and define all categories of component handlers:
AgentState: This entity records agent state (running, idle, solution, ...) and some statistic performance parameter (number of concurrent constraint checks).
TimerAgent records start time and time limit of resolution.
ConstraintsHandler: This entity handles constraints. The ConstraintsHandler interface defines a method for initializing the constraints to evaluate and it defines procedures for processing these constraints. (For example Distributed breakout algorithm uses all external constraints to solve the problem; ABT uses only the constraints with higher agents. In addition, constraint processing in DBA is different from that in ABT). Thus, both ABTConstraintsHandler and DBAConstraintsHandler must implement the ConstraintsHandler interface.
MessagesHandler: Modeled by MessagesHandler interface. This interface defines ProcessMessage method to treat communicating message. The user can define the core of this method according to the used resolution protocol. For example in ABT implementation we have implemented this interface with two components: InfosMsgHandler and NogoodsMsgHandler. They respectively process info messages and nogood messages.
Communicator: Defines communication procedure (send, receive and broadcast message), records agents neighbors, and dispatches arrived messages to its corresponding handler and to the TerminationDetector.
TerminationDetector: This class implements MessagesHandler, it processes message if it contains a termination detecting information. For solution detection of an asynchronous search, DisChoco uses Silaghi et al.'s solution detection (see above).

ChocoSolver: This entity defines the local solver. The ChocoSolver class extends thread objects. It controls local solver. The local solver is a Choco solver that is responsible of local problem resolution and it reports local solutions to the agent. The ChocoSolver can be started, suspended, resumed, and stopped by its owner agent.
LocalSolutionManager: The LocalSolutionManager class extends thread objects and exchanges solution with local solver. The LocalSolutionManager stores reported local solutions and uses interchangeability to avoid redundant work [2].
Main class of program: The dischoco.Main class task is to load the environment class, instruct it to load the problem from a file, create a distributed representation of the problem (i.e., create a new instance of DisProblem class using a specific Factory class), create the agents for simulation use or a single agent in realistic use, and instruct the environment to start the agents. For example **dischoco.samples.urbdcsp** is a uniform random binary DisCSP Factory class.

During asynchronous search, ChocoAgent runs a generic procedure presented in Figure 5. The agent starts its local solver (ChocoSolver) and waits for state of the local problem (Fig. 5, line 2). If not feasible it sends a no solution message and terminates. Otherwise (i.e., one solution is found), the agent starts its LocalSolutionManager that manages reported local solutions from ChocoSolver. Next, the agent starts the distributed search (Fig. 5, line 4). It chooses an instantiation (a local solution), sends it through its outgoing links. After that, the agent waits for arrival messages. In parallel, the local solver and the local solution manager continue to exchange discovered local solutions, so that other solutions are immediately available if needed. The function waitForArrivalMessages() (Fig. 5, line 7) is used to set agent in a wait state. When the agent is notified that a message has arrived, the agent suspends its local solver if needed, and calls DispatchReceivedMessages() defined in its communicator. This procedure dispatches received messages to all messages handler components.

DisChoco is extensible and powerful enough to allow users to extend it and implement any distributed constraint/optimization algorithm. In this section we present a sample implementation of ABT, the most well-known solving protocol for DisCSPs [10, 1]. The first step for implementing this protocol is to create a new ABT agent class. Our ABT agent must extend **dischoco.search.ChocoAgent** and define the CheckAgentView(), Backtrack() and checkAddLink() procedures. The ChocoAgent class creates a Communicator, an AgentState, a TimerAgent and a local Choco problem, and initiates all implemented components handlers. The Choco problem allows the user to define local variables, local constraints and their propagation procedures. The next step focuses on the components handler implementation. The requested components handler are:
ABTConstraintsHandler: it must extend AbstractConstraintsHandler class and define methods for handling constraints. The ABTConstraintsHandler processes constraints that are activated by an AgentView modification event.
ABTInfosMessagesHandler and **ABTNogoodsHandler** that both must extend AbstractMessageHandler class. The ABTInfosMessagesHandler processes Infos messages and updates the AgentView. The ABTNogoodsHandler defines

methods for processing the Nogood messages and some procedures for storing an resolving nogoods.
The final step is to record the ABTInfosMessagesHandler and the ABTNogoodsHandler in the communicator of the agent.

```
0. start ChocoSolver;
1. WaitForLocalProblemState();
2. if( local problem is not feasible)
     send no solution and terminate;
3. start LocalSolutionManager ;
4. start distributed search;
5. End= false;
6. While (not End)
8.          waitForArrivalMessages();
9.          If(ChocoSolver has not finished)
                  suspend ChocoSolver;
10.         communicator.DispatchReceivedMessages();
11.         If(ChocoSolver has not finished)
                  resume ChocoSolver;
```

Fig. 5. The main procedure running by a ChocoAgent in DisChoco.

4 Example of Implementation

DisChoco is extensible and powerful enough to allow users to extend it and implement any distributed constraint/optimization algorithm. In this section we present a sample implementation of ABT, the most well-known solving protocol for DisCSPs [10, 1]. The first step for implementing this protocol is to create a new ABT agent class. Our ABT agent must extend **dischoco.search.ChocoAgent** and define the CheckAgentView(), Backtrack() and checkAddLink() procedures. The ChocoAgent class creates a Communicator, an AgentState, a TimerAgent and a local Choco problem, and initiates all implemented components handlers. The Choco problem allows the user to define local variables, local constraints and their propagation procedures. The next step focuses on the components handler implementation. The requested components handler are:
ABTConstraintsHandler: it must extend AbstractConstraintsHandler class and define methods for handling constraints. The ABTConstraintsHandler processes constraints that are activated by an AgentView modification event.
ABTInfosMessagesHandler and **ABTNogoodsHandler** that both must extend AbstractMessageHandler class. The ABTInfosMessagesHandler processes Infos messages and updates the AgentView. The ABTNogoodsHandler defines methods for processing the Nogood messages and some procedures for storing an resolving nogoods.

The final step is to record the ABTInfosMessagesHandler and the ABTNogoodsHandler in the communicator of the agent.

5 Experiments

We tested DisChoco on uniform random binary DisCSPs. A uniform random binary DisCSP class is characterized by $(\#A, n, d, iC, iT, Cx, C, T)$ where $\#A$ is the number of agents, n is the number of variables, d the number of values per variable, iC the number of intra-agent constraints on each agent, iT the constraint tightness defined as the number of forbidden value pairs on intra-agent constraints, Cx the number of edges in the agents connexion graph, C the number of interagent constraints on each edge, and T the interagent constraint tightness. We have tested DisChoco with 100 problem instances of five different classes of uniform random binary DisCSPs. These classes are:

C1: (4, 16, 8, 6, 28, 6, 5, 28)
C2: (6, 30, 8, 8, 22, 8, 5, 22)
C3: (6, 30, 8, 10, 22, 15, 6, 22)
C4 (6, 48, 4, 18, 6, 15, 7, 6)
C5 (9, 54, 4, 15, 6, 10, 5, 5)

Figure 6 shows the results without message delay and with message delay simulator. #ccks represents number of concurrent constraint checks and #Tmsgs the total number of messages sent in the system. We compare Dischoco both without message delay and with message delay simulator. This simple experiment shows that it is easy to evaluate any implemented protocol with performance measures implemented in Dischoco.

	Without Message Delay		With Message Delay	
Class	#ccks	#Tmsgs	#ccks	#Tmsgs
C1	1522	20	1622	21
C2	3819	36	4006	39
C3	24163	2546	25767	2679
C4	37366	9934	39453	10359
C5	14408	1266	15345	1312

Fig. 6. Simulation results averaged over 100 problems per class.

6 Conclusion

We presented DisChoco, a platform for distributed constraint programming. DisChoco is used for simulation of a multi-agent system in a single Java virtual

machine and can be performed in an environment physically distributed using SACI infrastructure. The platform allows all the agents in the system to execute concurrently, and can be extended to implement any distributed constraint programming method. For simulation use, DisChoco models the realistic communication network by the MailAMDS simulator. The MailAMDS simulator takes into account message loss, message corruption, and message delay. Future work will address efficiency issues, parallel and distributed constraint propagation.

References

1. C. Bessiere, I. Brito, A. Maestre, and P. Meseguer. Asynchronous backtracking without adding links: A new member in the ABT family. Artificial Intelligence, 161:7–24, 2005.
2. D.A. Burke and K.N. Brown.. Applying interchangeability to complex local problems in distributed constraint reasoning. In *Proceedings AAMAS'06 workshop on Distributed Constraint Reasoning*, Hakodate, Japan, 2006.
3. J.F. Hübner and J.S. Sichman. SACI: A simple agent communication infrastructure. http://www.lti.pcs.usp.br/saci/, 2005.
4. Y. Hamadi and Y. Chong Distributed Log-based Reconciliation In European Conference on Artificial Intelligence Riva del Garda, Italy ,2006.
5. A. Petcu. Frodo: a FRamework for Open/Distributed Optimization. Technical Report Technical Report EPFL:2006/001, LIA, EPFL, CH-1015 Lausanne, http://liawww.epfl.ch/frodo/, 2006.
6. Michel Galley Distributed constraint programming platform using sJavap. http://cs.fit.edu/Projects/asl/#MELY.
7. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.
8. M.C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with aggregations. In *Proceedings AAAI'00*, pages 917–922, Austin TX, 2000.
9. M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 614–621, 1992.
10. 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.
11. R. Zivan and A. Meisels. Message delay and discsp search algorithms. In *Annals of Mathematics and Artificial Intelligence*, Springer Netherlands, vol. 46, no. 4, pp. 415-439, 2006

Asynchronous Forward-Bounding with Backjumping ⋆

Amir Gershman, Amnon Meisels, and Roie Zivan
email: {amirger,am,zivanr}@cs.bgu.ac.il

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

Abstract. The Asynchronous forward-bounding (*AFB*) algorithm [GMZ06] is a distributed constraint optimization problem (*DisCOP*) solver. We extend the *AFB* algorithm by adding a backjumping mechanism, resulting in the *AFB-BJ* algorithm. *AFB-BJ* is based on accumulated information on bounds of all values and on processing concurrently a queue of candidate goals for the next move back. The *AFB-BJ* algorithm is shown experimentally to be a very efficient algorithm for *DisCOP*s. The algorithm is described in detail and its correctness proven. Experimental evaluation of *AFB-BJ* on random *Max-DisCSP*s reveals a phase transition as the tightness of the problem increases. This phenomenon was found first for *Max-CSPs* [LM96] and more recently also for DisCOPs [GMZ06]. *AFB-BJ* outperforms asynchronous distributed optimization *ADOPT*, as well as *AFB*, on the harder instances of randomly generated DisMax-CSPs.

1 Introduction

The Distributed Constraint Optimization Problem (*DisCOP*) is a general framework for distributed problem solving that has a wide range of applications in Multi-Agent Systems and has generated significant interest from researchers [MSTY05,ZXWW05]. *DisCOP*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 variables and communicate with each other, attempting to generate a solution that is globally optimal with respect to the costs of constraints between agents (cf. [MSTY05,PF04]).

Distributed *COP*s are an elegant model for many every day combinatorial problems that are distributed by nature. Take for example a large hospital that is composed of many wards. Each ward constructs a weekly timetable assigning its nurses to shifts. The construction of a weekly timetable involves solving a constraint optimization problem for each ward. Some of the nurses in every ward are qualified to work in the *Emergency Room*. Hospital regulations require a certain number of qualified nurses (e.g. for Emergency Room) in each shift. This imposes constraints among the timetables of different wards and generates a complex Distributed COP [SGM96,KM05].

The Asynchronous Forward-Bounding (*AFB*) algorithm was presented in [GMZ06], and found to outperform *ADOPT*, the state of the art algorithm for solving DisCOPs [MSTY05]. Previous search algorithms for solving *DisCOPs* perform backtracking in

⋆ The research was supported by the Lynn and William Frankel Center for Computer Science, and by the Paul Ivanier Center for Robotics.

a naive way or according to a pseudo tree [MSTY05,GMZ06]. The present paper adds *Backjumping* to the *AFB* algorithm. Agents in the proposed algorithm use dynamic reasoning in order to choose intelligently the culprit agents to whom they backtrack. The resulting algorithm *AFB-BJ* outperforms both *ADOPT* and *AFB* by a large factor. It performs concurrently both distributed forward bounding and distributed backjumping , pruning the search space of *DisCOP*s very efficiently.

The rest of the paper is divided as follows: Section 2 describes the distributed constraint optimization problem. Section 3 reviews key concepts of asynchronous forward-bounding (*AFB*). Section 4 describes the main ideas of how to incorporate backjumping into *AFB*. Section 5 presents a detailed description of the *AFB-BJ* algorithm. A correctness proof for the new algorithm is given in Section 6. An extensive experimental evaluation of the proposed combined algorithm is in section 7. *AFB-BJ* is compared to *AFB*, which was found to outperform *ADOPT* by a large margin [GMZ06]. The comparison includes the two measures of distributed performance: the number of non concurrent steps of computation, and the total number of messages sent. Our conclusions are presented in section 8.

2 Distributed Constraint Optimization

A distributed constraint optimization problem (*DisCOP*) is composed of a set of **agents** $A_1, ..., A_n$, each holding a set of constrained **variables**. Each variable X_i has a **domain** D_i - a set of possible values. **Constraints** (or relations) exist between variables. Each constraint involves some variables (possibly belonging to different agents), and defines a none-negative **cost** for every possible value combination of these variables. A **binary constraint** is a constraint involving only two variables. An **assignment** (or a label) is a pair including a variable, and a value from that variable's domain. A **partial assignment** (PA) is a set of assignments, in which each variable appears at most once. The **cost of a partial assignment** is computed using all constraints that involve only variables that appear in the partial assignment. Each such constraint defines some cost for the value assignments detailed in the partial assignment. All these costs are accumulated, and the sum is denoted as the cost of the partial assignment. A **full assignment** is a partial assignment that includes all the variables. A **solution** to the *DisCOP* is a full assignment of minimal cost.

A widely used special case of $DisCOP$s is to use a cost of 1 unit for each broken (unsatisfied) constraint. This type of problem is termed *Max-DisCSP*s, in analogy to *Max-CSP*s for the centralized case [LM96,LS04]. In this paper, we will assume that each agent is assigned a single variable, and use the term "agent" and "variable" interchangeably. We will assume that constraints are at most binary and the delay in delivering a message is finite. Furthermore, we assume a static final order on the agents, known to all agents participating in the search process. These assumptions are commonly used in *DisCSP* and *DisCOP* algorithms [Yok00,MSTY05].

3 Asynchronous Forward Bounding

In the *Asynchronous Forward-Bounding* algorithm a single most up-to-date current partial assignment is passed among the agents [GMZ06]. Agents assign their variables only when they hold the up-to-date Current Partial Assignment (*CPA*). The *CPA* is a unique message that is passed between agents, and carries the partial assignment that agents attempt to extend into a complete and optimal solution by assigning their variables on it. The *CPA* also carries the accumulated cost of constraints between all assignments it contains, as well as a unique time-stamp. Only one agent performs an assignment on the *CPA* at any given time. Copies of the *CPA* are sent forward (on FB_CPA messages) and are concurrently processed by multiple agents. Each unassigned agent computes a lower bound on the cost of assigning a value to its variable, and sends this bound back to the agent which performed the assignment (on $FB\_ESTIMATE$ messages) [GMZ06]. The assigning agent uses these bounds to prune sub-spaces of the search-space which do not contain a full assignment with a cost lower than the best full assignment found so far. Thus, asynchronous forward bounding enables agents an early detection of partial assignments that can not be extended into complete assignments with cost smaller than the known upper bound. As a result, agents initiate backtracks as early as possible [GMZ06].

The lower bound is computed as follows: Denote by $\mathbf{cost}((i,v),(j,u))$ the cost of assigning $A_i = v$ and $A_j = u$. For each agent A_i and each value in its domain $v \in D_i$, denote the minimal cost of the assignment (i,v) incurred by an agent A_j by $h_j(v) = min_{u \in D_j}(cost((i,v),(j,u)))$. Define $h(v)$ to be the sum of $h_j(v)$ over all $j > i$. Intuitively, $h(v)$ is a lower bound on the cost of constraints involving the assignment $A_i = v$ and all agents A_j such that $j > i$.

An agent A_i, which receives an FB_CPA message, can compute for every $v \in D_i$ both the cost increment of assigning v as its value, i.e. the sum of the cost of conflicts that v has with the assignments included in the CPA, and $h(v)$. The sum of these, is denoted by $f(v)$. The minimal $f(v)$ over all values $v \in D_i$ is chosen to be the lower bound estimation of agent A_i.

Figure 1 presents a constraint network, in which A_1 already assigned the value v_1 and A_2, A_3, A_4 are unassigned. Let us assume that the cost of every constraint is one. The cost of v_3 will increase by one due to its constraint with the current assignment thus $f(v_3) = 1$. Since v_4 is constrained with both v_8 and v_9, assigning this value will trigger a cost increment when A_4 performs an assignment. Therefore, $h(v_4) = 1$ is an admissible lower bound of the cost of the constraints between this value (i.e. $A_1 = v_1$) and lower priority agents. Since v_4 does not conflict with assignments on the CPA, $f(v_4) = 1$ as well. $f(v_5) = 3$ because this assignment conflicts with the assignment on the CPA and in addition conflicts with all the values of the two remaining agents.

4 Adding Backjumping - Key Ideas

In both centralized and distributed *CSP*s backjumping can be accomplished by maintaining some data structures that will allow an agent to deduce who is the latest agent (in the order in which assignments were made) whose changed assignment could possibly

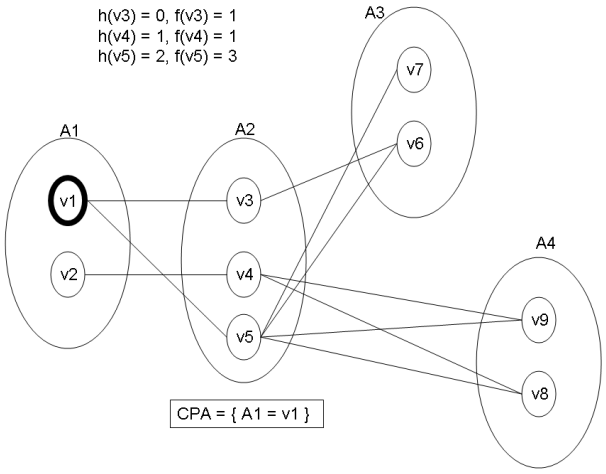


Fig. 1. A simple DisCOP, demonstration

lead to a solution. Once such an agent is found, the assignments of all following agents are unmade and the search process "backjumps" to that agent.

A similar process can be designed for branch and bound based solvers for *COP*s and *DisCOP*s. Consider a sequence of assignments by the agents A_1, A_2, A_3, A_4, A_5 where A_5 determined that none of its possible value assignments can lead to a full assignment with a cost lower than the cost of the best full assignment found so far. Clearly, A_5 must backtrack.

In chronological backtracking, the search process would simply return to the previous agent, namely A_4, and have it change its assignment. However, A_5 can sometimes determine that no value change of A_4 would suffice to reach a lower cost full assignment. Intuitively, A_5 can safely backjump to A_3, if it can compute a lower bound on the cost of a full assignment extended from the assignments of A_1, A_2 and A_3, and show that this bound is greater or equal to the cost of the best full assignment found so far. This is the intuitive basis of how backjumping can be added to *AFB*.

More formally, let us consider a scenario in which A_i decides to backtrack, and the cost of the best full assignment found so far is B. The current partial assignment includes the assignments of agents $A_1, ..., A_{i-1}$. We define:

- *Definition:* CPA[1..k] is the set of assignments made by agents $A_1, \ldots, A_k$ in the current partial assignment. We define $CPA[1..0] = \{\}$.
- *Definition:* FA[k] is the set of all full assignments, which include all the assignments appearing in CPA[1..k]. For example, FA[2] contains all full assignments in which both A_1 and A_2 have the same value assignments as they do in the current partial assignment. Naturally, FA[0] is the set of all full assignments.

On a backtrack, instead of simply backtracking to the previous agent, A_i performs the following actions: It computes a lower bound on the cost of any full assignment in FA[i-2]. If this bound is smaller than B, it backtracks to A_{i-1} just like it would do

in chronological backtracking. However if this bound is greater or equal to B, then backtracking to A_{i-1} would do little good, as no value change of A_{i-1} alone could result in a full assignment of cost lower than B. And so A_i knows it can safely backjump to A_{i-2}. It may be possible for A_i to backjump even further, depending on the lower bound on the cost of any full assignment in FA[i-3]. If this bound is smaller than B, it backjumps to A_{i-2}. Otherwise, it knows it can safely backjump to A_{i-3}. Similar checks can be made about the necessity to backjump even further.

The backjumping procedure relies on the computation of lower bounds for sets of full assignments (FA[k]). Next, we will show how can A_i compute such lower bounds. Let us define the notions of past,local and future costs:

- *Definition:* PC (Past-Costs) is a vector of size n+1, in which the k-th element ($0 \leq k \leq n$) is equal to the cost of CPA[1..k].
- *Definition:* LC(v) (Local-Costs) is a vector of size $n+1$, in which the k-th element ($0 \leq k \leq n$) is

$$LC(v)[k] = \sum_{(A_j,v_j) \in CPA \ s.t \ j \leq k} cost(A_i = v, A_j = v_j)$$

  Since the CPA held by A_i only includes assignments of $A_1, \ldots, A_{i-1}$, then $\forall j \geq i$ LC(v)[i-1] = LC(v)[j]. Intuitively, LC(v)[k] is the accumulated cost of the value v of A_i, with respect to all assignments in CPA[1..k].
- *Definition:* FC_j(v) (Future-Costs) is a vector of size n+1, in which the k-th element ($0 \leq k \leq n$) contains a lower bound on the cost of assigning a value to A_j with respect to the partial assignment CPA[1..k]. If $k \geq i$ then CPA[1..k] contains the assignment $Ai = v$, but for $k < i$ the value v of A_i is irrelevant as it does not appear in CPA[1..k].

The above vectors provide additive lower bounds on full assignments that start with the current CPA up to k, FA[k]. PC[k] is the exact cost of the first k assignments, LC(v)[k] is the exact cost of the assignment $A_i = v$, and $\sum_{j>i} FC_j(v)[k]$ is a lower bound on the assignments of $A_{i+1}, ..., A_n$. Therefore, the sum

$$FALB(v)[k] = LC(v)[k] + PC[k] + \sum_{j>i} FC_j(v)[k]$$

is a Full Assignment Lower Bound on the cost of any full assignment extended from CPA[1..k] in which $A_i = v$.

FA[k] contains all full assignments extended from CPA[1..k], and is not limited to assignments in which $A_i = v$. If we go over all FALB(v)[k], for all possible values $v \in D_i$ we produce a lower bound on any assignment in FA[k].

- *Definition:* $FALB[k] = \min_{v \in D_i}(FALB(v)[k])$.
  FALB[k] is a lower bound on the cost of any full assignment extended from CPA[1..k].

In a distributed branch and bound algorithm, this bound can be computed by A_i. PC - the cost of previous agents can be sent along with their value assignment messages

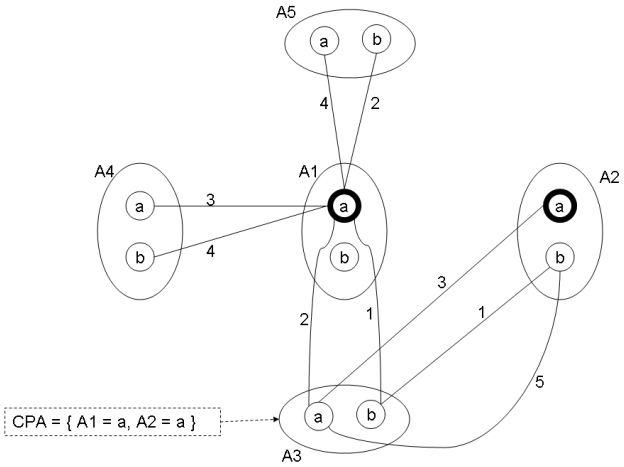


Fig. 2. A DisCOP example

to A_i. LC(v) - the cost of assigning v to A_i can be computed by A_i itself, and A_i can request all agents ordered after it, A_j ($j > i$), to compute FC$_j$ and send the result back to A_i.

In the *AFB* algorithm [GMZ06] A_i already requests unassigned agents to compute lower bounds on the CPA and send back the results. The bounds needed for backjumping can be easily added to the existing *AFB* framework.

4.1 A Backjumping Example

To demonstrate the backjumping possibility, consider the *DisCOP* in Figure 2. Let us assume that the search begins with A_1 assigning "a" as its value and sending the CPA forward to A_2. A_2, A_3, A_4, and A_5 all assign the value "a" and we get a full assignment with cost 12. The search continues, and after fully exploring the sub-space in which $A_1 = a, A_2 = a$, the best assignment found is $A_1 = a, A_2 = a, A_3 = b, A_4 = a, A_5 = b$ with a total cost of B=6. Assume that A_3 is now holding the CPA after receiving it from some future agent (A_4 or A_5). A_3 has exhausted its value domain and must backtrack. It computes:

$$FALB(a)[1] = PC[1] + LC(a)[1] + (FC_4(a)[1] + FC_5(a)[1])$$

$$= 0 + 2 + (3 + 2) = 7$$

$$FALB(b)[1] = PC[1] + LC(b)[1] + (FC_4(b)[1] + FC_5(b)[1])$$

$$= 0 + 1 + (3 + 2) = 6$$

$$FALB[1] = min(FALB(a)[1], FLAB(b)[1]) = 6$$

$FALB[1] \geq B$, therefore A_3 knows that any full assignment extended from $\{A_1 = a\}$ would cost at least 6. A full assignment with that cost was already discovered, so there is no need to explore the rest of this sub-space, and it can safely backjump the search process back to A_1, to change its value to "b". Backtracking to A_2 would still leave us within the $\{A_1 = a\}$ sub-space, which A_3 knows cannot lead to a full assignment with a lower cost.

5 The AFB-BJ Algorithm

The *AFB-BJ* algorithm is run on each of the agents in the *DisCOP*. Each agent first calls the procedure *init* and then responds to messages until it receives a TERMINATE message. The algorithm is presented in Figure 3. As in pure *AFB*, a timestamping mechanism is used on all messages.

Timestamping is used to determine which messages are relevant and which are obsolete. For simplicity, we choose to omit the description of this mechanism from the pseudo-code, referring the user to the above former references. For the same reason we choose to omit the pseudo-code detailing the calculation of LC,PC,FC and FALB.

5.1 Algorithm description

The algorithm starts by each agent calling **init** and then awaiting messages until termination. At first, each agent updates B to be the cost of the best full assignment found so far and since no such assignment was found, it is set to infinity (line 1). Only the first agent (A_1) creates an empty CPA and then begins the search process by calling **assign_CPA** (lines 3-4), in order to find a value assignment for its variable.

An agent receiving a CPA (when received **CPA_MSG**), checks the time-stamp associated with it. An out of date CPA is discarded. When the message is not discarded, the agent saves the received PA in its local CPA variable (line 7). In case the CPA was received from a higher priority agent, the estimations of future agents in FC_j are no longer relevant and are discarded, and the domain values must be reordered by their updated cost (lines 9-11). Then, the agent attempts to assign its next value by calling **assign_CPA** (line 16) or to backtrack if needed (line 14).

Procedure **assign_CPA** attempts to find a value assignment, for the current agent. The assigned value must be such that the sum of the cost of the CPA and the lower bound of the cost increment caused by the assignment will not exceed the upper bound B (lines 23). If no such value is found, then the assignment of some higher priority agent must be altered, so backtrack is called (line 25). When a full assignment is found which is better than the best full assignment known so far, it is broadcast to all agents (line 29). After succeeding to assign a value, the CPA is sent forward to the next unassigned agent (line 33). Concurrently, forward bounding requests (i.e. FB_CPA messages) are sent to all lower priority agents (lines 34-35).

An agent receiving a bound estimation (when received **FB_ESTIMATE**) from a lower priority agent A_j (in response to a forward bounding message) ignores it if it is an estimate to an already abandoned partial assignment (identified using the time-stamp mechanism). Otherwise, it saves this estimate (line 17) and checks if this new estimate

```
procedure init:
1.  B ← ∞
2.  if (A_i = A_1)
3.      generate_CPA()
4.      assign_CPA()

when received (FB_CPA, A_j, PA)
5.  V ← estimation vector
          for each PA[1..k] (0 ≤ k ≤ n)
6.  send (FB_ESTIMATE, V, PA, A_i) to A_j

when received (CPA_MSG, PA, A_j)
7.  CPA ← PA
8.  TempCPA ← PA
9.  if (j = i − 1)
10.     ∀j re-initialize FC_j(v)
11.     reorder domain values v ∈ D_i by LC(v)[i]
              (from low to high)
12. if (TempCPA contains an assignment to A_i)
              remove it
13. if (TempCPA.cost ≥ B)
14.     backtrack()
15. else
16.     assign_CPA()

when received (FB_ESTIMATE, V, PA , A_j)
17. FC_j(v) ← V
18. if ( FALB(v)[i] ≥ B )
19.     assign_CPA()

when received (NEW_SOLUTION, PA)
20. B_CPA ← PA
21. B ← PA.cost

procedure assign_CPA:
22. if CPA contains an assignment
          A_i = w, remove it
23. iterate (from last assigned value)
          over D_i until found
      v ∈ D_i s.t. CPA.cost + f(v) < B
24. if no such value exists
25.     backtrack()
26. else
27.     assign A_i = v
28.     if CPA is a full assignment
29.         broadcast (NEW_SOLUTION, CPA )
30.         B ← CPA.cost
31.         assign_CPA()
32.     else
33.         send(CPA_MSG, CPA, A_i) to A_{i+1}
34.         forall j > i
35.         send(FB_CPA, A_i, CPA) to A_j

procedure backtrack:
36. if (A_i = A_1)
37.     broadcast(TERMINATE)
38. else
39.     j ← backtrackTo()
40.     remove assignments of A_{j+1}, .., A_i from CPA
41.     send(CPA_MSG, CPA, A_i) to A_j

function backtrackTo:
42. for j = i − 1 downto 1
43.     foreach v ∈ D_i
44.         if ( FALB(v)[j-1] + (PC[j] - PC[j-1]) < B )
45.             return j
46. broadcast(TERMINATE)
```

Fig. 3. The procedures of the AFB-BJ Algorithm

causes the current partial assignment to exceed the bound B (line 18). In such a case, the agent calls $assign\_CPA$ (line 19) in order to change its value assignment (or backtrack in case a valid assignment cannot be found).

The call to **backtrack** is made whenever the current agent cannot find a valid value (i.e. below the bound B). In such a case, the agent calls backtrackTo() to compute to which agent the CPA should be sent, and backtracks the search process (by sending the CPA) back to that agent. If the agent is the first agent (nowhere to backtrack to), the terminate broadcast ends the search process in all agents (line 37). The algorithm then reports that the optimal solution has a cost of B, and the full assignment corresponding to this cost is B_CPA.

The function **backtrackTo** computes to which agent the CPA should be sent. This is the kernel of the backjumping (BJ) mechanism. It goes over all candidates, from $j-1$ downto 1, looking for the first agent it finds that has a chance of reaching a full

assignment with a lower cost than B. FALB(v)[j-1] is a lower bound on the cost of a full assignment extended from CPA[1..j-1], and PC[j]-PC[j-1] is the cost added to that CPA by A_j's assignment. Since A_j picked the lowest cost value in its domain (its domain was ordered in line 11), the addition of these two components produces a more accurate lower bound on the cost of a full assignment extended from CPA[1..j-1]. If this bound is not smaller than B, then surely any combination of assignments made by A_j and any following agent could only raise the cost, which is already too high. In case even backjumping back to A_1 will not prove helpful, the search process is terminated (line 46).

6 Correctness of AFB-BJ

In order to prove the correctness of the *AFB_BJ* algorithm we first prove the correctness of the proposed Backjumping method and then show that its combination with *AFB* does not violate *AFB*'s correctness as proven in [GMZ06].

In order to prove the correctness of the backjumping method one need only show that none of the agents' assignments that the algorithm backjumps over, can lead to a solution with a lower cost than the current upper bound. The condition for performing backjumping over A_j (line 44) is that the lower bound on the cost of a full assignment extended from the assignments of $A_1, .., A_{j-1}$ and of the assignment cost of A_j exceeds the global upper bound B. Since A_j picked the lowest cost value in its remaining domain (as the domain is ordered), extending the assignments of $A_1, .., A_{j-1}$ must lead to a cost greater or equal to B. Therefore, backjumping back to A_{j-1} cannot discard any potentially lower cost solutions. This completes the correctness proof of the $AFB-BJ$ backjumping (function **backtrackTo**) method.

Assuming the correctness of *AFB* as proven in [GMZ06], in order to prove the correctness of the composite algorithm *AFB-BJ* it is enough to prove the consistency of the lower bounds computed by the agents in *AFB-BJ*. The lower bounds computed by *AFB-BJ* include FC, LC and PC as described in Section 4. PC is contained in the CPA, and is updated by any agent that receives it and adds an assignment (not shown in the code). LC(v) is computed by the current agent A_i whenever it assigns v as its value assignment. FC_j is computed by A_j in line 5, and is send back to A_i in line 6. A_i receives and saves this in line 17. The lower bounds contained inside these vectors are correct as PC was exactly calculated when holding the CPA, LC was exactly calculated by the current agent A_i, and the bounds in FC_j are the same bounds computed in *AFB* which were proven to be correct lower bounds for the assignment of A_j in [GMZ06]. The FC_j bounds are accurate and based on the current partial assignment since the timestamp mechanism prevents processing of bounds which are based on an obsolete CPA. Whenever the CPA is altered by some higher priority agent, the previous bounds are cleared (line 10).

7 Experimental Evaluation

All experiments were performed on a simulator in which agents are simulated by threads which communicate only through message passing. The Distributed Optimization prob-

lems used in all of the presented experiments are random *Max-DisCSPs*. *Max-DisCSP* is a subclass of *DisCOP* in which all constraint costs (weights) are equal to one [MSTY05]. 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 (a non zero cost) 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. [Pro96]). *Max-CSP*s are commonly used in experimental evaluations of constraint optimization problems (*COP*s) [LS04]. Other experimental evaluations of DisCOPs include graph coloring problems [MSTY05,ZXWW05], which are a subclass of *Max-DisCSP*.

In order to evaluate the performance of distributed algorithms, two independent measures of performance are commonly used - run time, in the form of non-concurrent steps of computation, and communication load, in the form of the total number of messages sent [Lyn97,Yok00]. We use the method described in [MRKZ02] for counting non-concurrent computational steps.

Our experiments include the ADOPT algorithm and three versions of the *AFB* algorithm: *AFB*, *AFB-minC* - a variation of *AFB* which includes dynamic ordering of values based on minimal cost (with the current CPA), and *AFB-BJ* which is the composite backjumping and forward-bounding algorithm. *AFB-BJ* includes the same value ordering heuristic as *AFB-minC*. This was selected in order to show that the improved performance of *AFB-BJ* does indeed come from the backjumping feature and not from the value ordering heuristic.

Figure 4 presents the average run-time in number of non-concurrent computation steps, of the all algorithms: *Adopt*, *AFB*, *AFB-minC* and *AFB-BJ*, on *Max-DisCSP*s with with $n = 10$ agents, domain size $k = 10$, and a high constraint density of $p_1 = 0.7$. In accordance with former findings, asynchronous optimization (*ADOPT*) is much slower than the standard version of *AFB* [GMZ06]. Also clear from this figure, is that the value ordering heuristic greatly improves *AFB*'s performance. The added backjumping improves the performance much further. The RHS of the figure provides a "zoom in" on the section of the graph between $p_2 = 0.9$ and $p_2 = 0.98$.

For tightness values that are higher than 0.9 *AFB* demonstrates a "phase transition", that was reported in [GMZ06]. *AFB-minC* as well as *AFB-BJ* display the same phenomenon. This "phase transition" behavior of the *AFB* algorithms is very similar to that of lookahead algorithms on centralized *Max-CSP*s [LM96,LS04].

Figure 5 presents the total number of messages sent by each of the algorithms. The results of this measurement closely match the results of run-time, as measured by non-concurrent steps.

8 Conclusions

The present paper extends Asynchronous Forward-Bounding (*AFB*) by adding to it a *backjumping mechanism*. The resulting algorithm, *AFB-BJ*, performs both backjumping and asynchronous forward-bounding to solve Distributed Constraint Optimization Problems (*DisCOP*s).

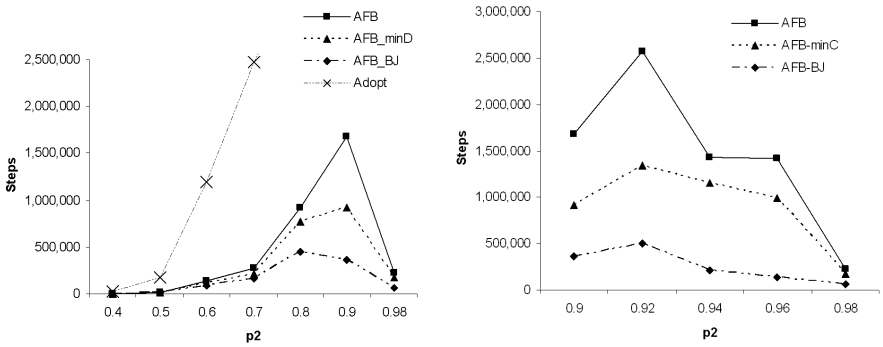


Fig. 4. (a) Number of none-concurrent steps performed by AFB, AFB-minC and AFB-BJ for high density MaxDisCSP ($p_1 = 0.7$). (b) A closer look at $p_2 > 0.9$

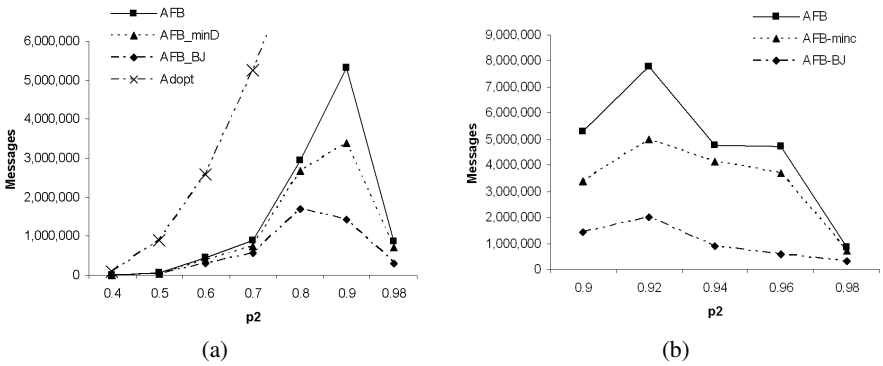


Fig. 5. (a) Number of messages sent by AFB, AFB-minC and AFB-BJ for high density MaxDisCSP ($p_1 = 0.7$). (b) A closer look at $p_2 > 0.9$

In *AFB-BJ*, extended consistency maintenance procedures are performed concurrently to compute lower bounds on partial assignments. These bounds are used both to prune sub-spaces of the search space, and to enable backjumping when a sub-space has been fully explored or pruned.

Backjuming was proposed before for *CSP*s, *DisCSP*s, and *COP*s but, to the best of our knowledge, this is the first *DisCOP* solver to implement backjumping.

The combination of the proposed backjumping method with *AFB* enables several agents which request forward bounding from future agents, to detect the need to backtrack to some higher priority agent. Due to the asynchronous nature of the algorithm, it is possible that one of the agents will receive enough bounds to detect the need to backjump before the other agents, and it will perform the backjump. This creates a form of "open race" between all agents. Even if one of the agents is suffering from delayed replies, others may detect and initiate the backjump sooner.

The results presented in [GMZ06] demonstrate the importance of consistency maintenance for distributed search, and show a phase-transition in the performance of all versions of *AFB* as the problems become increasingly tighter. In contrast, asynchronous optimization (*ADOPT*) does not produce this phase-transition, but rather grows exponentially fast in both run time and network load.

The present paper introduced *AFB* with value ordering, and experimentally demonstrated that this version of *AFB* outperforms standard *AFB*. When adding the backjumping feature, the performance of the algorithm improves further, and the resulting *AFB-BJ* vastly outperforms *AFB* with value ordering. This proves that the main improvement in the performance of *AFB-BJ* is due to backjumping and no due to the value ordering heuristic used.

References

[GMZ06] A. Gershman, A. Meisels, and R. Zivan. Asynchronous forward-bounding for distributed constraints optimization. In *Proc. ECAI-06*, pages 103–108, Riva Del Garda, Italy, August 2006.

[KM05] E. Kaplansky and A. Meisels. Distributed personnel scheduling: Negotiation among scheduling agents. *Annals of Operations Research*, 2005.

[LM96] J. Larrosa and P. Meseguer. Phase transition in max-csp. In *Proc. ECAI-96*, Budapest, 1996.

[LS04] J. Larrosa and T. Schiex. Solving weighted csp by maintaining arc consistency. *Artificial Intelligence*, 159:1–26, 2004.

[Lyn97] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Series, 1997.

[MRKZ02] A. Meisels, I. Razgon, E. Kaplansky, and R. Zivan. Comparing performance of distributed constraints processing algorithms. In *Proc. AAMAS-2002 Workshop on Distributed Constraint Reasoning DCR*, pages 86–93, Bologna, July 2002.

[MSTY05] P. J. Modi, W. Shen, M. Tambe, and M. Yokoo. Adopt: asynchronous distributed constraints optimization with quality guarantees. *Artificial Intelligence*, 161:1-2:149–180, January 2005.

[PF04] A. Petcu and B. Faltings. A value ordering heuristic for distributed resource allocation. In *Proc. CSCLP04, Lausanne, Switzerland*, http://liawww.epfl.ch/Publications/Archive/Petcu2004.pdf, 2004.

[Pro96] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.

[SGM96] G. Solotorevsky, E. Gudes, and A. Meisels. Modeling and solving distributed constraint satisfaction problems (dcsps). In *Constraint Processing-96*, pages 561–2, New Hamphshire, October 1996.

[Yok00] M. Yokoo. Algorithms for distributed constraint satisfaction problems: A review. *Autonomous Agents & Multi-Agent Sys.*, 3:198–212, 2000.

[ZXWW05] W. Zhang, Z. Xing, G. Wang, and L. Wittenburg. Distributed stochastic search and distributed breakout: properties, comparishon and applications to constraints optimization problems in sensor networks. *Artificial Intelligence*, 161:1-2:55–88, January 2005.

H-DPOP: Using Hard Constraints to Prune the Search Space

Akshat Kumar, Adrian Petcu and Boi Faltings

Ecole Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne (Switzerland)
akshat.kumar@gmail.com, {adrian.petcu, boi.faltings}@epfl.ch

Abstract. In distributed constraint optimization problems, dynamic programming methods have been recently introduced (e.g. DPOP [9]). These methods group many valuations together in fewer (and also larger) messages, as opposed to sending them individually. Therefore, they have the important advantage that they produce small communication overheads. However, the maximum message size is always exponential in the induced width of the constraint graph, leading to excessive memory/communication requirements for problems with large width.
Many real problems contain hard constraints that significantly reduce the space of feasible assignments. The basic DPOP does not take advantage of the pruning power of these hard constraints; thus, it sends messages that explicitely represent all value combinations, including many infeasible ones.
To address this problem, we introduce H-DPOP, a hybrid algorithm that is based on DPOP. H-DPOP uses Constraint Decision Diagrams (CDDs, see [2]) to rule out infeasible combinations, and thus compactly represent UTIL messages. For highly constrained problems, CDDs prove to be extremely space-efficient when compared to the extensional representation used by DPOP: experimental results show space reductions of more than 99% for some instances.

1 Introduction

Constraint satisfaction and optimization are powerful paradigms that can model a wide range of tasks like scheduling, planning, optimal process control, etc. Traditionally, such were gathered into a single place, and a centralized algorithm was applied to find a solution. However, problems are sometimes naturally distributed, so Distributed Constraint Satisfaction (DisCSP) was formalized in [12]. Here, the problem is divided between a set of agents, which have to communicate among themselves to solve it.

To address distributed optimization, complete algorithms like ADOPT, DPOP and OptAPO have been introduced.

ADOPT [8] is a backtracking based bound propagation mechanism. It operates completely decentralized, and asynchronously. It requires polynomial memory. Its downside is that it may produce a very large number of small messages, resulting in large communication overheads.

OptAPO [6] is a centralized-distributed hybrid that uses *mediator nodes* to centralize subproblems and solve them in dynamic and asynchronous mediation sessions. The authors show that its message complexity is significantly smaller than ADOPT's. However, it is possible that several mediators solve overlapping problems, thus needlessly duplicating effort.

DPOP [9] is a complete algorithm based on dynamic programming which generates only a linear number of messages. However, *DPOP* is time and space exponential in the *induced width* of the problem. Therefore, for problems with high induced width, the messages generated in high-width areas get large, therefore requiring exponential communication and memory.

We present H-DPOP, a new hybrid algorithm which uses hard constraint propagation to prune the search space and compactly represent the resulting messages with Constraint Decision Diagrams (CDDs - [2]).

The rest of this paper is structured as follows: Section 2 introduces the distributed optimization problem. Section 3 presents some background on the *DPOP* algorithm (section 3.1), and on constraint decision diagrams (section 3.2). Section 4 introduces the H-DPOP algorithm.

Section 5 shows the efficiency of this approach with experimental results. Section 7 concludes.

2 Definitions & problem modeling

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

- $\mathcal{X} = \{X_1, ..., X_n\}$ *is a set of variables*
- $\mathcal{D} = \{d_1, ..., d_n\}$ *is a set of finite variable domains*
- $\mathcal{C} = \{c_1, ..., c_n\}$ *is a set of constraints, where a constraint* c_i *is a function with the scope* $(X_{i_1}, \cdots, X_{i_k})$*,* $c_i : d_{i_1} \times .. \times d_{i_k} \rightarrow \{true, false\}$*, where* `true` *corresponds to allowed tuples, and* `false` *corresponds to disallowed tuples specifying hard constraint.*
- $\mathcal{R} = \{r_1, ..., r_m\}$ *is a set of relations, where a relation* r_i *is any function with the scope* $(X_{i_1}, \cdots, X_{i_k})$*,* $r_i : d_{i_1} \times .. \times d_{i_k} \rightarrow \mathbb{R}$*, which denotes how much utility is assigned to each possible combination of values of the involved variables. Negative utilities mean cost.*

In a DCOP, each variable and constraint is owned by an agent. A simplifying assumption [12] is that each agent controls a virtual agent for each one of the variables X_i that it owns. To simplify the notation, we use X_i to denote either the variable itself, or its (virtual) agent.

This is a multiagent instance of the *valued CSP* framework as defined in [11]. The goal is to find a complete instantiation $\mathcal{X}^*$ for the variables X_i that *maximizes* the sum of utilities of individual relations.

We assume here only unary and binary constraints/relations. However, DPOP and H-DPOP can easily extend to non-binary constraints ([10]).

2.1 Using hard constraints to prune the search space

Without loss of generality, hard constraints can be simulated using soft constraints by assigning large negative numbers to disallowed tuples, and 0 to allowed tuples. Then,

simply using any utility maximization algorithm on such a representation avoids infeasible assignments and finds the optimal solution.

However, by doing so one does not take advantage of the pruning power of hard constraints. This drawback becomes severe for difficult problems (high width). We introduce below one such real world problem and show the space reduction ability of hard constraints.

Optimal query placement Consider the problem of optimally placing a set of query operators in an overlay network. Each user wants a set of services to be performed by servers in the network. Servers are able to perform services with distinct network and computational characteristics. Each server receives hosting requests from its users (together with the associated utilities). We model the resulting DCOP with servers as variables (agents) and the possible service combinations as the domains.

To avoid accounting the utility from the same service being placed simultaneously on two servers we introduce hard constraints between server pairs. These constraints disallow the same service to be executed by two servers at a time. Although this constraint is simple, it makes the problem highly constrained and computationally difficult.

Note that the above model may not be an exactly equivalent model for optimal query placement but it helps to make the problem tractable. The optimal solution may include running a service on more than one server but the problem would become much more complex in its originality. Hard constraints offer a convenient way to simplify problem modeling and keep it computationally tractable.

Figure 2(a) shows a DFS tree arrangement for servers in an overlay network. The services each server can execute are listed adjacent to nodes. During the utility propagation phase of DPOP node X_4 will send a hypercube with X_1, X_2 and X_3 as context variables to its parent X_3 (see figure 2(b)). However such a message scheme will send combinations which will never appear in a valid solution. For e.g. combinations like $< (X_1, a)(X_2, a)(X_3, b) >$ which share a common service are infeasible. The total size of this hypercube will be 64 (4^3) with only 24 ($4!$) valid combinations. Eliminating these combinations using hard constraints can provide significant savings.

Consider an instance of server problem with 9 variables (servers) with the same domain of size 9. The resulting network will be a chain with constraints between every server pair. The maximum size of hypercube in DPOP will be 9^9 and the number of valid combinations will be only $9!$. So we are wasting 99.9% of the space in the message by sending irrelevant combinations. With the help of hard constraints we can prune such infeasible combinations and get extreme savings.

2.2 Depth-First Search Trees (DFS)

As with DPOP, H-DPOP works on a DFS traversal of the problem graph.

Definition 2 (DFS tree). *A DFS arrangement of a graph G is a rooted tree with the same nodes and edges as G and the property that adjacent nodes from the original graph fall in the same branch of the tree (e.g. X_0 and X_{11} in Figure 1).*

DFS trees have already been investigated as a means to boost search [5, 4]. Due to the relative independence of nodes lying in different branches of the DFS tree, it is possible to perform search in parallel on independent branches, and then combine the results.

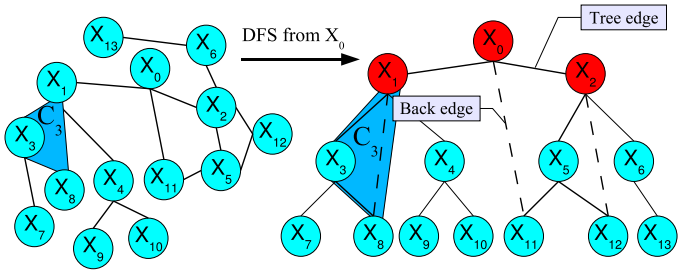


Fig. 1. *A problem graph (a) and a DFS tree (b).*

Figure 1 shows an example DFS tree that we shall refer to in the rest of this paper. We distinguish between *tree edges*, shown as solid lines (e.g. $0-2$), and *back edges*, shown as dashed lines (e.g. $0-11$, $2-12$).

Definition 3 (DFS concepts). *Given a node x_i, we define:*

- **parent** P_i **/ children** C_i*: these are the obvious definitions (e.g.* $P_1 = x_0$, $C_0 = \{x_1, x_2\}$*).*
- **pseudo-parents** PP_i *are x_i's ancestors that are connected to x_i directly through back-edges (*$PP_8 = \{x_1\}$*).*
- **pseudo-children** PC_i *are x_i's descendents directly connected to x_i through back-edges (e.g.* $PC_1 = \{x_8\}$*).*
- Sep_i *is the* **separator** *of x_i: ancestors of x_i which are directly connected with x_i or with descendants of x_i (e.g.* $Sep_{11} = \{x_5, x_0\}$*). Sep_i is the set of ancestors of X_i whose removal completely disconnects the subtree rooted at X_i from the rest of the problem. In case the problem is a tree, then* $Sep_i = \{P_i\}, \forall X_i \in \mathcal{X}$*.*

3 Background: DPOP and CDDs

This section introduces the DPOP algorithm for distributed constraint optimization (Section 3.1) and the constraint decision diagrams of [2] in Section 3.2.

3.1 DPOP: dynamic programming optimization

The basic utility propagation scheme *DPOP* has been introduced in [9]. *DPOP* is an instance of the general bucket elimination scheme from [3], which is adapted for the distributed case, and uses a DFS traversal of the problem graph as an ordering.

DPOP has 3 phases:

Phase 1 - a **DFS traversal** of the graph is done using a distributed DFS algorithm. To save space, we refer the reader to an algorithm like [10]. The outcome of this protocol is that all nodes consistently label each other as parent/child or pseudoparent/pseudochild, and edges are identified as tree/back edges. This can be achieved for any graph with a linear number of messages. The DFS tree thus obtained serves as a communication structure for the other 2 phases of the algorithm: UTIL messages (phase 2) travel bottom-up, and VALUE messages (phase 3) travel top down, only through tree-edges.

Phase 2 - **UTIL propagation**: the agents (starting from the leaves) send *UTIL* messages to their parents (called *hypercubes* in the following). The basic process is as follows: the leaves start by computing and sending *UTIL* messages to their parents. Subsequently, all nodes do:

- receive and join all messages from their children
- join also the relations they have with their parents/pseudoparents
- project themselves out of the resulting join, by picking their optimal values for each combination of values of the other variables in the join
- send result to parent as a new UTIL message.

The subtree of a node X_i can influence the rest of the problem only through X_i's separator, Sep_i. Therefore, a message contains the optimal utility obtained in the subtree for each instantiation of Sep_i. Each such utility is represented explicitely as a valuation in the outgoing message, even though the corresponding instantiation of Sep_i may not lead to any feasible solution for X_i's subtree. Thus, message size is always exponential in the separator size (which in turn is bounded by the induced width).

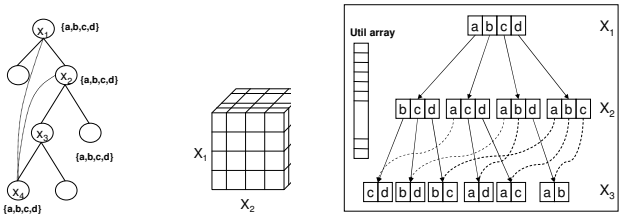
Phase 3 - **VALUE propagation** top-down, initiated by the root, when phase 2 has finished. Each node determines its optimal value based on the computation from phase 2 and the *VALUE* message it has received from its parent. Then, it sends this value to its children through *VALUE* messages.

It has been proven in [9] that *DPOP* produces a linear number of messages. Its complexity lies in the size of the *UTIL* messages: the largest one is space-exponential in the width of the DFS ordering used.

3.2 CDDs: Constraint Decision Diagrams

CDDs (constrained decision diagrams) [2] are compact representations for general n-ary constraints. They generalize binary decision diagrams (BDD) [1]. Their main feature is that they combine constraint reasoning and consistency techniques with a compact data structure. Unlike extensional representations that store each individual tuple separately (therefore requiring memory exponential in the arity of the constraint), CDDs have the potential to drastically reduce space requirements.

A CDD is a rooted, directed acyclic graph (DAG) $G = (V \cup T, E)$. The $0 - terminal$ $(0 \in T)$ represents $false$ and $1 - terminal$ $(1 \in T)$ represents $true$. Each non terminal node $v \in V$ connects to a subset of nodes $U \subseteq V \cup T - \{v\}$. It is denoted by a non-empty set $\{(c_1, u_1), ..., (c_m, u_m)\}$. Each branch (c_j, u_j) consists of a constraint $c_j(x_1, ..., x_k)$ and a successor u_j of v.



(a) DFS Tree (b) Hypercu (c) CDD Message

Fig. 2. A DFS tree network with CDD equivalent of hypercube sent from X_4 to X_3

A CDD rooted at the node $v = \{(c_1, u_1), ..., (c_m, u_m)\}$ is $reduced$ if and only if each CDD graph G'_j rooted at u_j is either terminal or reduced, and

$$c_i \wedge c_j \equiv false \tag{1}$$

$$u_i \not\equiv u_j \tag{2}$$

We show in Figure 2(c) an example CDD that represents compactly a ternary constraint (C) between X_1, X_2 and X_3 with domain (D_i) listed adjacent. The constraint C requires the variables to take unique values. Each CDD node is of the form $\{(x_k \in r_1, u_1), ..., (x_k \in r_m, u_m)\}$ where $r_1, ..., r_2 \subset D_k$ are pairwise disjoint to satisfy Property (1) of a reduced CDD. Property (2) allows node sharing marked by dashed lines in Figure 2(c).

4 H-DPOP - Pruning problem space

The pruning abilities of hard constraints were hinted at in section 2.1. The H-DPOP algorithm combines this pruning ability with CDDs to effectively reduce message size. Figure 2(c) shows the corresponding CDD message for the hypercube X_4 sends to X_3 in Figure 2(a). In a CDD, every path from root to leaf is a valid combination of domain values of the involved variables. The explicit representation of domain values and an insight into the problem nature allows us to prune combinations like $< (X_1, a)(X_2, a)(X_3, b) >$ for service placement problems even at the bottommost level.

Utilities in a CDD message are represented by a linear array storing utility values indexed by path numbers. Each path in CDD can be assigned a unique index obtained by a DFS traversal of the CDD tree. Additionally we also need to send the domain values of each involved variable in a CDD message. This step is necessary to ensure pruning at higher levels which is based on examining combinations of actual domain values.

Definition 4. $Dim(X)$ *- the tuple* $< id_x, d_x >$ *where* id_x *is the id of node* X *and* d_x *is its domain.*

We now describe the **CDDMessage** which node X will send to $P(X)$. It is composed of three components:

– *CDDTree* : It represents all valid combinations of variables involved in the message. Each level in *CDDTree* corresponds to one variable.
– *UtilArray*: It is the array of all utilities corresponding to each path in *CDDTree.*
– *DimensionArray* : It is an array containing $Dim(X_i)$ where $X_i in$ {variables involved in message}.

As in the DPOP algorithm, we have three phases in H-DPOP: DFS arrangement of the problem graph, bottom up UTIL propagation and top down VALUE propagation. The DFS and VALUE phases are identical to the ones of DPOP (see section 3.1). The UTIL propagation phase is described below, in Sectio 4.1.

4.1 UTILs using CDDs

This phase is similar to the UTIL phase of DPOP (see Section 3.1). The difference is that the extensional representations of UTIL messages from DPOP (hypercubes) are replaced with CDD trees and the associated utility vectors. All corresponding operations on hypercubes (join, project) are redefined in the following for CDD messages.

Building CDDs from constraints: Algorithm 1 describes the construction of CDDTree corresponding to the Hypercube with dimension set *Dim[dimSize]. C* is an array of domain values (initially empty) which stores combinations currently found to lead to a valid solution. Whenever a new domain value is added to *C*, a consistency check is performed in line 9 to see if newly instantiated domain value will lead to a solution. This is a key step in pruning the problem space as inconsistent combinations are ruled out via this check. Parameter *currentLevel* denotes the current level in CDDTree under exploration. Its initial value is zero denoting the root.

The procedure `ConstructCDD` is based on depth-first backtracking search algorithm. A detailed explanation can be found in [2], though we describe the procedure briefly here. Set *S* (initially empty) consists the branches of the CDDNode, and D'_k consists values of variable X_k which can lead to valid combinations . Next for each value in domain of X_k (=D_k), we check if it can lead to a feasible solution (line 9). If no, it is ruled out otherwise we recursively invoke `ConstructCDD` to find the CDDNode *u* for the next level (*currentlevel+1*, line 10). If *u* is a 1-terminal (*null node*) or is not a 0-terminal , we add the branch (*d,u*) to *S*, and insert d to D'_k (lines 11 to 13). If $D'_k = \emptyset$ after all iterations are over, a 0-terminal is returned. Otherwise, `mkNode` is called to return the CDDNode for the current variable with given children and domain set (S and D'_k respectively).

Procedure `mkNode` is shown in algorithm 1. In line 18, an intermediate node *v* is created such that for every $d \in r$, $X_k \mapsto d$ leads to the same child node *u*. Next we check if an equivalent node $v\prime$ exists for node v to satisfy property (2) of a reduced CDD (line 19). If an equivalent node exists we reuse that node otherwise insert *v* to *V* and return *v* (line 22).

Joining two CDD messages In Algorithm 2 we describe the method for combining two CDD messages. The extra parameter *leafDimension* defines the dimension of the

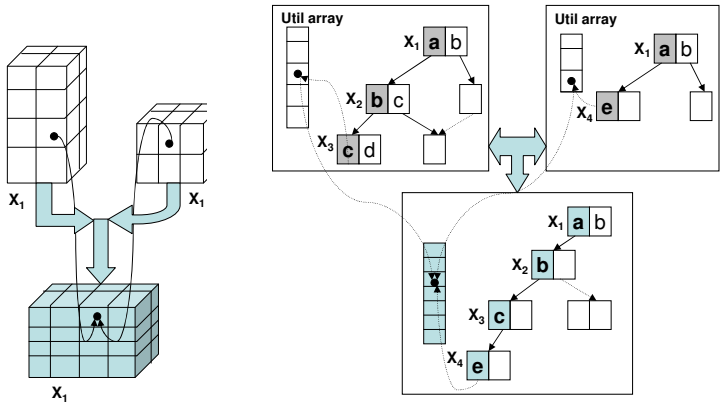
Algorithm 1: *Construction of a CDDtree*

Procedure `ConstructCDD`
input : Dim*[dimSize], C[dimSize], currentLevel*
output : The root of the CDDTree

1 **if** *dimSize == 0 || currentLevel == dimSize* **then**
2 | **return** *null*
end
3 X_k = node at currentLevel
4 $D_k = Dim[currentLevel].domain$
5 $S = \emptyset$
6 $D'_k = \emptyset$
7 **forall** $d \in D_k$ **do**
8 | $C[currentLevel] = d$
9 | **if** *isConsistent(C, currentLevel) == true* **then**
10 | | u = `ConstructCDD(`$Dim, C, currentLevel + 1$`)`
11 | | **if** u *==* $null$ $||$ $(u! = null$ $\&\&$ $u! = 0)$ **then**
12 | | | $S = S \cup \{< d, u >\}$
13 | | | $D'_k = D'_k \cup \{d\}$
| | **end**
| **end**
end
14 **if** $D'_k = \emptyset$ **then**
15 | **return** *0*
else
16 | v = `mkNode(`X_k, S, D'_k`)`
17 | **return** v
end

Procedure `mkNode`
input : X_k, S, D'_k
output : A CDDNode corrresponding to variable X_k with given domain and children set

18 $v = \{(X_k \in r, u) : d, d' \in r \Longleftrightarrow < d, u >, < d', u > \in S\}$
//`i.e.` $X_k \mapsto d$ *and* $X_k \mapsto d'$ *point to same node* u
19 **if** *htable.get(*v.`hashKey()`*)* $== null$ **then**
20 | $htable$.`add(`*v.hashKey()*, *v*`)`
| //`htable is the hashtable containing all discovered nodes`
21 | $V = V \cup \{v\}$
| //`Variable V contains all CDD nodes`
22 | **return** *v*
else
| //`i.e.` $\exists v' \in V$ *s.t.* $v' \equiv v$
23 | **return** v'
end



(a) Hypercube Join (b) CDDMessage join

Fig. 3. Join of two hypercubes and CDDMessages

nodes at the last level in CDDTree for the combined message (line 2). This step ensures a speedy projection of self's dimension by a node (see function `projectMine`, algorithm 3). The node which combines child CDD messages always sets *leafDimension* to be its own. Union of dimensions of individual messages forms the dimensions of combined message (line 1). With this new dimension set a new CDD message is constructed with empty *UtilArray*. The for loop in line 5 iterates over all paths of newly formed CDDTree, finds the relevant contributions from individual CDD messages (function `findUtil`, line 7), and sets the utility of new path in combined message (line 9). Finally the combined CDD message is returned after utility setting. Figure 3 shows the join of hypercubes and CDDs.

The procedure `findUtil` (algorithm 2) returns the utility value corresponding to CDDMessage's local contribution for the input source path with given dimension set *unionDim*. Array *myPath* stores the local contribution of the CDDMessage to input *srcPath*. It is initialized with values from *srcPath* for corresponding dimensions in *myDim* and *unionDim* (line 13). The utility value for *myPath* is extracted from *UtilArray* by finding the index of this path (line 14). For speeding up the performance each CDDMessage hashes every path of its CDDTree with value as path index and key as the path itself.

Projecting out a dimension Procedure `projectMine` (algorithm 3) is used by a node to project its own dimension after it has combined util messages from its children and pseudo parents. Since the `combineCDDMessages` function makes current node's dimension at the last, its projection becomes easy. We iterate through all the paths (line 2) and choose the best utility among paths having same prefix (excluding leaf level)(line 7). Note that in original DPOP after projecting the dimension we do not need to reconstruct the hypercube message but in case of CDDs it is necessary as the

Algorithm 2: *Combining two CDDMessages: JOIN operation*

Procedure `combineCDDMessages`
input : *Msg1, Msg2, leafDimension*
output : Combined CDDMessage of Msg1 and Msg2

begin
1 Dim[] *union* = $Msg1.DimensionArray \cup Msg2.DimensionArray$
2 Rearrange *union* array to make $leafDimension$ as the last one
3 CDDNode *CDDRoot* = `ConstructCDD(`*union, new Array(union.length), 0*`)`
4 *combinedMsg* = new `CDDMessage(`*CDDRoot, union, CDDRoot.pathsCount*`)`
`//pathsCount represents total paths from root to leaves`
end
5 **foreach** *path of CDDTree with root = CDDRoot* **do**
6 *path* = current path under consideration
7 *util1* = Msg1.`findUtil(`*union, path*`)`
8 *util2* = Msg2.`findUtil(`*union, path*`)`
9 *combinedMsg*.`setUtility(`*util1+util2, path.index*`)`
end
10 **return** *combinedMsg*

Procedure `findUtil`
input : *unionDim, srcPath*
output : The utility value corresponding to local contribution to *srcPath*

11 myDim = this.*DimensionArray*
12 Initialize *myPath* = new Vector(myDim.*length*)
13 *myPath* = $< (d_i = srcPath[j]) >$:
$i \in [0, myDim.length] \cap \exists j \ s.t. \ srcDim[j].id = myDim[i].id$
14 *index* = htable.`getvalueByKey(`*myPath*`)`
15 **return** *this.UtilArray[index]*

Algorithm 3: *PROJECT operation for a CDDMessage*

Procedure `projectMine`
output : Projects last dimension of this CDDMessage and returns new CDDMessage

```
1  Initialize BestUtilities = new Vector()
2  foreach path of CDDTree of this CDDMessage do
3  |   path = currentPath under consideration
4  |   pathPrefix = path.prefix(0, path.size-1)
5  |   if utility already set for pathPrefix then
6  |   |   continue
   |   else
7  |   |   util = Max(P_i.util : P_i.prefix(0,P_i.size − 1) = pathPrefix ∩ P_i ∈
   |   |   {paths of CDDTree})
8  |   |   BestUtilities.set(util, pathPrefix)
   |   end
   end
9  Initialize newDim[] to this.DimensionArray[0] to [totalSize-1]
10 newTree = constructCDD(newDim, new Array(newDim.length), 0)
11 newMsg = new CDDMessage(newTree.root, newDim, newTree.pathsCount)
12 newMsg.UtilArray = BestUtilities
13 return newMsg
```

tree will not be optimal (in terms of size) after removing the leaf level (line 10). Finally the newly formed CDDMessage is returned to be sent to the parent of current node.

4.2 The `isConsistent` plugin mechanism

The `isConsistent` (see algorithm 1, line 9) function is like a gateway to the constraint problem being solved and uses hard constraint propagation for pruning problem space. Until now existing DCOP algorithms like ADOPT, DPOP did not try to take advantage by gaining insights into the problem domain. H-DPOP is unique in this sense as it provides the problem representation to the constraint algorithm through this modular plugin mechanism. Our results show that this knowledge can help reducing UTIL message size with up to 99%.

Consequently, this function is very problem specific and encapsulates the pruning logic into the H-DPOP algorithm. The input to this function is the constraint array C, which is a combination of domain values. It then processes this input using hard constraint propagation and determines if C represents a feasible combination.

We now describe the implementation of this function for the server problem described in section 2.1. The underlying logic is that, whenever two variables assume same domain value, it represents an invalid combination as problem hard constraint does not allow this.

It can be observed that this function is very simple to implement. Although we need to invoke this function for each combination of domain values, in practice savings provided by it often dominate the execution overhead. Next section provides experimental results for H-DPOP and discusses some practical issues related to it.

Procedure `isConsistent(`*C, currentIndex*`)`

output : true or false depending C is valid or not

for $i = 0$ ***to*** $currentIndex - 1$ **do**

  if $C[i] == C[currentIndex]$ **then**

    return false

  end

end

return true

5 Experimental Results

This section discusses the performance of H-DPOP on optimal query placement in an overlay network. This problem has been introduced in section 2.1.

5.1 Optimal query placement in an overlay network

For experiments the problem is made deliberately very constrained by allowing each server to execute same set of services. For simplicity sake, each server can execute only a single service at a time. The objective of the DCOP algorithm is to maximize the overall utility.

We generated random problems of different sizes, with random number of soft constraints among variables. Hard constraints are then introduced to make the constraint graph fully connected. We performed two runs for each problem size and averages were taken for results.

Problem Size	**Maximum Size(DPOP)**	**Maximum Size(H-DPOP)**	**Space Wastage (Hypercube)**	**Savings by CDD**
5*5	5^4	325	96.1%	48.0%
5*10	10^4	7500	69.7%	25.0%
5*15	15^4	43900	52.54%	13.3%
6*6	6^5	1248	98.4%	83.9%
6*10	10^5	35820	84.8%	64.1%
6*12	12^5	108342	77.7%	56.4%
7*7	7^6	6321	99.3%	94.6%
7*8	8^6	22816	98.0%	91.3%
7*9	9^6	65637	96.2%	87.6%
7*11	11^6	344366	91.4%	80.5%
8*8	8^7	43312	99.7%	97.9%
8*9	9^7	187773	99.1%	96.0%
8*10	10^7	613490	98.1%	93.8%
9*9	9^8	369693	99.9%	99.1%

Table 1. Maximum message size: DPOP Vs H-DPOP

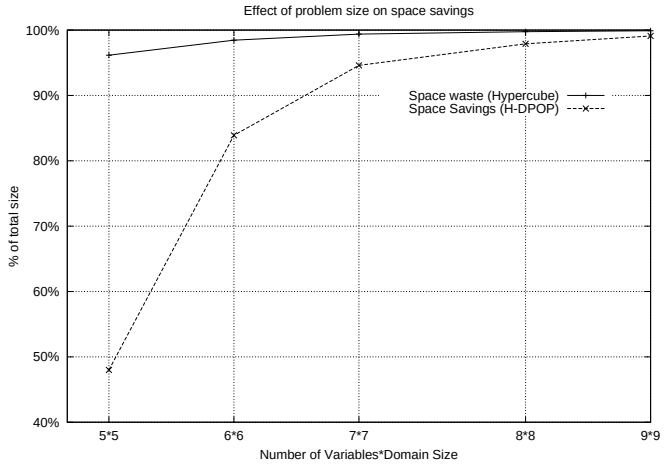


Fig. 4. H-DPOP performance - Space Savings

Space Savings in H-DPOP: Table 1 shows maximum message size in H-DPOP versus normal DPOP. Problem size is denoted by $m * n$ implying m variables each having same domain with size n. Results show that H-DPOP is much superior than DPOP for all problem sizes.

However the extent of savings provided by CDDs is dependent on both the problem size and space wastage in DPOP. Space wastage can be calculated by finding the valid combinations and the total message size. Savings by CDDs follow the trend of space wastage. As the memory wastage increases/decreases CDD savings also increase/decrease. This is obvious and expected (problem size 5*15 where memory wastage is relatively minimum, CDDs provide minimum savings, 13% only).

Figure 4 shows the effect of problem size on space savings provided by CDDs. As the problem size increases the graph of CDD savings follows closely the graph of memory wastage (at 5*5 although space wastage is 96.16% but CDDs provide only 48% savings, for 6*6 space wastage is similar being 98.4% but CDDs provide higher savings, 83.9%). This trend occurs because size of CDDTree is very comparable to the DPOP hypercube message at small problem sizes. At small instances the node sharing in CDDTree is relatively less. However at larger problem instances the size of CDDTree is much less than hypercube message and the pruning is greater. So CDD is able to provide more savings.

Effect of problem complexity on H-DPOP Execution Time: Figure 5 shows the execution time of H-DPOP versus DPOP. H-DPOP is slightly expensive for small to medium problem instances (5*5 to 7*9). This is because in such cases the memory requirements for DPOP are not excessive. In H-DPOP the CDDTree formation and domain pruning require execution overhead (see section 4.2).

We also see sudden peaks in H-DPOP execution time (at 5*15, 6*12, 7*11). These arise due to high number of valid domain combinations (for eg. 5*15 has many valid server-service combinations than 5*5 and 5*10). Due to this exponential increase in the number of valid paths CDDTree formation overhead makes H-DPOP expensive than DPOP.

When the problem size becomes very large (from 8*8 to 8*10 with search space 8^8, 10^8) H-DPOP is much better than DPOP. This happens because now the memory requirements for DPOP becomes excessive which dominates its execution time. For size 9*9 DPOP was unable to find an answer but H-DPOP executed successfully (in 125 Sec).

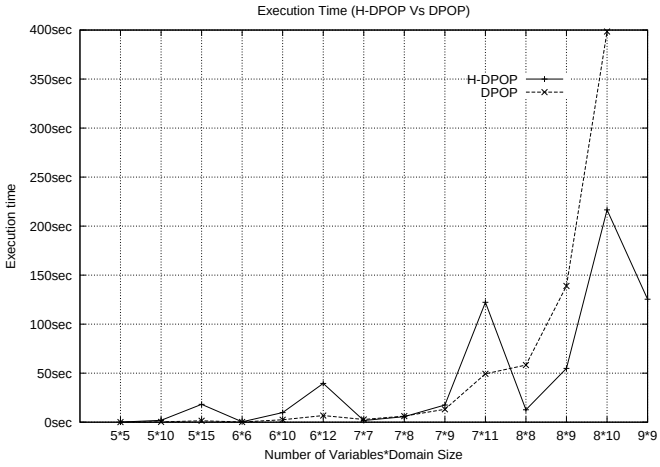


Fig. 5. H-DPOP performance - Execution Time

6 Related work

This paper draws mostly from the dynamic programming algorithm DPOP (Petcu and Faltings [9]), and Constraint Decision Diagrams (Cheng and Yap [2]). DPOP is a dynamic programming algorithm that produces large arity relations that are sent over the network. On the other hand, CDDs can represent compactly such large arity relations, thus being a well suited alternative for minimizing network traffic and memory requirements for DPOP.

Recently, And/Or Multi-valued Decision Diagrams (AOMDDs) have been introduced by Mateescu and Dechter in [7]. They first arrange the problem as a pseudotree (of which DFS is a special case). Subsequently, on that pseudotree structure, they start a bottom-up compilation, by computing (and subsequently joining) high-arity relations

(as in DPOP). However, their purpose is to have a compact compilation of the entire constraint network in the root node. Therefore, they do not execute projections at each node along the way, thus obtaining a large AOMDD at the root, that represents the entire network. AOMDDs are space- and computation-exponential in the induced-width of the problem.

In principle, CDDs are OR-based structures, so for a complete compilation of the network, they are exponential in the size of the problem, therefore less efficient than AOMDDs. However, since each variable projects itself out of the outgoing message, our CDD representations are also guaranteed to be only exponential in the induced width of the problem, as opposed to exponential in the problem size.

7 Conclusion and future work

In this paper we presented a new algorithm for constraint optimization based on DPOP. Replacing hypercubes with constraint decision diagrams (CDDs) as message passing mechanism, and efficient pruning of problem space using hard constraints provide significant space savings. The core of the algorithm is a simple plugin mechanism for pruning search space which can be easily implemented for a range of problems. Our experiments show that H-DPOP provides significant savings for large and highly constrained problems while incurring a little time overhead for smaller instances.

There are many realistic scenarios where H-DPOP can prove very effective. For example, problems involving bidding on paths in space like railroad auctions and truck routes auctions have this property. Such problems are highly constrained and large but relatively sparse and can be efficiently solved by H-DPOP. Other examples include time slot auctions (airport time allocation) and advanced versions of the service allocation problem mentioned in this paper. In future we plan to work on these problems. Currently we also need to send domains of context variables to higher nodes, which may pose some security concerns. Future work also aims at finding a secure way of doing this.

References

1. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
2. Kenil C. K. Cheng and Roland H. C. Yap. Constrained decision diagrams. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-05*, pages 366–371, Pittsburgh, USA, 2005.
3. Rina Dechter. Bucket elimination: A unifying framework for processing hard and soft constraints. *Constraints: An International Journal*, 7(2):51–55, 1997.
4. Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
5. Eugene C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32(14):755–761, 1985.
6. Roger Mailler and Victor Lesser. Asynchronous partial overlay: A new algorithm for solving distributed constraint satisfaction problems. *Journal of Artificial Intelligence Research (JAIR)*, 2005. to appear.

7. Robert Mateescu and Rina Dechter. Compiling Constraint Networks into AND/OR Multi-Valued Decision Diagrams (AOMDDs). In *Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming (CP'06)*, Nantes, France, October 2006.
8. P. J. Modi, W. M. Shen, and M. Tambe. An asynchronous complete method for distributed constraint optimization. In *Proc. AAMAS*, 2003.
9. Adrian Petcu and Boi Faltings. A scalable method for multiagent constraint optimization. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI-05*, Edinburgh, Scotland, Aug 2005.
10. Adrian Petcu, Boi Faltings, and David Parkes. MDPOP: Faithful Distributed Implementation of Efficient Social Choice Problems. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS-06)*, Hakodate, Japan, May 2006.
11. Thomas Schiex, Hélène Fargier, and Gerard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence, IJCAI-95*, Montreal, Canada, 1995.
12. 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.

IDB-ADOPT : A Depth-First Search DCOP Algorithm

William Yeoh[†], Sven Koenig[†], Ariel Felner[‡]

[†]Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
{wyeoh, skoenig}@usc.edu

[‡]Department of Information Systems Engineering
Ben-Gurion University of the Negev
Beer-Sheva, 85104, Israel
felner@bgu.ac.il

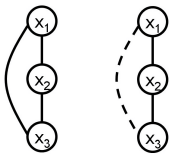
Abstract. Many agent coordination problems can be modeled as distributed constraint optimization problems (DCOPs). ADOPT is an asynchronous and distributed search algorithm that is able to solve DCOPs optimally. In this paper, we introduce Iterative Decreasing Bound ADOPT (IDB-ADOPT), a modification of ADOPT that changes the search strategy of ADOPT from performing one best-first search to performing a series of depth-first searches. Each depth-first search is provided with a bound, initially a large integer, and returns the first solution whose cost is smaller than or equal to the bound. The bound is then reduced to the cost of this solution minus one and the process repeats. If there is no solution whose cost is smaller than or equal to the bound, it returns a cost-minimal solution. Thus, IDB-ADOPT is an anytime algorithm that solves DCOPs with integer costs optimally. Our experimental results for graph coloring problems show that IDB-ADOPT runs faster (that is, needs fewer cycles) than ADOPT on large DCOPs, with savings of up to one order of magnitude.

1 Introduction

Many agent coordination problems can be modeled as distributed constraint optimization problems (DCOPs), including the scheduling of meetings [8], the coordination of unmanned aerial vehicles [14], and the allocation of targets to sensors in sensor networks [7, 10, 13]. Unfortunately, solving DCOPs optimally is NP-hard. A variety of search algorithms have therefore been developed to solve DCOPs as fast as possible to scale up to real-world domains [9, 10, 12, 3]. ADOPT (Asynchronous Distributed Constraint Optimization) [10] is one of the pioneering DCOP algorithms and currently probably the most extended one [11, 4, 2]. It is an asynchronous and distributed best-first search algorithm that only needs a bounded amount of memory at each vertex and is able to solve DCOPs optimally. Researchers have recently scaled up ADOPT by one order of magnitude by providing it with informed heuristics that focus its search [1]. However, its runtime is still large for realistic DCOPs and it therefore needs to get scaled up further. In particular, in the original ADOPT, each vertex can switch back and forth between different values and then has to redo many searches since the results

from the previous searches have already been purged from memory due to its memory limitations. In this paper, we address this problem of ADOPT by introducing Iterative Decreasing Bound ADOPT (IDB-ADOPT), a modification of ADOPT that changes the search strategy of ADOPT from performing one best-first search to performing a series of depth-first searches, where vertices do not switch back and forth between different values. IDB-ADOPT is motivated by insights from heuristic search that depth-first searches can outperform best-first searches in combinatorial domains with search trees whose depths are bounded [15], and DCOPs are such domains. Each depth-first search of IDB-ADOPT is provided with a bound, initially a large integer, and returns the first solution whose cost is smaller than or equal to the bound. The bound is then reduced to the cost of this solution minus one. If there is no solution whose cost is smaller than or equal to the bound, it returns a cost-minimal solution. Thus, IDB-ADOPT is an anytime algorithm that solves DCOPs with integer costs optimally. Our experimental results for graph coloring problems show that IDB-ADOPT runs faster than ADOPT on large DCOPs, with savings of up to one order of magnitude.

2 DCOPs



x_1	x_2	Cost
0	0	5
0	1	8
1	0	20
1	1	3

x_1	x_3	Cost
0	0	5
0	1	8
1	0	20
1	1	3

x_2	x_3	Cost
0	0	5
0	1	8
1	0	20
1	1	3

Fig. 1. DCOP Example

Distributed constraint optimization problems (DCOPs) model agent coordination problems as constraint optimization problems on *constraint graphs*. Each vertex of a constraint graph represents an agent (sometimes referred to as variable) and can *take on* a value from a given set (its domain). Edges denote constraints. The cost of a constraint depends on the values of the vertex endpoints of the corresponding edge, given by a table. We assume in this paper that all costs are non-negative integers. An assignment of values to all vertices is a (complete) *solution*. The *cost* of the solution is the sum of the costs of the constraints. One wants to find a cost-minimal solution. As an example, Figure 1 (left) shows a simple DCOP with three vertices, x_1, x_2 and x_3, that each can take on the values zero or one. There are three constraints, whose costs are specified by the tables. For example, there is one constraint between vertices x_1 and x_2. If both vertices take on value zero, then the cost of the constraint is five. The cost of the solution $x_1 := 1$, $x_2 := 1$ and $x_3 := 1$ is nine and cost-minimal.

3 ADOPT

We now give a slightly simplified description of ADOPT that makes the search principle behind it easy to understand and is sufficient for our purposes. The reader is referred to the original paper [10] for a full step-by-step description of DCOPs, ADOPT and the message passing mechanism used by ADOPT. ADOPT basically operates as follows: In a preprocessing step, ADOPT transforms the constraint graph into a *constraint tree* with the property that constraints exist only between a vertex and its ancestors and/or descendants. To simplify our description further, we assume that every vertex has at most one child in the constraint tree. In other words, the constraint tree is a chain, which is the case for our example DCOP. Figure 1 (center) shows one possible constraint tree for our example DCOP. ADOPT then performs a search as will be described next.

3.1 Values of ADOPT

During the search of ADOPT, every vertex of the constraint graph maintains some values. Every vertex maintains the value from its domain that it currently takes on (called its *current value*), initially the best value (the best value is defined below). Every vertex also maintains the values of its (connected) ancestors in the constraint tree (called its *current context*). These values correspond to a partial solution of the DCOP. Every vertex maintains, for each possible value that it can take on, *lower bounds* on the cost of the cost-minimal solution of the DCOP that is consistent with this value and its current context. These lower bounds are initialized with the sum of the costs of the constraints between the (connected) ancestors, which can be calculated since the current context is known. One can obtain larger initial lower bounds to speed up the search by adding informed pre-computed values (called heuristics), if available, to the the sum of the costs of the constraints between the (connected) ancestors. We call the lower bound of the current value of a vertex its *current lower bound*. We call the smallest lower bound over all values that a vertex can take on its *best lower bound* and the corresponding value its *best value*. Every vertex also maintains an *upper bound* on the cost of the cost-minimal solution of the DCOP that is consistent with its current context. The upper bound is simply the cost-minimal solution found so far during the search that is consistent with the current context, initially infinity. Finally, every vertex also maintains a *threshold* (whose role will be explained below), initially zero. For every vertex, ADOPT maintains the following threshold invariant: The threshold of a vertex is guaranteed to be between its best lower bound and its upper bound. To keep the threshold invariant satisfied, the vertex changes the value of its threshold as follows: If the threshold is smaller than the best lower bound of the vertex then the threshold is increased to the best lower bound. Similarly, if the threshold is larger than the upper bound then the threshold is decreased to the upper bound. These situations occur when the best lower bound increases above the threshold or the upper bound decreases below the threshold.

3.2 Operation of ADOPT

Each vertex operates as follows: If its current lower bound is smaller than or equal to the threshold, then the vertex keeps its current value. Otherwise, it changes its current

value by taking on its best value and then informs its (connected) descendants in the constraint tree about its new value. Its descendants then perform similar computations to help the vertex decrease its upper bound and increase its current lower bound. ADOPT terminates when the threshold of the root vertex of the constraint tree is equal to its upper bound.

3.3 Thresholds of ADOPT

The threshold of a vertex is of special importance in the remainder of this paper. We now explain how it influences the values taken on by the vertex. As already explained above, if the current lower bound of a vertex is smaller than or equal to the threshold, then the vertex keeps its current value. Otherwise, it changes its current value by taking on its best value. At this point in time, there are two possible cases:

- **Case 1:** If there are still values whose lower bounds are smaller then its threshold, then the vertex takes on its best value and keeps it until the lower bound of that value increases above the threshold. The vertex repeats this procedure until all lower bounds are larger than or equal to the threshold and Case 2 is reached. Note that, during Case 1, the vertex takes on each value only once, unless its ancestors switch values, and keeps this value as long as the lower bound of the value is smaller than or equal to the threshold even if some other value has a smaller lower bound. This results in a depth-first search behavior.
- **Case 2:** If all lower bounds are larger than or equal to the threshold, then the vertex increases the threshold to the best lower bound (to satisfy the threshold invariant), if necessary, and then takes on its best value until the lower bound of that value increases. The vertex then repeats this procedure. Note that, once Case 2 is reached, the vertex cannot go back to Case 1 unless its ancestors switch values. During Case 2, the vertex always takes on its best value. This results in a best-first search behavior. The vertex can switch back and forth between values and, in the process, take on the same value several times.

ADOPT initializes the threshold of the root vertex of the constraint tree to zero. The root vertex therefore starts with Case 2, and ADOPT performs a best-first search. The threshold is important when a vertex switches back to a value that it had taken on earlier already during the best-first search. In this case, it has already increased the lower bound of this value, otherwise it would not have switched from the value to a different one earlier. It now has to redo this search to restore the lower bounds of its descendants at the point in time when it last switched from this value to another value. These lower bounds have been purged from memory since each vertex uses only a bounded amount of memory. ADOPT restores these lower bounds with a depth-first search (inside the best-first search) to be efficient. A best-first search is not needed for this purpose since ADOPT only repeats a previous search. Case 1 performs this depth-first search automatically.

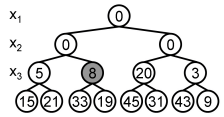


Fig. 2. Search Tree

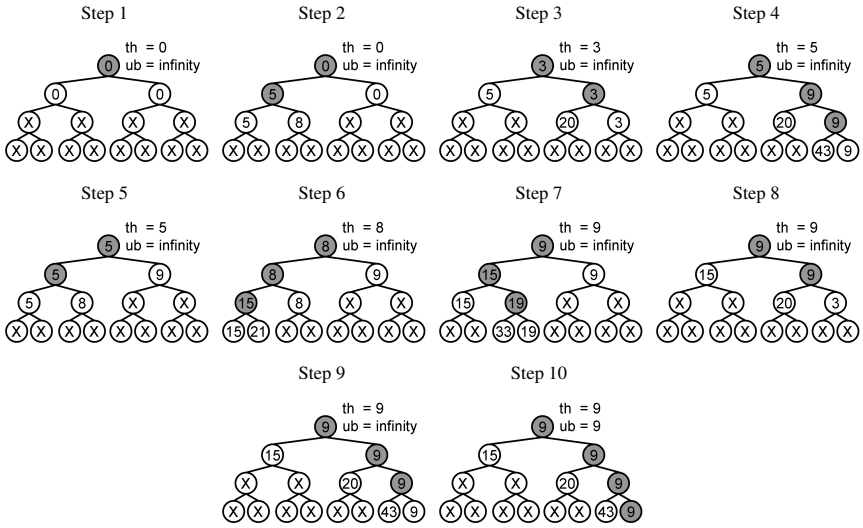


Fig. 3. Illustration of ADOPT

4 Illustration of ADOPT

We now visualize the searches that ADOPT performs for our example DCOP. We use search trees for this purpose, as shown in Figure 2. The constraint tree of our example DCOP orders the vertices completely since it is a chain. The search tree reflects this ordering. Its first (second, third) layer corresponds to vertex x_1 (x_2, x_3), respectively. Left branches means that the corresponding vertex takes on the value zero, and right branches mean that it takes on the value one. Thus, each node in the search tree represents the values of all ancestors of the corresponding vertex in the constraint tree. The shaded node in the search tree, for example, corresponds to $x_1 := 0$ and $x_2 := 1$. It is annotated with the initial lower bound on the cost of the cost-minimal solution of the DCOP that is consistent with these values for the zero heuristics (no heuristics were added), in this case the cost of the constraint between vertices x_1 and x_2 (= 8). This value is the initial lower bound of vertex x_2 for value one if its ancestor x_1 has value zero. Thus, the lower bounds of a vertex label the children of the vertex in the search tree. To simplify our description, we assume that ADOPT performs a synchronous rather than an asynchronous search and propagates information with infinite speed. Figure 3 shows

the resulting search. Consider the root vertex x_1. ADOPT initializes its threshold (th) with zero, its upper bound (ub) with infinity, and its lower bounds with the initial lower bounds shown earlier. Both of its lower bounds are equal to zero. Thus, it breaks ties and takes on value zero (Step 1). The lower bounds of vertex x_2 are initialized with the initial lower bounds shown earlier for $x_1 := 0$. The best value of vertex x_2 is zero and its best lower bound is five. Thus, vertex x_1 can update its lower bound for value zero from zero to five (Step 2). The best value of vertex x_1 is now one and its best lower bound (shown inside the root node of the search tree) is zero. Vertex x_1 then switches to value one. The lower bounds of vertex x_2 are initialized with the initial lower bounds shown earlier for $x_1 := 1$. The best value of vertex x_2 is one and its best lower bound is three. Thus, vertex x_1 can update its lower bound for value one from zero to three. This violates the threshold invariant, and thus the threshold is also increased to three. Its best value, however, remains unchanged. Vertex x_2 thus takes on value one. The lower bounds of vertex x_3 are initialized with the initial lower bounds shown earlier for $x_1 := 1$ and $x_2 := 1$. The best value of vertex x_3 is one and its best lower bound is nine. Thus, vertex x_2 can update its lower bound for value one from three to nine. Then, vertex x_1 can update its lower bound for value one from three to nine (Step 4). The best value of vertex x_1 is now zero and its best lower bound is five. Vertex x_1 thus updates its threshold to five and then switches to value zero. The lower bounds of vertex x_2 are initialized with the initial lower bounds shown earlier for $x_1 := 0$ and the previous lower bounds are purged from memory. (If the ancestors of a vertex switch their values, then the vertex changes its node in the search tree to a different node in its layer. Since each vertex has only a bounded amount of memory, it can store information only for its current node in the search tree. Thus, it has to delete its current lower bounds, as shown in the figure with the X's, and replace them with the initial lower bounds for the new values of its ancestors.) The search continues and eventually reaches the node with $x_1 := 1$, $x_2 := 1$ and $x_3 := 1$ in Step 10. This is a solution with cost nine. Thus, vertex x_1 updates its upper bound to nine, the termination condition is satisfied, and ADOPT terminates with the cost-minimal solution $x_1 := 1$, $x_2 := 1$ and $x_3 := 1$.

It is interesting to see that the behavior of ADOPT is similar to that of Korf's recursive best-first search (RBFS) [6], which ADOPT generalizes to the asynchronous and distributed case. For example, ADOPT does not need centralized control and is able to take advantage of parallel computations in case it operates on constraint trees that are not chains. ADOPT and RBFS operate under the same memory limitations. They both perform best-first searches and use depth-first searches to redo previous best-first searches in order to restore information already purged from memory. Vertex x_1 in the example switches back and forth between values zero and one, and then has to redo the previous searches. For example, the best value of vertex x_1 is one and its best lower bound is nine in Step 7. Vertex x_1 thus switches to value one. The lower bounds of vertex x_2 are initialized with the initial lower bounds shown earlier for $x_1 := 1$ (namely, 20 and 3), but were already larger at the end of the previous search with $x_1 := 1$ in Step 4 (namely, 20 and 9). ADOPT uses a depth-first search to restore them in Steps 9-10, which is similar to what RBFS does in this situation.

```
procedure IDB-Adopt()
{01} threshold := a large integer;
{02} loop
{03}    set the threshold of the root vertex to threshold;
{04}    run the original ADOPT algorithm;
{05}    if (solution quality found > threshold)
{06}       return solution found;
{07}    end if;
{08}    threshold := solution quality found - 1;
{09} end loop;
```

Fig. 4. IDB-ADOPT

5 IDB-ADOPT

A best-first search without memory limitations visits only the necessary nodes in the search tree to find the optimal solution [5]. However, ADOPT performs a best-first search where each vertex has only a bounded amount of memory and thus has to redo many searches. To remedy this situations, we make the following observation about ADOPT: If the cost of the cost-minimal solution is less than or equal to the threshold of the root vertex, then ADOPT performs a depth-first search and terminates after finding the first solution whose cost is less than or equal to the threshold.

> **Explanation:** When the initial threshold of the root vertex is smaller than the initial upper bound of the root vertex but larger than or equal to the cost of the cost-minimal solution (which implies that it is also larger than or equal to the best lower bound of the root vertex), ADOPT performs a depth-first search. The upper bound of the root vertex is the cost of a cost-minimal solution found so far. If the upper bound is larger than the threshold, then the depth-first search continues. Once the upper bound is smaller than or equal to the threshold, then the threshold gets set to the upper bound and the termination condition of ADOPT is satisfied. Thus, once the depth-first search finds a solution with a cost that is smaller than or equal to the threshold, ADOPT terminates with that solution.

Our objective is to make ADOPT faster by only modifying it slightly based on the above observation. We introduce Iterative Decreasing Bound ADOPT (IDB-ADOPT), a modification of ADOPT that changes the search strategy of ADOPT from performing a best-first search to performing a series of depth-first searches. It assumes that the constraint costs are non-negative integers. Thus, if there is no solution of integer cost x or smaller, then the cost-minimal solution must have a cost of $x+1$ or larger. Figure 4 shows the pseudo code of IDB-ADOPT, which uses ADOPT to implement the depth-first searches. IDB-ADOPT sets the threshold of the root vertex to a large integer, that is, an integer larger than or equal to the cost of a cost-minimal solution. Such an integer can easily be obtained by summing the largest possible cost of each constraint over all constraints. IDB-ADOPT then runs ADOPT. According to the above observation, ADOPT terminates with a solution whose cost is less than or equal to the threshold if

such a solution exists, which is the case since the threshold is larger than or equal to the cost of a cost-minimal solution. IDB-ADOPT then sets the threshold of the root vertex to the cost of the solution minus one and runs ADOPT again. This process continues until ADOPT terminates with a solution whose cost is larger than the threshold. This solution is a cost-minimal solution.

> **Explanation:** We use x to refer to the threshold of the root vertex of the constraint tree at the beginning of the last search of ADOPT. Note that the previous (second-to-last) search of ADOPT has already found a cost-minimal solution of cost $x+1$. The last search of ADOPT only verifies that the solution is indeed cost-minimal. It behaves as follows: ADOPT performs a depth-first search until all lower bounds of the root vertex of the constraint tree are larger than x. At this point in time, at least one of its lower bounds is smaller than or equal to the cost of a cost-minimal solution and thus equal to $x+1$. ADOPT either has not found a solution of cost $x+1$ yet or has found such a solution already. In the first case, the root vertex takes on its best value whose lower bound is, as argued above, $x+1$ and performs a depth-first search until it either finds a solution with that cost or increases the lower bound of that value and then repeats the process with a different value whose lower bound is $x+1$. (In this case, it redoes one search for each value that it revisits. However, it cannot revisit any value more than once since it will find a solution with cost $x+1$ during one of the searches and thus will not take on values whose lower bounds are larger than $x+1$. Notice that this property is not guaranteed for initial thresholds of the root node of the constraint tree that are smaller than x, including the zero value used by ADOPT.) Finally, it finds a solution with cost $x+1$ since one exists. Its upper bound is then set to $x+1$. In the second case, its upper bound is already equal to $x+1$. Either way, its best lower bound is now equal to its upper bound and its threshold is always between the two. Thus, its threshold is now equal to its upper bound and the termination condition of ADOPT is satisfied. ADOPT then terminates with a solution with cost $x+1$, which must be a cost-minimal solution since the best lower bound of the root vertex of the constraint tree is equal to its upper bound.

Thus, IDB-ADOPT is, like ADOPT, an asynchronous and distributed search algorithm that only needs a bounded amount of memory at each vertex and is able to solve DCOPs optimally. IDB-ADOPT checks whether the cost of the solution found by ADOPT is larger than the threshold. If so, it returns this solution, which is a cost-minimal solution. Otherwise, it runs ADOPT again (from scratch) with a new threshold. Since this threshold is reduced from one ADOPT search to the next, ADOPT finds solutions of smaller and smaller costs until it eventually finds the cost-minimal solution. Thus, IDB-ADOPT can be used as an anytime algorithm [16].

6 Illustration of IDB-ADOPT

Figure 5 shows the searches of IDB-ADOPT for our example DCOP. Consider the root vertex x_1. IDB-ADOPT initializes its threshold with 60 (the sum of the largest possible

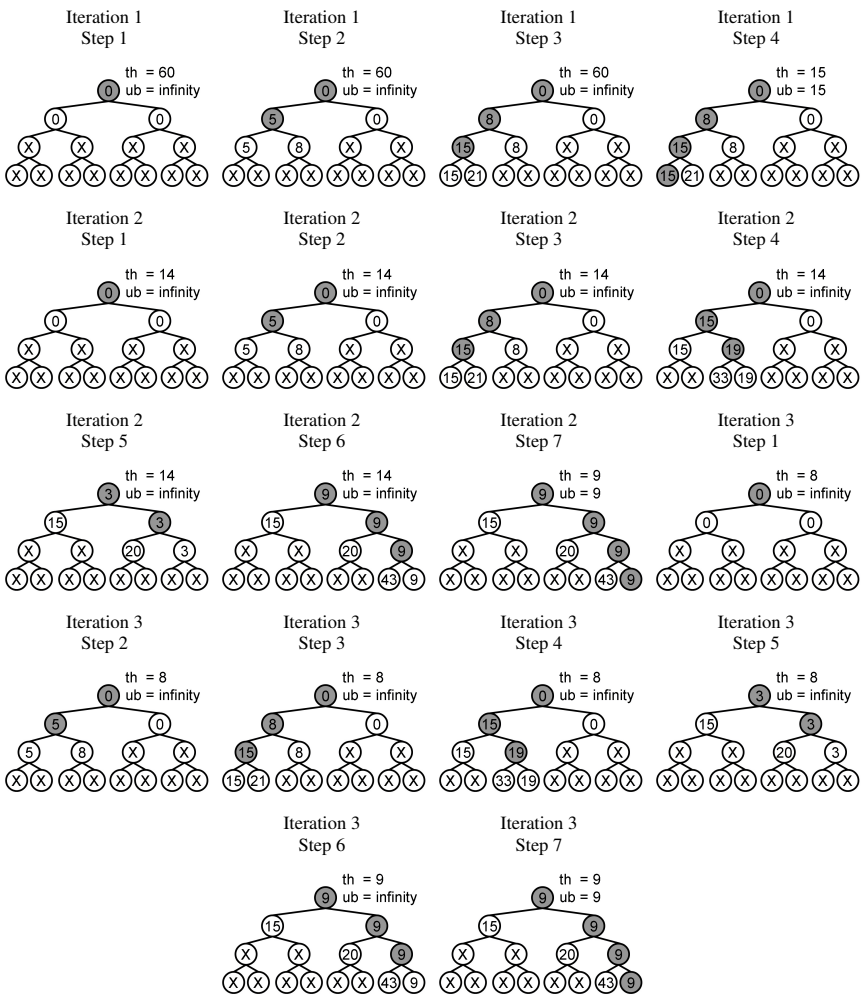


Fig. 5. Illustration of IDB-ADOPT

cost of each constraint over all constraints, which is guaranteed to be larger than or equal to the cost of a cost-minimal solution), its upper bound with infinity, and its lower bounds with the initial lower bounds shown earlier. IDB-ADOPT then starts the first ADOPT search. Both of its lower bounds are equal to zero. Thus, it breaks ties and takes on value zero (Iteration 1, Step 1). Now consider vertex x_2. Its lower bounds are initialized with the initial lower bounds shown earlier for $x_1 := 0$. The best value of vertex x_2 is zero and its best lower bound is five. Thus, vertex x_1 can update its lower bound for value zero from zero to five. The best value of vertex x_1 is now one and its best lower bound is zero. However, vertex x_1 does *not* change its value since the lower bound of its current value remains below the threshold. Vertex x_2 thus takes on value

zero. The lower bounds of vertex x_3 are initialized with the initial lower bounds shown earlier for $x_1 := 0$ and $x_2 := 0$. The best value of vertex x_3 is zero and its best lower bound is fifteen (Iteration 1, Step 3). Thus, vertex x_2 can update its lower bound for value zero from five to fifteen. The best value of vertex x_2 is now one and its best lower bound is eight. However, vertex x_2 does not change its value since the lower bound of its current value remains below the threshold (Iteration 1, Step 2). Thus, the search has reached the node with $x_1 := 0$, $x_2 := 0$ and $x_3 := 0$ in Iteration 1, Step 4. This is a solution with cost fifteen. Thus, vertex x_1 updates first its upper bound to fifteen and then also its threshold to fifteen. The termination condition is satisfied, and the ADOPT search terminates with the solution $x_1 := 0$, $x_2 := 0$ and $x_3 := 0$. Consider again the root vertex x_1. IDB-ADOPT then initializes its threshold with fourteen, its upper bound with infinity, and its lower bounds with the initial lower bounds shown earlier. IDB-ADOPT then starts the second ADOPT search, and so on. Three observations are important here: First, each ADOPT search performs a depth-first search and backtracks only when the lower bound of a value is larger than the threshold. For example, vertex x_2 switches from value zero to value one in Iteration 2, Steps 3-4 because the lower bound of its value zero has become larger than the threshold. No vertex switches back during an ADOPT search to a previous value unless its ancestors have switched values. Second, different ADOPT searches do repeat some of the effort. All three ADOPT searches, for example, consider the case where $x_1 := 0$ and $x_2 := 0$. Finally, the second ADOPT search already found the cost-minimal solution but the third ADOPT search is needed to verify that it is indeed cost-minimal. Note that our example DCOP is too small for IDB-ADOPT to run faster than ADOPT, which we will explain below.

7 ADOPT versus IDB-ADOPT

ADOPT and IDB-ADOPT compare as follows: IDB-ADOPT performs repeated ADOPT searches that produce better and better solutions. Each ADOPT search performed by IDB-ADOPT has the property that vertices do not switch back and forth between different values, unless their connected ancestors have switched values, and thus does not incur the overhead of redoing previous searches. In fact, IDB-ADOPT redoes no search within an ADOPT search (except for the last one). It achieves this efficiency by performing depth-first searches rather than best-first searches. However, depth-first searches are sources of a different inefficiency since they explore partial solutions that best-first searches do not explore. Thus, there is a trade-off between using a best-first search and having to explore partial solutions repeatedly, and using a depth-first search and having to explore additional (unimportant) partial solutions. We expect a best-first search to do better if the heuristics (that are used to initialize the lower bounds) are good and it thus does not have to redo many searches. We expect a depth-first search to do better if the heuristics are misleading, for example, if they are uninformed or the DCOPs are large. Our experimental results for graph coloring problems indeed show that IDB-ADOPT runs faster than ADOPT on large DCOPs. Note, however, that it was our objective to modify ADOPT only slightly. Indeed, IDB-ADOPT modifies ADOPT by putting a control loop on top of ADOPT that is only 9 lines long and calls it repeatedly with different thresholds for the root vertex of the constraint tree. This slight

modification, however, does not implement the principle of a depth-first search fully. In fact, IDB-ADOPT needed to perform only a single complete branch-and-bound depth-first search and return the cost-minimal solution found. Instead, IDB-ADOPT performs repeated depth-first searches, each of which repeats parts of the previous depth-first searches, which results in additional (unnecessary) overhead because IDB-ADOPT partially redoes searches from one ADOPT search to the next. Every vertex takes on all of its values that are smaller than or equal to the threshold, unless the ADOPT search terminated before that. Since the threshold of the previous ADOPT search was larger, the vertex has taken on these values already during the previous ADOPT search, unless the previous ADOPT search terminated before that. (The previous ADOPT search terminated earlier than the current one since the threshold of the previous ADOPT search was smaller than the one of the current ADOPT search.) Thus, the current ADOPT search can prune more than the previous ADOPT search but needs to search beyond the solution found by the previous ADOPT search. Our experimental results show that IDB-ADOPT still runs faster than ADOPT on large DCOPs in spite of this overhead. An asynchronous and distributed branch-and-bound depth-first search algorithm would run even faster than IDB-ADOPT. It would share with ADOPT and IDB-ADOPT that it only needs a bounded amount of memory at each vertex and is able to solve DCOPs optimally. It is future work to develop such an algorithm.

8 Experiments

We evaluated IDB-ADOPT against ADOPT with uninformed heuristics (zero heuristics) and the currently best-known informed heuristics (dp2 heuristics) [1] on graph-coloring problems. Their number of vertices varied from 5 to 10. Their constraint costs were in the range from one to an upper bound that varied from 3 over 10, 25, 50, 100 to 10000. We randomly generated 500 graph-coloring problems with three values per vertex and an average link density of four for each configuration of these two parameters.

In Experiment 1, we measured the average number of cycles needed by IDB-ADOPT and ADOPT for finding optimal solutions for graph-coloring problems with constraint costs ranging from 1 to 10000, as shown in Figure 6. (The number of cycles is a measure of the runtime that takes into account that the vertices can process information in parallel [10]. A smaller number of cycles implies a smaller runtime.) Heuristics speed up both IDB-ADOPT and ADOPT but the number of cycles needed by IDB-ADOPT with uninformed heuristics is already smaller than the one needed by ADOPT with informed heuristics. The speed up of informed IDB-ADOPT over informed ADOPT tends to increase with the number of vertices, as shown in Figure 7. IDB-ADOPT is 88.7 percent faster than ADOPT when the number of vertices is 10. That is, IDB-ADOPT speeds up ADOPT by a factor of about 9 in this case, which is about one order of magnitude.

In Experiment 2, we measured the speed up of informed IDB-ADOPT over informed ADOPT for finding optimal solutions for graph-coloring problems with 10 vertices. The speed up tends to increase with the range of constraint costs, as shown in Figure 8. IDB-ADOPT is 36.0 percent slower than ADOPT when the constraint costs range from 1 to 3, but 88.7 percent faster than ADOPT when the constraint costs range

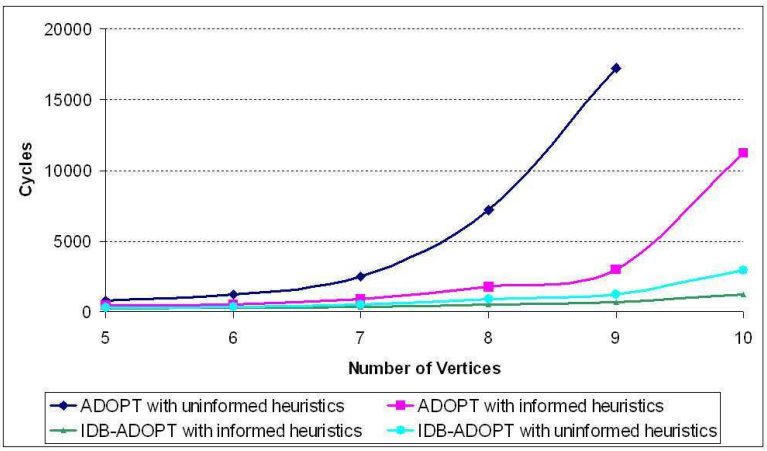


Fig. 6. Experiment 1

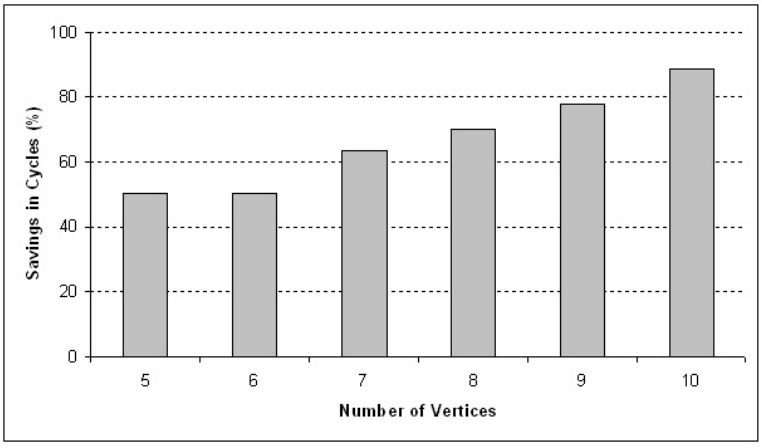


Fig. 7. Speed Ups for Experiment 1

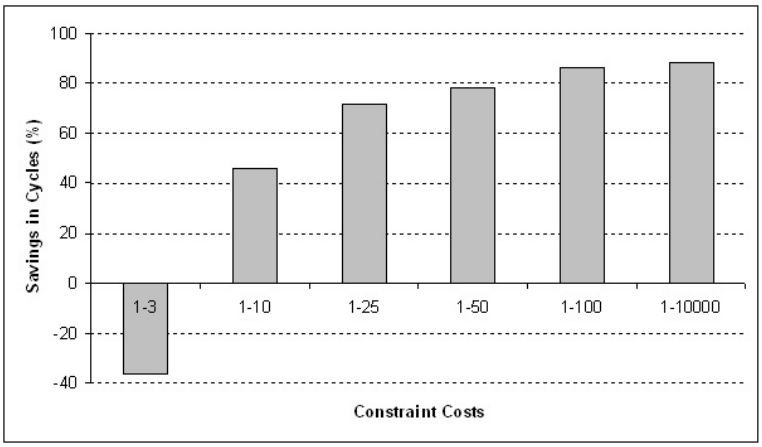


Fig. 8. Speed Ups for Experiment 2

from 1 to 10000 and seems to converge to about this value. The larger the ranges of constraint costs, the more complex the DCOPs and the more misleading the heuristics tend to be, which explains the results.

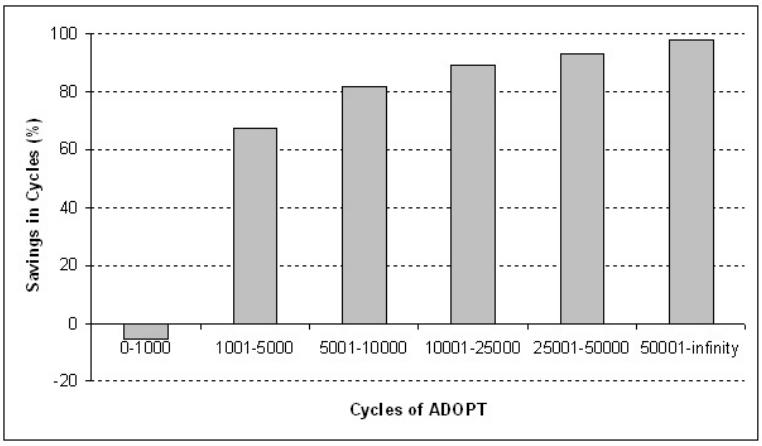


Fig. 9. Speed Ups for Experiment 3

In Experiment 3, we measured the speed up of informed IDB-ADOPT over informed ADOPT for finding optimal solution for graph-coloring problems with 10 vertices and constraint costs ranging from 1 to 10000. We classified them into buckets depending on how many cycles ADOPT needed to solve them: 0-1000, 1001-5000, 5001-10000, 10001-25000, 25001-50000 and 50001-∞ cycles. The speed up tends to in-

crease with the number of cycles ADOPT needed, as shown in Figure 9. IDB-ADOPT is 5.8 percent slower than ADOPT in the bucket 0-1000, but 97.8 percent faster than ADOPT in the bucket 50001-∞ and seems to converge to about this value. Again, the more cycles ADOPT needs, the more complex the DCOPs and the more misleading the heuristics tend to be, which explains the results.

9 Conclusions

In this paper, we introduced Iterative Decreasing Bound ADOPT (IDB-ADOPT), a modification of ADOPT that changes the search strategy of ADOPT from performing one best-first search to performing a series of depth-first searches. IDB-ADOPT is, like ADOPT, an asynchronous and distributed search algorithm that only needs a bounded amount of memory at each vertex and is able to solve DCOPs optimally. Our experimental results for graph coloring problems showed that IDB-ADOPT has smaller cycle counts than ADOPT on large DCOPs, with savings of up to one order of magnitude. In addition, IDB-ADOPT produces suboptimal solutions quickly and then improves them. It can thus be used as an anytime algorithm.

Acknowledgments

This research has been partly supported by an NSF award to Sven Koenig under contract IIS-0350584. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, companies or the U.S. government.

References

1. S. Ali, S. Koenig, and M. Tambe. Preprocessing techniques for accelerating the dcop algorithm adopt. In *AAMAS*, pages 1041–1048, 2005.
2. E. Bowring, M. Tambe, and M. Yokoo. Multiply-constrained distributed constraint optimization. In *AAMAS*, pages 1413–1420, 2006.
3. A. Chechetka and K. Sycara. No-commitment branch and bound search for distributed constraint optimization. In *AAMAS*, pages 1427–1429, 2006.
4. J. Davin and J. Modi. Hierarchical variable ordering for multiagent agreement problems. In *AAMAS*, pages 1433–1435, 2006.
5. R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the Association for Computing Machinery*, 32(3):505–536, 1985.
6. R. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.
7. V. Lesser, C. Ortiz, and M. Tambe, editors. *Distributed sensor networks: A multiagent perspective*. Kluwer, 2003.
8. R. Maheswaran, M. Tambe, E. Bowring, J. Pearce, and P. Varakantham. Taking DCOP to the real world: Efficient complete solutions for distributed event scheduling. In *AAMAS*, pages 310–317, 2004.
9. R. Mailler and V. Lesser. Solving distributed constraint optimization problems using cooperative mediation. In *AAMAS*, pages 438–445, 2004.

10. P. Modi, W. Shen, M. Tambe, and M. Yokoo. ADOPT: asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161:149–180, 2005.
11. F. Pecora, J. Modi, and P. Scerri. Reasoning about and dynamically posting n-ary constraints in adopt. In *DCR*, 2006.
12. A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. In *IJCAI*, pages 1413–1420, 2005.
13. P. Scerri, J. Modi, M. Tambe, and W. Shen. Are multiagent algorithms relevant for real hardware? A case study of distributed constraint algorithms. In *ACM Symposium on Applied Computing*, pages 38–44, 2003.
14. N. Schurr, S. Okamoto, R. Maheswaran, P. Scerri, and M. Tambe. Evolution of a teamwork model. In R. Sun, editor, *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation*, pages 307–327. Cambridge University Press, 2005.
15. W. Zhang and R. Korf. Performance of linear-space search algorithms. *Artificial Intelligence*, 79(2):241–292, 1995.
16. S. Zilberstein. *Operational Rationality through Compilation of Anytime Algorithms*. PhD thesis, Computer Science Department, University of California at Berkeley, Berkeley (California), 1993.

Lower Bounds on the Quality of k-optimal DCOP Solutions with Respect to the Global Optimum

Jonathan P. Pearce and Milind Tambe

Computer Science Department
University of Southern California
Los Angeles, CA 90089
{jppearce, tambe}@usc.edu

Abstract. A distributed constraint optimization problem (DCOP) is a formalism that captures the rewards and costs of local interactions within a team of agents. Because complete algorithms to solve DCOPs are unsuitable for some dynamic or anytime domains, researchers have explored incomplete DCOP algorithms that result in locally optimal solutions. One type of categorization of such algorithms, and the solutions they produce, is k-optimality; a k-optimal solution is one that cannot be improved by any deviation by k or fewer agents. This paper presents the first known guarantees on solution quality for k-optimal solutions. The guarantees are independent of the costs and rewards in the DCOP, and once computed can be used for any DCOP of a given constraint graph structure.

1 Introduction

In a large class of multi-agent scenarios, a set of agents chooses a joint action as a combination of individual actions. Often, the locality of agents' interactions means that the utility generated by each agent's action depends only on the actions of a subset of the other agents. In this case, the outcomes of possible joint actions can be compactly represented in cooperative domains by a distributed constraint optimization problem (DCOP) [1, 2]. A DCOP can take the form of a graph in which each node is an agent and each edge denotes a subset of agents whose actions, taken together, incur costs or rewards to the agent team. Applications of DCOP include sensor networks [1], meeting scheduling [3] and RoboCup soccer [4].

Globally optimal DCOP algorithms can incur large computation or communication costs for domains where the number of agents is large or where time is limited. However, incomplete algorithms in which agents react on the basis of local knowledge of neighbors and constraint utilities can lead to a system that scales up easily and is more robust to dynamic environments. Researchers have introduced k-optimal algorithms in which small groups of agents optimize based on their local constraints, resulting in a k-optimal DCOP assignment, in which no subset of k or fewer agents can improve the overall solution. Some examples include the 1-optimal algorithms DBA [5] and DSA [6] for distributed constraint satisfaction problems (DisCSPs), which were later extended to DCOPs [2], as well as the 2-optimal algorithms in [7], in which optimization was done by agents acting in pairs. Previous work has focused on upper bounds

on the number of k-optima in DCOPs [8], as well as experimental analysis of k-optimal algorithms [2, 7].

Unfortunately, the lack of theoretical guarantees on the quality of solutions obtained by k-optimal algorithms was a fundamental limitation; until now, we could not guarantee a lower bound on the quality of the solution obtained with respect to the quality of the global optimum. In this paper, we introduce such guarantees. These guarantees can help determine an appropriate k-optimal algorithm, or possibly an appropriate constraint graph structure, for agents to use in situations where the cost of coordination between agents must be weighed against the quality of the solution reached. If increasing the value of k will provide a large increase in guaranteed solution quality, it may be worth the extra computation or communication required to reach a higher k-optimal solution. For example, consider a team of autonomous underwater vehicles (AUVs) [9] that must quickly choose a joint action in order to observe some transitory underwater phenomenon. The combination of individual actions by nearby AUVs may generate costs or rewards to the team, and the overall utility of the joint action is determined by their sum. If this problem were represented as a DCOP, nearby AUVs would share constraints in the graph, while far-away AUVs would not. However, the actual rewards on these constraints may not be known until the AUVs are deployed, and in addition, due to time constraints, an incomplete, k-optimal algorithm, rather than a complete algorithm, must be used to find a solution. In this case, worst-case quality guarantees for k-optimal solutions for a given k, that are independent of the actual costs and rewards in the DCOP, are useful to help decide which algorithm to use. Alternatively, the guarantees can help to choose between different AUV formations, i. e. different constraint graphs.

We present guarantees in Sections 3 and 4, as a lower bound on the quality of any k-optimum, expressed as a fraction of the quality of the optimal solution. We provide general bounds that apply to all constraint graph structures, as well as tighter bounds made possible if the graph is known in advance.

2 DCOP and k-optima

We consider a DCOP in which each agent controls a variable to which it must assign a value. Constraints exist on subsets of these variables; each constraint generates a cost or reward to the team based on the values assigned to each variable in the corresponding subset. Although we assume in this paper that each agent controls a single variable, all results are valid for cases in which agents control more than one variable.

Formally, a DCOP is a set of variables (one per agent) $N := \{1, \ldots, n\}$ and a set of domains $\mathcal{A} := \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$, where the i^{th} variable takes value $a_i \in \mathcal{A}_i$. We denote the assignment of the multi-agent team by $a = [a_1 \cdots a_n]$. Valued constraints exist on various subsets $S \subset N$ of these variables. A constraint on S is expressed as a reward function $R_S(a)$. This function represents the reward generated by the constraint on S when the agents take assignment a; costs are expressed as negative rewards. θ is the set of all such subsets S on which a constraint exists, and no $S \in \theta$ is a subset of any other $S \in \theta$. For convenience, we will refer to these subsets S as "constraints" and the functions $R_S(\cdot)$ as "constraint reward functions." The solution quality for a particular

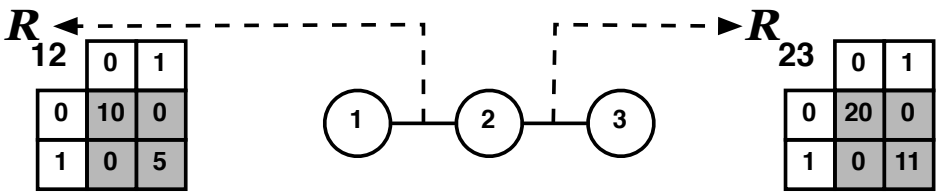


Fig. 1. DCOP example

complete assignment a is the sum of the rewards for that assignment from all constraints in the DCOP:

$$R(a) = \sum_{S \in \theta} R_S(a).$$

In [8], the *deviating group* between two assignments, a and $\tilde{a}$, was defined as

$$D(a, \tilde{a}) := \{i \in N : a_i \neq \tilde{a}_i\},$$

i.e. the set of variables whose values in $\tilde{a}$ differ from their values in a. The *distance* between two assignments was defined as

$$d(a, \tilde{a}) := |D(a, \tilde{a})|$$

where $|\cdot|$ denotes the size of the set. An assignment a is classified as a *k-optimum* if

$$R(a) - R(\tilde{a}) \geq 0 \ \forall \tilde{a} \text{ such that } d(a, \tilde{a}) \leq k.$$

Equivalently, at a k-optimum, no subset of k or fewer agents can improve the overall reward by choosing different values; every such subset is acting optimally given the values of the others.

Example 1. Figure 1 is a binary DCOP in which agents choose values from $\{0, 1\}$, with constraints $S_1 = \{1, 2\}$ and $S_2 = \{2, 3\}$ with rewards shown. The assignment a = [1 1 1] is 1-optimal because any single agent that deviates reduces the team reward. However, [1 1 1] is not 2-optimal because if the group $\{2, 3\}$ deviated, making the assignment $\tilde{a}$ = [1 0 0], team reward would increase from 16 to 20. The globally optimal solution, a^* = [0 0 0] is k-optimal for all $k \in \{1, 2, 3\}$.□

In addition to categorizing local optima, k-optimality provides a natural classification for DCOP algorithms. Many algorithms are guaranteed to converge to k-optima, including DBA [2], DSA [6], and coordinate ascent [4] for $k = 1$, and MGM-2 and SCA-2 [7] for $k = 2$. Globally optimal algorithms such as Adopt [1], OptAPO [10] and DPOP [3] converge to a k-optimum for $k = n$.

3 Quality guarantees on *k*-optima

This section provides reward-independent guarantees on solution quality for any k-optimal DCOP assignment. If we must choose a k-optimal algorithm for agents to use, it is useful to see how much reward will be gained or lost in the worst case by choosing a higher or lower value for k. We assume the actual costs and rewards on the DCOP are not known *a priori* (otherwise the DCOP could be solved centrally ahead of time). We provide a guarantee for a k-optimal solution as a fraction of the reward of the optimal solution, assuming that all rewards in the DCOP are non-negative (the reward structure of any DCOP can be normalized to one with all non-negative rewards as long as no infinitely large costs exist).

Proposition 1. *For any DCOP of n agents, with maximum constraint arity of m, where all constraint rewards are non-negative, and where a^* is the globally optimal solution, then, for any k-optimal assignment, a, where $m \leq k < n$,*

$$R(a) \geq \frac{\binom{n-m}{k-m}}{\binom{n}{k} - \binom{n-m}{k}} R(a^*). \tag{1}$$

Proof: By the definition of k-optimality, any assignment $\tilde{a}$ such that $d(a, \tilde{a}) \leq k$ must have reward $R(\tilde{a}) \leq R(a)$. We call this set of assignments $\tilde{A}$. Now consider any non-null subset $\hat{A} \subset \tilde{A}$. For any assignment $\hat{a} \in \hat{A}$, the constraints θ in the DCOP can be divided into three discrete sets, given a and $\hat{a}$:

- $\theta_1(a, \hat{a}) \subset \theta$ such that $\forall S \in \theta_1(a, \hat{a}), S \subset D(a, \hat{a})$.
- $\theta_2(a, \hat{a}) \subset \theta$ s.t. $\forall S \in \theta_2(a, \hat{a}), S \cap D(a, \hat{a}) = \emptyset$.
- $\theta_3(a, \hat{a}) \subset \theta$ s.t. $\forall S \in \theta_3(a, \hat{a}), S \notin \theta_1(a, \hat{a}) \cup \theta_2(a, \hat{a})$.

$\theta_1(a, \hat{a})$ contains the constraints that include only the variables in $\hat{a}$ which have deviated from their values in a; $\theta_2(a, \hat{a})$ contains the constraints that include only the variables in $\hat{a}$ which have not deviated from a; and $\theta_3(a, \hat{a})$ contains the constraints that include at least one of each. Thus:

$$R(\hat{a}) = \sum_{S \in \theta_1(a,\hat{a})} R_S(\hat{a}) + \sum_{S \in \theta_2(a,\hat{a})} R_S(\hat{a}) + \sum_{S \in \theta_3(a,\hat{a})} R_S(\hat{a}).$$

And, the sum of rewards of all assignments $\hat{a}$ in $\hat{A}$ is:

$$\begin{aligned} \sum_{\hat{a} \in \hat{A}} R(\hat{a}) &= \sum_{\hat{a} \in \hat{A}} \Big(\sum_{S \in \theta_1(a,\hat{a})} R_S(\hat{a}) + \sum_{S \in \theta_2(a,\hat{a})} R_S(\hat{a}) + \sum_{S \in \theta_3(a,\hat{a})} R_S(\hat{a}) \Big) \\ &\geq \sum_{\hat{a} \in \hat{A}} \sum_{S \in \theta_1(a,\hat{a})} R_S(\hat{a}) + \sum_{\hat{a} \in \hat{A}} \sum_{S \in \theta_2(a,\hat{a})} R_S(\hat{a}). \end{aligned}$$

Since $R(a) > R(\hat{a}), \forall \hat{a} \in \hat{A}$,

$$R(a) \geq \frac{\sum_{\hat{a} \in \hat{A}} \sum_{S \in \theta_1(a,\hat{a})} R_S(\hat{a}) + \sum_{\hat{a} \in \hat{A}} \sum_{S \in \theta_2(a,\hat{a})} R_S(\hat{a})}{|\hat{A}|}. \tag{2}$$

Now, if the two numerator terms and the denominator can be expressed in terms of $R(a^*)$ and $R(a)$, then we have a bound on $R(a)$ in terms of $R(a^*)$. To do this, we consider the particular $\hat{A}$ which contains all assignments $\hat{a}$ such that:

- $d(a, \hat{a}) = k$, and
- $\forall \hat{a} \in \hat{A}, \forall \hat{a}_i \in D(a, \hat{a}), \hat{a}_i = a_i^*$. This means that exactly k variables in $\hat{a}$ have deviated from their value in a, and these variables are taking the same values that they had in a^*.

There are $\binom{d(a,a^*)}{k}$ assignments $\hat{a} \in \hat{A}$. For *every* constraint $S \in \theta$, there are exactly $\binom{d(a,a^*)-|S|}{k-|S|}$ different assignments $\hat{a} \in \hat{A}$ for which $S \in \theta_1(a, \hat{a})$. This is because there exists a unique $\hat{a} \in \hat{A}$ for every subset of k variables in $D(a, a^*)$. If $S \subset D(a, \hat{a})$, as stipulated by the definition of $\theta_1(a, \hat{a})$, then there are $d(a, a^*) - |S|$ remaining variables from which $k - |S|$ must be chosen to complete $D(a, \hat{a})$, and so there are $\binom{d(a,a^*)-|S|}{k-|S|}$ possible assignments $\hat{a}$ for which this is true. For all $\hat{a}$, for all $S \in \theta_1(a, \hat{a})$, $R_S(\hat{a}) = R_S(a^*)$, so

$$\sum_{\hat{a} \in \hat{A}} \sum_{S \in \theta_1(a,\hat{a})} R_S(\hat{a}) = \sum_{S \in \theta} \binom{d(a, a^*) - |S|}{k - |S|} R_S(a^*) \geq \binom{d(a, a^*) - m}{k - m} R(a^*).$$

Similarly, for every constraint $S \in \theta$, there are $\binom{d(a,a^*)-|S|}{k}$ different assignments $\hat{a} \in \hat{A}$ for which $S \in \theta_2(a, \hat{a})$. If $S \cap D(a, \hat{a}) = \emptyset$, as stipulated by the definition of $\theta_2(a, \hat{a})$, then there are $d(a, a^*) - |S|$ remaining variables from which k must be chosen to complete $D(a, \hat{a})$, and so there are $\binom{d(a,a^*)-|S|}{k}$ possible assignments $\hat{a}$ for which this is true. For all $\hat{a}$, for all $S \in \theta_2(a, \hat{a}), R_S(\hat{a}) = R_S(a)$, and so

$$\sum_{\hat{a} \in \hat{A}} \sum_{S \in \theta_2(a,\hat{a})} R_S(\hat{a}) = \sum_{S \in \theta} \binom{d(a, a^*) - |S|}{k} R_S(a) \geq \binom{d(a, a^*) - m}{k} R(a).$$

Therefore, from Equation 2,

$$\begin{aligned} R(a) &\geq \frac{\binom{d(a,a^*)-m}{k-m} R(a^*) + \binom{d(a,a^*)-m}{k} R(a)}{\binom{d(a,a^*)}{k}} \\ &\geq \frac{\binom{d(a,a^*)-m}{k-m}}{\binom{d(a,a^*)}{k} - \binom{d(a,a^*)-m}{k}} R(a^*) \end{aligned}$$

which is minimized when $d(a, a^*) = n$, so Equation 1 holds as a guarantee for a k-optimum in any DCOP. It is possible that $k > n - m$; in this case we take $\binom{n-m}{k}$ to be 0. ■

For binary DCOPs ($m = 2$), Equation 1 simplifies to:

$$R(a) \geq \frac{(k-1)}{(2n-k-1)} R(a^*).$$

The following example illustrates Proposition 1:

Example 2. Consider a DCOP with five variables numbered 1 to 5, with domains of {0,1}, fully connected with binary constraints between all variable pairs. Suppose that a = [0 0 0 0 0] is a 3-optimum, and that a^* = [1 1 1 1 1] is the global optimum. Then $d(a, a^*) = 5$, and $\hat{A}$ contains $\binom{d(a,a^*)}{k} = 10$ assignments: [1 1 1 0 0], [1 1 0 1 0], [1 1 0 0 1], [1 0 1 1 0], [1 0 1 0 1], [1 0 0 1 1], [0 1 1 1 0], [0 1 1 0 1], [0 1 0 1 1], [0 0 1 1 1]. Whatever the values of the rewards are, every constraint reward $R_S(a^*)$ will equal $R_S(\hat{a})$ for $\binom{n-2}{k-2} = 3$ assignments in $\hat{A}$ (e.g. $R_{\{1,2\}}(a^*) = R_{\{1,2\}}(\hat{a})$ for $\hat{a}$ = [1 1 1 0 0], [1 1 0 1 0], and [1 1 0 0 1]) and similarly, every constraint reward $R_S(a)$ equals $R_S(\hat{a})$ for $\binom{n-2}{k} = 1$ assignment in $\hat{A}$. Thus, $R(a) \geq \frac{3}{10-1} R(a^*) = \frac{1}{3} R(a^*)$.□

We now show that Proposition 1 is tight, i.e. that there exist DCOPs with k-optima of quality equal to the bound.

Proposition 2. *$\forall n, m, k$ such that $m \leq k < n$, there exists some DCOP with n variables, with maximum constraint arity m with a k-optimal assignment, a, such that, if a^* is the globally optimal solution,*

$$R(a) = \frac{\binom{n-m}{k-m}}{\binom{n}{k} - \binom{n-m}{k}} R(a^*) \tag{3}$$

Proof: Consider a fully-connected m-ary DCOP where the domain of each variable contains at least two values {0,1} and every constraint R_S contains the following reward function:

$$R_S(a) = \begin{cases} \frac{\binom{n-m}{k-m}}{\binom{n}{k} - \binom{n-m}{k}} & , \forall i \in S, a_i = 0 \\ 1 & , \forall i \in S, a_i = 1 \\ 0 & , \text{otherwise} \end{cases}$$

The optimal solution a^* is $a_i^* = 1, \forall i$. If a is defined such that $a_i = 0, \forall i$, then Equation 3 is true. Now we show that a is k-optimal. For any assignment $\hat{a}$, such that $d(a, \hat{a}) = k$,

$$
\begin{aligned}
R(\hat{a}) &= \sum_{S \in \theta_1(a,\hat{a})} R(\hat{a}_S) + \sum_{S \in \theta_2(a,\hat{a})} R(\hat{a}_S) + \sum_{S \in \theta_3(a,\hat{a})} R(\hat{a}_S). \\
&\leq \binom{k}{m} + \binom{n-k}{m} \frac{\binom{n-m}{k-m}}{\binom{n}{k} - \binom{n-m}{k}} + 0 \\
&= \frac{\binom{k}{m}\left[\binom{n}{k} - \binom{n-m}{k}\right] + \binom{n-k}{m}\binom{n-m}{k-m}}{\binom{n}{k} - \binom{n-m}{k}} \\
&= \frac{n!}{m!(k-m)!(n-k)!} \div \frac{n!(n-m-k)! - (n-m)!(n-k)!}{k!(n-k)!(n-m-k)!} \\
&= \frac{n!k!(n-m-k)!}{m!(k-m)![n!(n-m-k)! - (n-k)!(n-m)!]} \\
&= \binom{n}{m} \frac{(n-m)!k!(n-m-k)!}{(k-m)![n!(n-m-k)! - (n-m)!(n-k)!]} \\
&= \binom{n}{m} \frac{\binom{n-m}{k-m}k!(n-m-k)!}{\frac{n!(n-m-k)!}{(n-k)!} - (n-m)!} \\
&= \binom{n}{m} \frac{\binom{n-m}{k-m}}{\binom{n}{k} - \binom{n-m}{k}} \\
&= R(a)
\end{aligned}
$$

because in a, each of the $\binom{n}{m}$ constraints in the DCOP are producing the same reward. Since this can be shown for $d(a, \hat{a}) = j, \forall\, j$ such that $1 \leq j \leq k$, a is k-optimal. ■

4 Graph-based quality guarantees

The guarantee for k-optima in Section 3 applies to all possible DCOP graph structures. However, knowledge of the structure of constraint graphs can be used to obtain tighter guarantees. This is done by again expressing the two numerator terms in Equation 2 as multiples of $R(a^*)$ and $R(a)$. However, for a sparse graph, if $\hat{A}$ is chosen as defined in Proposition 1, there may be many assignments $\hat{a} \in \hat{A}$ that have few or no constraints S in $\theta_1(a, \hat{a})$ because the variables in $D(a, \hat{a})$ may not share any constraints. Instead, exploiting the graph structure by choosing a smaller $\hat{A}$ can lead to a tighter bound. We can take $\hat{A}$ from Proposition 1, i.e. $\hat{A}$ which contains all $\hat{a}$ such that $d(a, \hat{a}) = k$ and $\forall \hat{a} \in \hat{A}, \forall \hat{a}_i \in D(a, \hat{a}), \hat{a}_i = a^*_i$. Then, we restrict this $\hat{A}$ further, so that $\forall \hat{a} \in \hat{A}$, the variables in $D(a, \hat{a})$ form a connected subgraph of the DCOP graph (or hypergraph), meaning that any two variables in $D(a, \hat{a})$ must be connected by some chain of constraints. This allows us to again transform Equation 2 to express $R(a)$ in terms of $R(a^*)$; this new method can produce tighter guarantees for k-optima in sparse graphs. As an illustration, provably tight guarantees for binary DCOPs on ring graphs (each variable has two constraints) and star graphs (each variable has one constraint except the central variable, which has $n - 1$) are given below.

Proposition 3. *For any binary DCOP of n agents with a ring graph structure, where all constraint rewards are non-negative, and a^* is the globally optimal solution, then, for any k-optimal assignment, a, where $k < n$,*

$$R(a) \geq \frac{k-1}{k+1} R(a^*). \tag{4}$$

Proof: Returning to Equation 2, $|\hat{A}| = n$ because $D(a, \hat{a})$ could consist of any of the n connected subgraphs of k variables in a ring. For any constraint $S \in \theta$, there are $k-1$ assignments $\hat{a} \in \hat{A}$ for which $S \in \theta_1(a, \hat{a})$ because there are $k-1$ connected subgraphs of k variables in a ring that contain S. Therefore,

$$\sum_{\hat{a} \in \hat{A}} \sum_{S \in \theta_1(a,\hat{a})} R_S(\hat{a}) = (k-1) R(a^*).$$

Also, there are $n-k-1$ assignments $\hat{a} \in \hat{A}$ for which $S \in \theta_2(a, \hat{a})$ because there are $n-k-1$ ways to choose S in a ring so that it does not include any variable in a given connected subgraph of k variables. Therefore,

$$\sum_{\hat{a} \in \hat{A}} \sum_{S \in \theta_2(a,\hat{a})} R_S(\hat{a}) = (n-k-1) R(a).$$

So, from Equation 2,

$$R(a) \geq \frac{(k-1)R(a^*) + (n-k-1)R(a)}{n}$$

and therefore Equation 4 holds. ■

Proposition 4. *For any binary DCOP of n agents with a star graph structure, where all constraint rewards are non-negative, and a^* is the globally optimal solution, then, for any k-optimal assignment, a, where $k < n$,*

$$R(a) \geq \frac{k-1}{n-1} R(a^*). \tag{5}$$

Proof: The proof is similar to the previous proof. In a star graph, there are $\binom{n-1}{k-1}$ subgraphs of k variables, and therefore $|\hat{A}| = \binom{n-1}{k-1}$. Every constraint $S \in \theta$ includes the central variable and one other variable, and thus there are $\binom{n-2}{k-2}$ connected subgraphs of k variables that contain S, and therefore

$$\sum_{\hat{a} \in \hat{A}} \sum_{S \in \theta_1(a,\hat{a})} R_S(\hat{a}) = \binom{n-2}{k-2} R(a^*).$$

Finally, there are no ways to choose S so that it does not include any variable in a given connected subgraph of k variables. Therefore,

$$\sum_{\hat{a} \in \hat{A}} \sum_{S \in \theta_2(a,\hat{a})} R_S(\hat{a}) = 0 R(a).$$

So, from Equation 2,

$$R(a) \geq \frac{\binom{n-2}{k-2}R(a^*) + 0R(a)}{\binom{n-1}{k-1}}$$

and therefore Equation 5 holds. ■

Tightness can be proven by constructing DCOPs on ring and chain graphs with the same rewards as in Proposition 2; proofs are omitted for space. The bound for rings can also be applied to chains, since any chain can be expressed as a ring where all rewards on one constraint are zero.

Finally, bounds for DCOPs with arbitrary graphs and non-negative constraint rewards can be found using a linear-fractional program (LFP). This method gives a tight bound for any graph, since it instantiates the rewards for all constraints, but requires a globally optimal solution to the LFP, in contrast to the constant-time guarantees of Equations 1, 4 and 5. An LFP such as this is reducible to a linear program (LP) [11]. The objective is to minimize $\frac{R(a)}{R(a^*)}$ such that $\forall \tilde{a} \in \tilde{A}, R(a) - R(\tilde{a}) \geq 0$, given $\tilde{A}$ as defined in Proposition 1. Note that $R(a^*)$ and $R(a)$ can be expressed as $\sum_{S \in \theta} R_S(a^*)$ and $\sum_{S \in \theta} R_S(a^*)$. We can now transform the DCOP so that every $R(\tilde{a})$ can also be expressed in terms of sums of $R_S(a^*)$ and $R_S(a)$, without changing or invalidating the guarantee on $R(a)$. Therefore, the LFP will contain only two variables for each $S \in \theta$, one for $R_S(a^*)$ and one for $R_S(a)$, where the domain of each one is the set of non-negative real numbers. The transformation is to set all reward functions $R_S(\cdot)$ for all $S \in \theta$ to 0, except for two cases: when all variables $i \in S$ have the same value as in a^*, or when all $i \in S$ have the same value as in a. This has no effect on $R(a^*)$ or $R(a)$, because $R_S(a^*)$ and $R_S(a)$ will be unchanged for all $S \in \theta$. It also has no effect on the optimality of a^* or the k-optimality of a, since the only change is to reduce the global reward for assignments other than a^* and a. Thus, the tight lower bound on $\frac{R(a)}{R(a^*)}$ still applies to the original DCOP.

5 Experimental results

While the main thrust of this paper is on theoretical guarantees for k-optima, this section gives an illustration of the guarantees in action, and how they are affected by constraint graph structure. Figures 2a, 2b, and 2c show quality guarantees for binary DCOPs with fully connected graphs, ring graphs, and star graphs, calculated directly from Equations 1, 4 and 5.

Figure 2d shows quality guarantees for binary-tree DCOPs, obtained using the LFP from Section 4. The x-axis plots the value chosen for k, and the y-axis plots the lower bound for k-optima as a percentage of the optimal solution quality for systems of 5, 10, 15, and 20 agents. These results show how the worst-case benefit of increasing k varies depending on graph structure. For example, in a five-agent DCOP, a 3-optimum is guaranteed to be 50% of optimal whether the graph is a star or a ring. However, moving to $k = 4$ means that worst-case solution quality will improve to 75% for a star, but only to 60% for a ring. For fully connected graphs, the benefit of increasing k goes up as k increases; whereas for stars it stays constant, and for chains it decreases, except for when $k = n$. Results for binary trees are mixed.

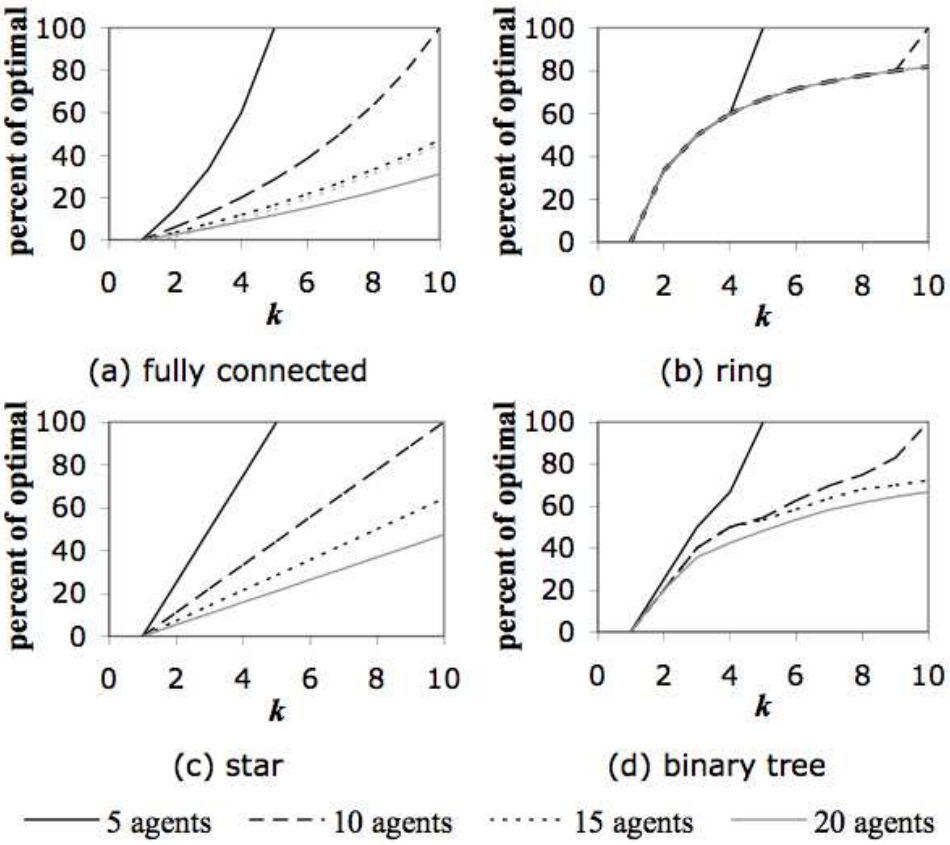


Fig. 2. Quality guarantees for k-optima with respect to the global optimum for DCOPs of various graph structures.

6 Related work and conclusion

This paper contains the first guarantees on solution quality for k-optimal DCOP assignments. The performance of any local DCOP algorithms can now be compared in terms of worst case guaranteed solution quality, either on a particular constraint graph, or over all possible graphs. In addition, since the guarantees are reward-independent, they can be used for any DCOP of a given graph structure, once computed.

In [8], upper bounds on the number of possible k-optima that could exist in a given DCOP graph were presented. The work in this paper focuses instead on lower bounds on solution quality for k-optima for a given DCOP graph. This paper provides a complement to the experimental analysis of local optima (1-optima) arising from the execution of incomplete DCOP algorithms [2, 7]. However, in this paper, the emphasis is on the worst case rather than the average case.

The results in this paper can help illuminate the relationship between local and global optimality in many types of multi-agent systems, e. g. networked distributed POMDPs [13]. All results in this paper also apply to centralized constraint reasoning. However, examining properties of solutions that arise from coordinated value changes of small groups of variables is especially useful in distributed settings, given the computational and communication expense of large-scale coordination.

References

1. Modi, P.J., Shen, W., Tambe, M., Yokoo, M.: Adopt: Asynchronous distributed constraint optimization with quality guarantees. Artificial Intelligence **161**(1-2) (2005) 149–180
2. Zhang, W., Wang, G., Xing, Z., Wittenberg, L.: Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks. Artificial Intelligence **161**(1-2) (2005) 55–87
3. Petcu, A., Faltings, B.: A scalable method for multiagent constraint optimization. In: IJCAI. (2005)
4. Vlassis, N., Elhorst, R., Kok, J.R.: Anytime algorithms for multiagent decision making using coordination graphs. In: Proc. Intl. Conf. on Systems, Man and Cybernetics. (2004)
5. Yokoo, M., Hirayama, K.: Distributed breakout algorithm for solving distributed constraint satisfaction and optimization problems. In: ICMAS. (1996)
6. Fitzpatrick, S., Meertens, L.: Distributed coordination through anarchic optimization. In Lesser, V., Ortiz, C.L., Tambe, M., eds.: Distributed Sensor Networks: A Multiagent Perspective. Kluwer (2003) 257–295
7. Maheswaran, R.T., Pearce, J.P., Tambe, M.: Distributed algorithms for DCOP: A graphical-game-based approach. In: PDCS. (2004)
8. Pearce, J.P., Maheswaran, R.T., Tambe, M.: Solution sets for DCOPs and graphical games. In: AAMAS. (2006)
9. Zhang, Y., Bellingham, J.G., Davis, R.E., Chao, Y.: Optimizing autonomous underwater vehicles' survey for reconstruction of an ocean field that varies in space and time. In: American Geophysical Union, Fall meeting. (2005)
10. Mailler, R., Lesser, V.: Solving distributed constraint optimization problems using cooperative mediation. In: AAMAS. (2004)
11. Boyd, S., Vandenberghe, L.: Convex Optimization. Cambridge U. Press (2004)

12. Gutin, G., Yeo, A.: Domination analysis of combinatorial optimization algorithms and problems. In Golumbic, M., Hartman, I., eds.: Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications. Kluwer (2005)
13. Nair, R., Varakantham, P., Tambe, M., Yokoo, M.: Networked distributed POMDPs: A synthesis of distributed constraint optimization and POMDPs. In: AAAI. (2005)

Discussion on the Three Backjumping Schemes Existing in ADOPT-ng

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

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

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

1 Introduction

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

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

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

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

2 Distributed Valued CSPs

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

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

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

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

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

3 ADOPT with nogoods

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

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

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

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

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

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

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

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

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

3.1 Exploiting DFS trees for Feedback

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

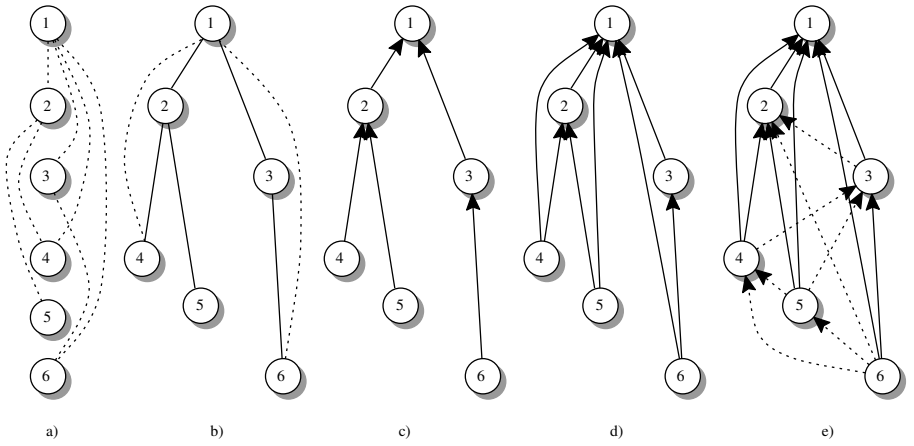


Fig. 1. Feedback modes in ADOPT-ng. a) a constraint graph on a totally ordered set of agents; b) a DFS tree compatible with the given total order; c) ADOPT-p__: sending valued nogoods only to parent (graph-based backjumping); d) ADOPT-d__ and ADOPT-D__: sending valued nogoods to any ancestor in the tree;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

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

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

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

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

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

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

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

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

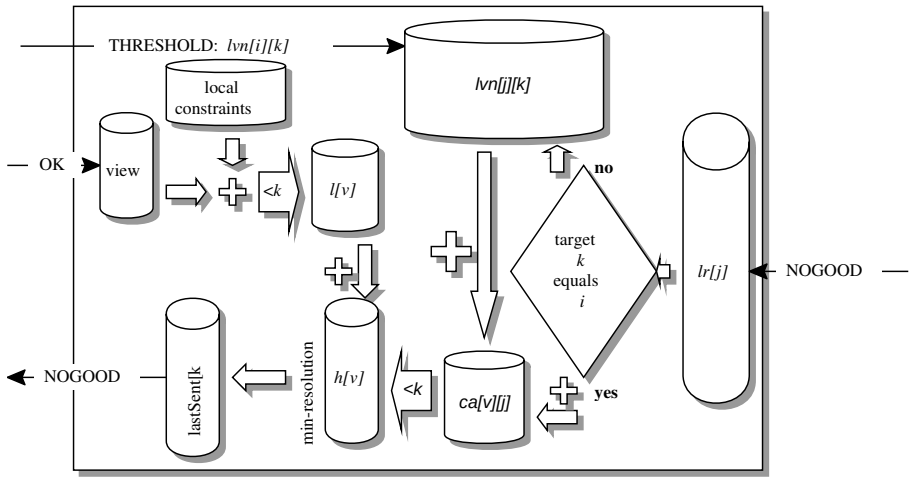


Fig. 2. Schematic flow of data through the different data structures used by an agent A_i in ADOPT-ng.

3.3 Data Structures

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

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

The names of the structures were chosen by following the relation of ADOPT with A* search [13]. Thus, h stands for the "heuristic" estimation of the cost due to constraints maintained by future agents (equivalent to the $h()$ function in A*) and l stands for the part of the standard $g()$ function of A* that is "local" to the current agent. Here, as in ADOPT, the value for $h()$ is estimated by aggregating the equivalent of costs received from lower priority agents. Since the costs due to constraints of higher priority agents are identical for each value,

they are irrelevant for the decisions of the current agent. Thus, the function $f()$ of this version of A* is computed combining solely l and h. We currently store the result of combining h and l in h itself to avoid allocating a new structure for $f()$.

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

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

3.4 Data flow in ADOPT-ng

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

The pseudocode is described in Algorithm 1. The $min\_resolution(j)$ function applies the min-resolution over the CAs associated to all the values of the variable of the current agent, but uses only CAs having no assignment from agents with lower priority than A_j. More exactly it first re-computes the array h using only CAs in ca and l that contain only assignments from $A_1, ..., A_j$, and then applies min-resolution over the obtained elements of h. As mentioned above, in the current implementation we recompute l and h at each call to $min\_resolution(j)$, and such a call is separately performed for each ancestor agent A_j.

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

Algorithm 1: Receiving messages of A_i in ADOPT-ng

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

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

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

3.5 Optimizing valued nogoods

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

3.6 Example

Now we give a detailed example of a run of ADOPT-ng basic versions ADOPT-aos and ADOPT-Aos. Let us take the problem in Figure 3. Note that in this simple case the two versions do not differ since any optional nogood message can only leave from A_3 to A_1. Such a message is sent in ADOPT-aos only if it has a non-zero threshold, which happens only when A_1 is a target of the message, which means that it will also be sent in ADOPT-Aos. A trace is shown in Figure 4 where identical messages sent simultaneously to several agents are grouped by displaying the list of recipients. The agents start selecting values for

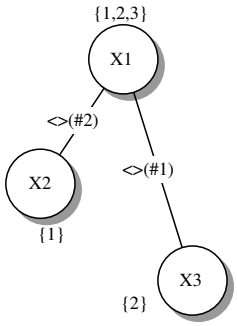


Fig. 3. A DisCOP with three agents and two inequality constraints. The fact that the cost associated with not satisfying the constraint $x_1 \neq x_2$ is 2, is denoted by the notation (#2). The cost for not satisfying the constraint $x_1 \neq x_3$ is 1.

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

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

their variables and announce them to interested lower priority agents. A_3 has no constraint between x_3 and x_2; therefore the first exchanged messages are **ok?** messages sent by A_1 to both successors A_2 and A_3 and proposing the assignment x_1=1.

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

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

Agent A_1 now updates its $h[1]$ by setting it to $ca[1][2]$ (since $l[1]$ and $ca[1][3]$ are empty). Since the threshold of $h[1]$ becomes 2 and is higher than the threshold

of the other two values, {2,3}, in the domain of x_1, A_1 changes the assignment of x_1 to one of them, here 2. This is announced through another **ok?** message to A_2 and A_3.

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

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

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

4 Experiments

The algorithms are compared on the same problems that are used to report ADOPT's performance in [11]. To correctly compare our techniques with the original ADOPT, we have used the same order (or DFS trees) on agents for each problem. The impact of the existence of a good DFS tree compatible with the used order is tested separately by comparison with a random ordering. The set of problems distributed with ADOPT and used here contains 25 problems for each problem size. It contains problems with 8, 10, 12, 14, 16, 18, 20, 25, 30, and 40 agents, and for each of these numbers of agents it contains test sets with density .2 and with density .3. The density of a (binary) constraint problem's graph with n variables is defined by the ratio between the number of binary constraints and $\frac{n(n-1)}{2}$. Results are averaged on the 25 problems with the same parameters.

The length of the longest causal (sequential) chain of messages of each solver, computed as the number of cycles of our simulator and averaged on problems with density .3, is given in Figure 5. Results for problems with density .2 are

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

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

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

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

given in Figure 6. It took more than two weeks for the original ADOPT implementation to solve one of the problems for 20 agents and density .3, and one of the problems for 25 agents and density .3 (at which moment the solver was interrupted). Therefore, it was evaluated using only the remaining 24 problems at those problem sizes.

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

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

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

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

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

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

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

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

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

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

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

Versions using DFS trees require fewer parallel/total messages, being more network friendly, as seen in Figure 7. Figure 7 shows that refraining from sending too many optional nogoods messages, as done in ADOPT-Aos and ADOPT-

Dos, is comparable to ADOPT-pon in terms of total number of messages, while maintaining the efficiency in cycles comparable to ADOPT-aos and ADOPT-dos.

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

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

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

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

5 Conclusions

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

We isolated and evaluated the contribution of using threshold valued nogoods in ADOPT-ng, which was found to be at most 5%. In addition, we determined the importance of precomputing and maintaining a short DFS tree of the constraint graph, or at least of guaranteeing that a DFS tree is compatible with the order on agents, which is almost an order of magnitude in our problems. Choosing a strategy of medium aggressiveness for sending valued nogoods to predecessors

brings slight improvements in terms of length of longest causal chain of messages (measured as number of cycles of the simulator). It brings an order of magnitude improvements in the total number of messages.

References

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

Delegation in Tree-search for Distributed Constraint Satisfaction

Muhammed Basharu[1]*, Ken Brown[1], and Youssef Hamadi[2]

[1] Cork Constraint Computation Centre, University College Cork, Ireland.
mb@4c.ucc.ie, k.brown@cs.ucc.ie
[2] Microsoft Research, 7 J J Thomson Avenue, Cambridge, United Kingdom.
youssefh@microsoft.com

Abstract. We introduce the idea of delegation in distributed tree-search, as a method to reduce the communication overhead when solving Distributed Constraint Satisfaction Problems (DisCSPs). With delegation, an agent can eliminate some direct forward links to child neighbours and choose intermediaries for communicating with such children. We present an algorithm which constructs long delegation paths automatically, and we prove that given certain assumptions it does not decrease privacy. We show experimentally that delegation can reduce messages by 50% for hard problems, although at the expense of more constraint checks.

1 Introduction

Distributed Constraint Satisfaction Problems (DisCSP) [10] are a generalisation of CSPs for tackling decision problems where the processing power and autonomy are naturally distributed - for example, meeting scheduling or sensor networks. Agents maintain local CSPs, which are linked through inter-agent constraints. DisCSPs are generally solved by distributed tree-based search, where a partial order of the agents is used to record the progress of the exploration. In most of these algorithms, agents send local solutions to their children (the set of neighbours lower than them in the ordering). Children in turn solve their local problems to be consistent with the incoming partial solutions. When an agent cannot find a local solution, a distributed backtracking step is started and addressed to a subset of the agent's parents. Within this broad framework, many different approaches are possible, balancing the issues of total run-time, network transmission costs, fair use of resources, and maintenance of agent privacy.

The main decision is whether the search should be synchronous or asynchronous. Synchronised search closely resembles standard non-distributed search processes. Using the tree-ordering, agents pass control up and down the tree, and each agent only operates when it has control. Typically, an agent receives a partial solution for all its ancestor agents, computes its own local extension, and passes the new partial solution onto its children. Backtracking is synchronised similarly. In asynchronous search, all agents

* This work is supported by grants from Microsoft Research, Science Foundation Ireland, and the Embark Initiative of the Irish Research Council of Science Engineering and Technology.

may operate simultaneously, computing their own local solution based on whatever current knowledge they have of the other agents' decisions, and updating those solutions when that knowledge is updated. Asynchronous search tends to have a smaller total runtime, since much computation is done in parallel and dead-ends can be identified early, but at the expense of more network traffic, and possibly redundant chains of computation. Synchronous search reduces the network traffic, but typically has a longer runtime. In addition, privacy can be compromised, since larger partial solutions are passed up and down the tree. The consensus view is that if message passing is relatively more expensive than computation, and privacy is not important, then synchronised search is better; on the other hand, if runtime is important, or privacy is important, then an asynchronous search is better.

Here, we consider the case where message passing is slow, unreliable or expensive, but where privacy is also important. We present a concept called delegation, where some agents may decide to transmit their local solutions through intermediate agents. Specifically, an agent may appoint one of its neighbours to relay messages to a second neighbour. This can be viewed as an implicit form of local synchronisation, although each agent is still free to act asynchronously, and indeed backtracking messages continue to be asynchronous. The intuition is that the second neighbour should receive larger and more coherent partial solutions from the intermediary, and thus should invoke fewer redundant chains of decisions, at the expense of a small delay in receiving the original message. We will show a simple delegation strategy which preserves the privacy level of existing algorithms. We also show that the delegation strategy can reduce the number of messages by approximately 50% for hard problems, but similarly increases the number of constraint checks, and thus is effective in scenarios where the cost of each message is high.

In the following, we start with an overview of the DisCSP formalism. Section 3 defines delegation, and in Section 4, we present an algorithm for performing delegation in advance of a search where links between unconnected agents are added prior to a search. In Section 5, we consider algorithms where new links are created as a search progresses and present a technique for performing delegation for such algorithms. Both Sections 4 and 5 also include results of evaluations of the respective delegation strategies.

2 Background

A DisCSP is a 4-tuple (X,D,C,A) where:

1. X is a set of n variables $X_1, X_2, \ldots, X_n$.
2. D is a set of domains $D_1, D_2, \ldots, D_n$ of possible values for the variables X_1, X_2, ..., X_n respectively.
3. C is a set of constraints on the values of the variables. The constraint C_k (X_{k1}, ..., X_{kj}) is a predicate defined on the Cartesian product $D_{k1} \times \ldots \times D_{kj}$. A constraint is satisfied if the value assignment of these variables satisfies the predicate.
4. A $= \{A_1, A_2, \ldots, A_p\}$is a partition of X among p autonomous processes or agents where each agent A_k "owns" a subset of the variables in X with respect to some mapping function $f : \mathsf{X} \rightarrow \mathsf{A}, s.t. f(X_i) = A_j$.

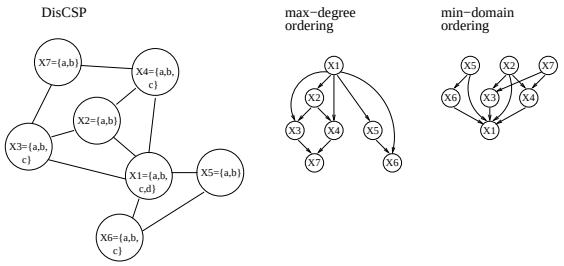


Fig. 1. A DisCSP and two agent orderings.

A solution to a DisCSP is, as for standard CSPs, an assignment to each variable of value from its domain, such that all constraints are satisfied.

In the solving process, we assume that each agent controls its own variables, and, as a default, knows only its own domains and the constraints defined on its variables. The agents must cooperate to find a global solution through message passing. A basic method for finding a global solution uses the distributed backtracking paradigm [8,3]. The agents are prioritized into a partial order $<_o$ such that any two agents are connected if there is at least one constraint between them. The ordering is determined by user-defined heuristics and classical CSP heuristics can be used as presented in Figure 1. Solution synthesis uses the partial ordering to perform an exhaustive search with backtracking. An agent instantiates its local problem w.r.t. higher priority agents and sends its local solution to lower priority neighbours, while backtracking messages are passed back up the ordering. This process computes a global solution by distributed aggregation of local solutions.

3 Delegation in DisCSP

Consider the situation shown in Figure 2. A_i has to share its partial solution with at least two connected children, A_j and A_k. On receiving this solution, both A_j and A_k make choices of their own, and transmit those to their chosen neighbours, invoking further search. Suppose A_k then receives A_j's choice, and discovers it is incompatible with its own choice. It must then find a new consistent choice and transmit that to its neighbours, overriding the previous message. This will invoke a new search, whose messages may take some time to catch up and override the previous one, and thus two searches, each requiring messages and computation, are spreading across the network at the same time, even though one of them is redundant. Alternatively, A_k may not be able to find a consistent value, and so must transmit a backtrack message to A_j; meanwhile, the previous redundant search continues in the rest of the network without being cancelled.

The question we ask here is whether a more selective procedure for transmitting partial solutions can improve efficiency by reducing the number and size of redundant searches. In particular, we consider whether an agent should reduce the number of forward messages it sends, by delegating some children to relay the messages to

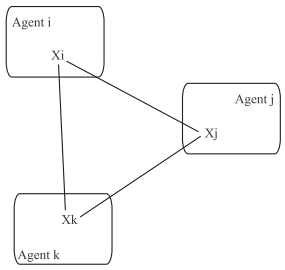


Fig. 2. A DisCSP agent 3-clique

other children. In our example, A_i might choose to delegate through A_j, and so A_k would receive messages from A_j only, where each message contains consistent value assignments for A_i and A_j. Initially, this reduces the number of messages (from 3 to 2 between the agents shown), and may stop a possibly redundant search initiated before A_j's value is transmitted. However, (i) the details of the search algorithm can interact in different ways to cause different searches, and (ii) the inherent unpredictability of message timings can have significant effects on the efficiency of an algorithm.

We define delegation in DisCSPs as follows:

Definition 1 *A DisCSP P=(X,D,C,A) can implement delegation iff,* $\exists A_i,\ A_j,\ A_k,\ X_i,\ X_j,\ X_k$ *s.t.* $f(X_i) = A_i, f(X_j) = A_j, f(X_k) = A_k$ *and* $\exists C_{ij}, C_{ik}, C_{jk} \in C$. A_i *can delegate its messages for* A_k *via* A_j*, denoted as* $(A_i, (A_j, A_k))$*, s.t.* A_i *does not send any messages directly to* A_k*, and* A_j *relays* A_i*'s decisions to* A_k *instead.*

Note that the definition ensures that the agents form a *3-clique*, and hence delegation does not add any extra links in the graph. Furthermore, delegation preserves the privacy of existing algorithms since although partial solutions are collated within agents, no agent will receive any values it would not have received without delegation.

3.1 Delegation vs. Synchronisation

The collation and transmission of partial solutions in delegation appears to be similar to synchronised search [2,9,11]. However, there are significant differences. In the following, we highlight some key features of synchronous search and we use these to show how it differs from delegation and why asynchrony is still retained with delegation.

1. **Privilege passing:** At the core of synchronous distributed search is the concept of *privilege passing* [2,11], where agents are in turn given a privilege to extend a partial solution or to revise earlier decisions as a search progresses. Generally, in synchronous search one agent is active at a time while all the other agents remain in a wait state, although in some versions [2] a DFS-tree ordering allows agents in unconnected branches of a search tree to be active simultaneously. Privilege passing ensures that agents have up-to-date information on the state of a search and as such minimises useless processing. In delegation there is no concept of privilege

passing. Agents still retain their autonomy and they carry out an asynchronous search where each agent is triggered into action whenever it receives messages irrespective of the state of its ancestors or successors. Therefore, an agent may be active simultaneously with other agents that are constrained with it. However, there is clearly some form of local synchronisation in delegation, since some agents only receive a parent's decision after some intermediary has processed it.

2. **Backtracking:** because of privilege passing, the processing of backtrack messages tends to be in reverse order of the search, and an agent must wait until all its children complete their actions before processing a backtrack from one of them. Using delegation, an agent responds immediately to a backtrack message, and can initiate new searches as a result, so there can be multiple searches proceeding simultaneously in the same sub-tree. The basic principle of delegation also makes no commitment to what should happen when a dead-end is discovered - depending on the details of the algorithm, an agent may decline to forward infeasible partial solutions, or may forward some sub-solution, allowing child agents to continue with a search or learn nogoods.
3. **Privacy:** In the synchronous algorithms of [9,11], the current partial solution for all ancestors is passed from one agent to the next, and thus agents will receive values for variables to which they are not connected. Using delegation, an agent should only receive values that it would also have received in the original algorithm. An additional consequence of this is that message packets should be smaller.

The idea of deputing agents was also explored in the Asynchronous Partial Overlay (APO) algorithm [5]. APO involves a resolution process that requires conflicting agents to centralise information about related parts of a problem within a mediator to resolve conflicts. There are significant privacy implications from mediation as agents have to reveal complete information about their domains and constraints violations for mediation to take place. In contrast, delegation requires agents simply to detect cliques, select intermediaries, and to route only the information that intermediaries are expected to see through them. Previous research on the performance trade-offs between synchronous and asynchronous backtracking have shown that message passing is reduced with synchronisation, but these savings come with the cost of an increase in run time (e.g. in [9]). However, later results reported in [11] suggest that synchronous algorithms may perform equally as well as asynchronous algorithms in runtime although idle time is much higher in synchronous search. Other results reported in [1] also show that the inclusion of some partial synchronisation improves efficiency of asynchronous backtracking - improving both the message count and the runtime.

4 Performing static delegation

We first consider delegation inside IDIBT/CBJ [4]. In the preprocessing phase of IDIBT /CBJ, agents are ordered with the Distributed Agent Ordering (DisAO) algorithm [4], part of which involves an extension of DisCSP graphs with the addition of tautological constraints between unconnected agents along different sub-trees. The extensions ensure the correctness of backtracking steps. The algorithm and its proof of complete-

ness have been described in [4]. In this section, we show how to plan delegation after ordering but before search, and evaluate its effect experimentally.

4.1 Establishing Delegation Paths

Algorithm 1 is presented for establishing the delegation paths below an agent A. This algorithm is independent of the tree search and it is run after agents construct an ordering with DisAO. Therefore each agent knows its parents, its children, and their positions in the ordering. The local data structures can be interpreted as follows: d[i] states whether A talks directly to c_i; l[i] is the length of the delegation path to c_i; m[i][j] indicates whether c_i is a parent of c_j; r[i][j] states whether c_i will relay messages to c_j; and f[i][j] states whether A must forward p_i's messages to c_j.

First, an agent must detect all ordered 3-cliques involving itself and two children. Each agent sends the full list of its children to each of its parents (line 2); the receiving agent can then populate its parenthood matrix m (3-7). The agent then processes each child in order of priority; if the child has no intermediate parents, it remains directly connected; otherwise, the intermediate parent that is furthest away from the agent (10) is selected to relay messages (11), the child's path length is updated (12), and it is marked as no longer directly connected (13). Once all delegations have been selected, each child is told to whom it must relay A's messages (15). Finally, when an agent receives those messages from its parents, it records the relay instructions (18). This algorithm selects the longest path for each delegation by chaining together overlapping 3-cliques. We aim for long delegation paths in order to remove many forward links, and so that the final messages aggregate as many local solutions as possible.

Figure 3(a) presents an example of this algorithm in use. We assume the 5 agents have been ordered using DisAO with the max-deg heuristic, as shown in Figure 3(b). First, X_5 will send an empty list to both X_4 and X_1, X_4 sends $\{X_5\}$ to X_3 and X_1, and X_3 sends $\{X_2, X_4\}$ to X_1. On receipt of these messages, X_1 keeps X_3 on a direct link, and then decides to delegate X_3 to relay messages to X_4. Similarly, X_1 eliminates the forward links with X_2, selecting X_3 as the intermediary, and eliminates the forward link to X_5, with X_4 as the intermediary. Figure 3(c) shows the DisCSP with the active forward links (solid links) after delegation paths have been established. Note that only 1 out of 4 links from X_1 is active. Therefore, during a tree search X_1 will only have to communicate with X_3 whenever it revises its value, but it knows the updates will reach all its children. Note that messages from X_1 to X_5 are relayed twice: X_3 will relay X_1's decision to X_4, and X_4 knows that if it receives a decision forX_1, it must extract it and relay it to X_5.

Algorithm 2 describes the process of relaying the appropriate decisions during search. A message to an agent from a parent will contain a decision for that parent, and may contain decisions for other parents as well. The agent first makes its own decision (line 1). The agent initialises a message for each of its children c_j with active links and places its value in the message if it has one (3). And then for each other parent decision (5), if there is another child c_t for which A uses c_j as an intermediary and to which A is expected to forward the other parent's decision (7), then A adds that decision to c_j's message (8). If however, the agent has no value, it holds on to the values for

Algorithm 1: choosing delegation paths for agent A

```
/* n children c_1, ... c_n, m parents p_1, ... p_m */
Data: d: array of n booleans, initially true
Data: l: array of n ints, initially 1
Data: m: array of n×n booleans, initially false
Data: r: array of n×n booleans, initially false
Data: f: array of m×n booleans, initially true
1  foreach parent p_i do
2      send message to p_i containing {c_1, ..., c_n}
3  foreach child c_i do
4      receive message from c_i containing child set C_i
5      for x ∈ C_i do
6          if x is a child of A (x = c_j) then
7              m[i][j] ← 1
8  foreach child c_j in order do
9      if ∃ k s.t. m[k][j] = 1 then
10         find i with highest l[i] s.t. m[i][j] = 1
11         r[i][j] ← true
12         l[j] ← l[i]+1
13         d[j] ← false
14 foreach child c_i do
15     send message to c_i with {c_j : r[i][j] = true}
16 foreach message with S_i received from a parent p_i do
17     foreach c_j ∈ S_i do
18         f[i][j] ← 1
```

each parent's decision that contributes to the domain wipe out (i.e. its conflict set) and it forwards other parents' decisions (10).

4.2 Theoretical properties

Here we proof that delegation paths created locally with local information are valid, showing that: (1) there is always a path of active forward links between each agent and each of its children, and (2) privacy is preserved in the paths generated. In particular, we will show that all intermediaries chosen to forward messages to child neighbours have active forward links with those children. We also give some other formal properties of Algorithm 1. For these results, we assume that the agent ordering algorithm, DisAO, produces a single highest priority agent, and recursively adds links to unconnected agents that share a child.

Theorem 1 *All agents in a delegation path from A_i to A_n are children of A_i*

Proof. From Algorithm 1, A_i only reasons and constructs paths with its children.

Algorithm 2: Agent A reacting to decision from p_i

Input: $M_i = \{(p_k, x_k) : p_k \text{ is parent of A}\}$

```
1  choose A's decision x_A
2  foreach c_j s.t. d[j] = true do
3      if x_A ≠ null then
4          message[j] ← {(A, x_A)}
5      foreach (k, x_k) ∈ M_i do
6          if x_A ≠ null then
7              if f[k][t] = true then
8                  message[j] ← message[j] ∪{(k, x_k)}
9          else
10             if (f[k][t] = true) ∧(k ∉ conflictSet_A) then
11                 message[j] ← message[j] ∪{(k, x_k)}
12     send message[j] to c_j
```

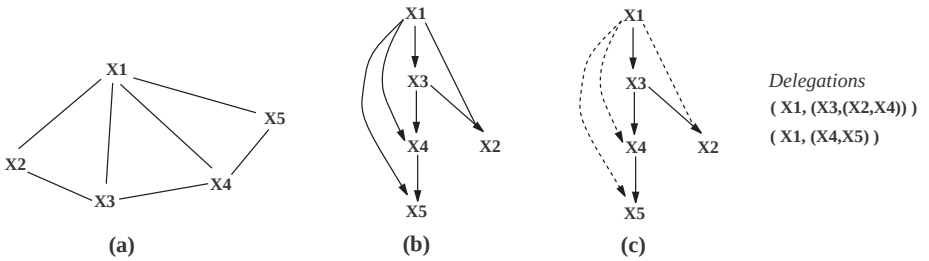


Fig. 3. An example DisCSP (a), a max-deg ordering established with DisAO (b), and active forward links (solid lines) after delegation (c).

Theorem 2 *The path length assigned to an agent A_j during the algorithm for agent A_i is the maximum path length from A_i to A_j using the sub-graph defined byA_i and A_i's children.*

Proof. (Omitted) by induction on the assigned path length

Corollary 3 *The delegation path from A_i to A_n is a maximum length path from A_i to A_n in the sub-graph defined by A_i and A_i's children.*

Proof. proof straight from Theorem 2.

Theorem 4 *For any 3-clique (A_i, A_j, A_n), where $A_i \rightarrow A_j, A_i \rightarrow A_n, A_j \rightarrow A_n$ and there is a delegation $(A_i, (A_j, A_n))$, A_j will send its own decisions directly to A_n.*

Proof. To prove this, we will show that there can be no chain of agents $C_1, C_2, \ldots, c_t$ between A_j and A_n in the ordering, such that A_j delegates its message for A_n through

this chain. The extension phase of DisAO is a recursive process that add directed links (tautological constraints) between pairs of unrelated agents if they have at least one child in common. If there was such a chain, the procedure would have added links $A_i \rightarrow C_k$ for each C_k, working backwards from the child A_n. But by Corollary 4.3, the chain C_1 to C_t must be the longest path between A_j and A_n. Substituting this chain for the direct link $A_j \rightarrow A_n$ would then give a longer path in A_i's delegation chain. But by Corollary 4.3, A_i has chosen the longest path. Contradiction. Thus there can be no such intermediate chain, and so A_j must send its decision directly to A_n.

Theorem 5 *An agent A_j can receive a decision $A_i \leftarrow v$ by delegation if and only if it can receive it without delegation, given the ordering induced by DisAO (and so we don't violate the privacy of any message).*

Proof. (Omitted) by inspection of the algorithm

Theorem 6 *No agent A_j receives the same decision $A_i \leftarrow v$ from two separate agents.*

Proof. (Omitted) by inspection of the algorithm

4.3 Algorithm modifications

To implement delegation in IDIBT/CBJ, following modifications were made to the algorithm:

- Agents construct delegation structures after an ordering has been established with DisAO by running the processes in Algorithm 1. Once the delegations have been selected, each child is told to whom it must relay its parents messages.
- A search proceeds as normal with IDIBT/CBJ, except that each agent will only send ***InfoVal*** messages to those child neighbours with whom it has an active forward link. And when an agent receives values from a parent neighbour, for whom it acts as an intermediary, it relays that value to the respective child neighbour after it has processed the message and selected its own value (see Algorithm 2).
- All backward links remain active. Even if there is an intermediary between agents A_i and A_k, A_k will bypass the intermediary and send ***Back*** messages directly to agent A_i.
- For the sake of simplicity, all search in IDIBT/CBJ is executed in a single search context (i.e. assuming NC=1).

In Section 3.1, we highlighted backtracking as one of key differences of delegation and synchronous search. In synchronous search, when the agent holding the current partial solution backtracks it returns the privilege (and by extension the partial solution) back to a culprit variable. In delegation, however, there are a number of options for dealing with partial solutions by dead end agents. In our preliminary evaluations, we considered the following:

1. Backtracking agents still pass down collated partial solutions to child neighbours. The case for doing this is that while a search below the dead end agents continues with an incorrect search context, it gives opportunities for child neighbours to quickly detect other conflicts with subsets of the partial solutions that remain coherent after the resolution of the original conflict.

2. An agent could hold back its conflict set i.e. the subset of parents' values that cause a domain wipe out, and relay other values. Again, this allows the search to retain its asynchrony as well as allow child neighbours to quickly detect conflicts with the other ancestors in the solutions they receive.
3. There is the option of allowing agents temporarily hold up the search beneath them and not relay any parent decisions to child neighbours whenever they can not extend the partial solutions. With this, child neighbours will eventually receive more coherent solutions but this approach comes with a risk of these pauses cascading up a search tree and gradually introducing additional synchronisation into agents activations.

While each of the choices has its merits, we chose the option of holding back conflict sets in the implementation of delegation in IDIBT/CBJ. This allows us to preserve more asynchrony in a search (compared to holding all collated values) while reducing the amount of redundant work would be performed if agents still forward down all collated values when they perform backtracking.

4.4 Evaluations

IDIBT/CBJ with delegation was implemented in a discrete event simulation environment using a shared simulated clock for all agents. In the simulation, we assume that the time taken to perform each constraint check is equivalent to one simulated time step. And, we also assume that message passing delay is uniform for all agents.

The modified algorithm was tested on random DisCSPs $\langle n = 30, d = 10\rangle$ on problems with different constraint densities ($p_1 = \{0.3, 0.5\}$) and different percentages of forbidden tuples (p_2) in the constraints for each density. 30 problems where generated for each combination of constraint density and tightness. For comparison, similar runs were made on the same problems with IDIBT/CBJ and a synchronous backjumping algorithm (SCBJ) [11]. SCBJ is a distributed equivalent of a centralised backjumping algorithm, where a total ordering [3] is imposed on agents and agents are activated in turn (one at a time) to extend a partial solution. A partial solution is passed from one agent to the next during a search. When the partial solution can not be extended, the earliest assignments in the partial solution that contribute to a domain wipe-out are resolved into a conflict set and used to determine (and activate) the culprit agent in backjumping.

In the charts plotted in Figure 4, we summarise the results from the experiments performed. The results show the average message count and the average Non-Concurrent Constraint Checks (NCCC) [6] from the runs. The average message count plotted in Figure 4 comprises the cost of running DisAO, messages exchanged in the course of the different searches, and where applicable, the messages exchanged in performing delegation with Algorithm 1. The results presented show with SCBJ, the average message count is lower than the asynchronous algorithms but its average NCCC is higher. With delegation in place, message count in the asynchronous search is reduced significantly especially on the most difficult problems.

[3] To keep comparisons fair, we use the same *max-deg* agent ordering from DisAO in SCBJ as well.

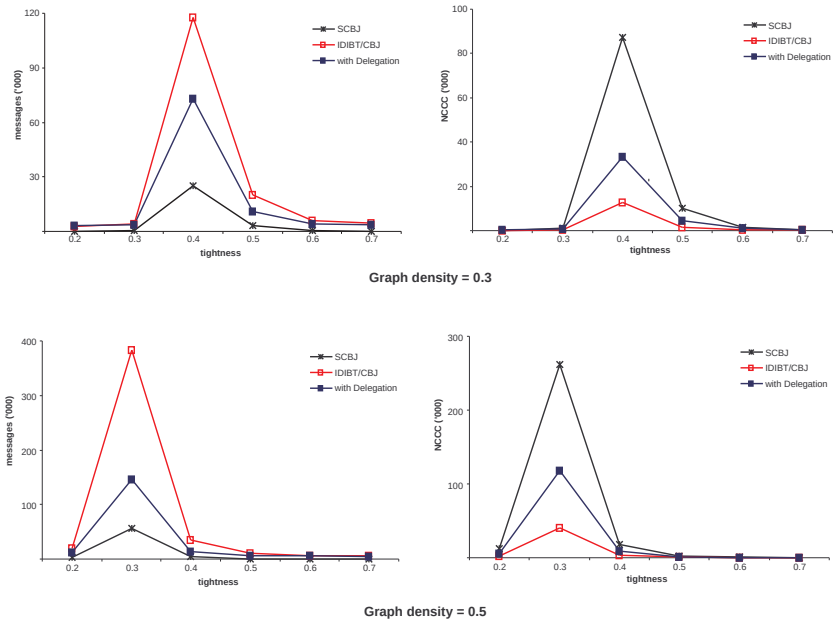


Fig. 4. Average message count and NCCC for solving DisCSPS with SCBJ, IDIBT/CBJ, and IDIBT/CBJ with delegation.

With respect to the NCCC, it appears that with delegation the average cost is higher than the corresponding cost for IDIBT/CBJ but lower than the averages for SCBJ. NCCC is a measure of the longest chain of sequential checks that can not be performed concurrently. Delegation appears to worsen performance on this metric because, firstly, it creates a physical chain of agents which can lengthen the sequential chain of constraint checks included in the final value of the metric. Secondly, delegation introduces some artificial delays in message passing, particularly for agents at the end of long message passing chains. It meant that in some cases the detection of conflicts with parents high up in a search tree was some times delayed. This resulted in intermediate agents having to abandon and reconstruct solutions, when such conflicts were discovered, with cost implications for the NCCC count.

Our motivation for delegation is to improve communication overhead of distributed backtracking in scenarios where the message passing is expensive relative to constraint checking but privacy is an issue. The results show that message passing is reduced with delegation compared to standard IDIBT/CBJ; however, this improvement comes with the cost of additional constraint checks. The results also show that by maintaining asynchrony in the search, constraint checking with delegation is lower than SCBJ. Message passing in SCBJ is lower but this is only achieved by violating privacy when partial solutions are passed from one agent to the next - and agents receive values they are not meant to see.

5 Dynamic delegation

Algorithms based on ABT [9,1] add links during search (as opposed to IDIBT/CBJ's preprocessing step). In such cases, precomputing the delegation paths before search is unlikely to be effective. Therefore, we now consider a dynamic delegation strategy, in which we allow agents to detect 3-cliques from the nogoods generated during a search, and then to chain together valid delegation paths. In this section, we describe our dynamic delegation method, implement it in a variant of ABT, and evaluate it experimentally.

5.1 Performing dynamic delegation

As mentioned earlier, in this form of delegation, coherent nogoods generated during a search are used to discover 3-cliques and to establish delegation paths. The key idea here is to allow an agent that receives nogoods to use the information to identify cliques and to determine if it can act as an intermediary for parent neighbours in the nogood.

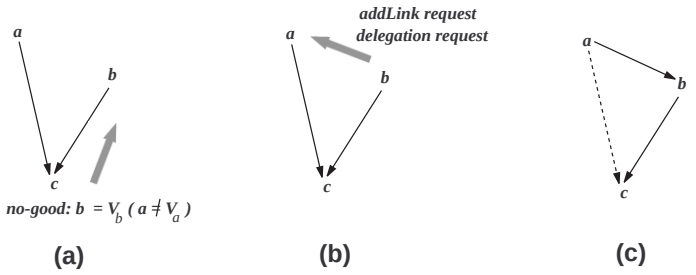


Fig. 5. Discovering cliques from nogoods.

The example in Figure 5 is used to illustrate the process. A_b receives the nogood ($b \neq V_b \Rightarrow (a = V_a)$) from which it can detect the 3-clique involving itself and both A_a and A_c. If the link from A_b to A_c is active, A_b can determine if it can perform delegation on behalf of the parent A_a to the child A_c. In this example, such an opportunity exists. Therefore, using Algorithm 3, A_b will send a delegation request to A_a for the delegation. After which the request is saved, to prevent A_b sending the same request repeatedly if there is thrashing along its path with both the parent and the child.

When A_a receives the delegation request, it processes the request with Algorithm 4[4]. A delegation request is rejected if either the forward link from the receiving agent to the target agent is inactive or the receiving agent is already delegating messages for a parent to the target agent. Otherwise, the request is accepted; which prompts the recipient to implement the delegation and to respond with an acceptance e.g. the acceptance from A_a to A_b, prompting agent A_b to relay A_a's decisions to A_c whenever ever it receives a value update from A_a; and agent A_a de-activates its forward link with A_c.

[4] Algorithm 4 uses the same data structures with Algorithm 1.

Algorithm 3: Agent A using nogoods to discover 3-cliques and delegations.

Input: $NG = \{(p_1, ...p_n) : p_i$ is var $\in rhs(NG)$; s is nogood sender$\}$
Data: Q: list of delegation requests sent by A

```
1 foreach p_i ∈ rhs(NG) do
2     if (p_i, (A, s)) ∉ Q then
3         send delRequest(p_i, A, s) to p_i;
4         Q ← Q ∪ (p_i, (A, s))
```

Algorithm 4: Agent A responding to delegation requests.

Input: $delRequest(A, (A_i, A_t)) : A_i$ is intermediate agent, A_t is delegation target

```
1 i ← index of A_i in children(A);
2 t ← index of A_t in children(A);
3 if (d[t] = false) ∨ (f[∗][t] = true) then
4     reject (A, (A_i, A_t));
5 /* accepting delegation */;
6 d[t] = false;
7 r[i][t] = true;
8 send message to A_i with {A_t : r[i][t] = true};
```

5.2 Algorithm modifications

For dynamic delegation, we modified ABT with partial synchronisation (ABTHyb) [1] as follows:

- Firstly, two new message types are introduced:
  - *delRequest*(A_i, A_j, A_k) - delegation request sent from an intermediate neighbour (A_j) to a parent (A_i) to inform the parent of the opportunity for delegating messages for A_k through A_j.
  - *delegation*(A_i, A_j, A_k) - an acceptance for a delegation request. This is sent from the parent A_i to the the intermediary A_j.
- Agents run the steps outlined in Algorithm 3 whenever nogoods are received from child neighbours. In the case that nogoods received contain values for unconnected agents, new links with these agents are first created before any delegation requests are sent.
- In response to delegation requests received, steps in Algorithm 4 are used to determine if such requests are accepted.
- Backward links between all agents remain active. Therefore nogoods are sent directly culprit parents irrespective of the delegation structures that exist at the time.
- When an agent sends a nogood, it moves into the synchronous phase of the search as described in [1]. With delegation, the agent will continue this partial synchronisation by holding on to any collated partial solutions until it returns to the asynchronous search.

5.3 Evaluation

The same problems from Section 4.4 were used to evaluate ABTHyb with dynamic evaluation. As previously, we also compared the modified algorithm with its ancestor and with SCBJ. However, for these evaluations we make SCBJ a nogood recording algorithm (and therefore it is referred to as SBT in the results for consistency), so that all three algorithms are compared on like terms. Furthermore, a total (*max-deg*) ordering is imposed on agents in all three algorithms.

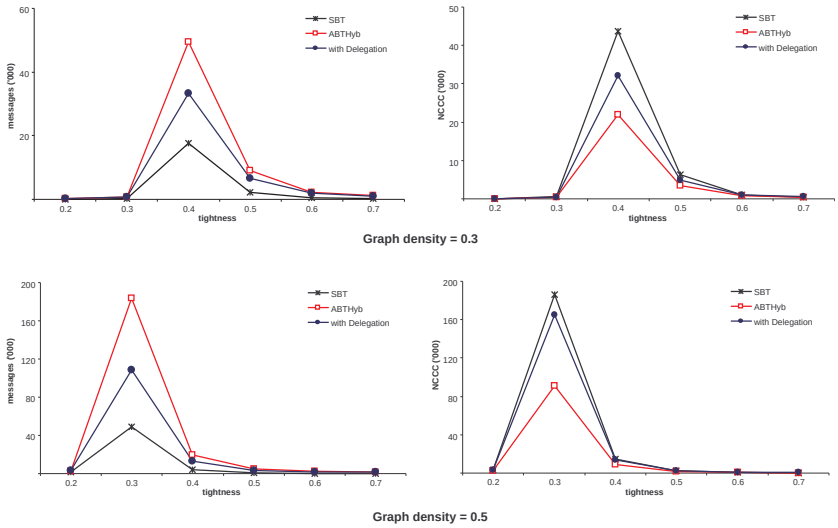


Fig. 6. Average message count and NCCC for solving DisCSPS with SBT, ABTHyb, and ABTHyb with delegation.

In Figure 6, we plot the averages of the evaluation metrics from the runs of the three algorithms. The results are consistent with the previous findings. They show that the synchronised algorithm still saves on message passing (although at the cost violating privacy) and that the NCCCs remain higher. Between the asynchronous algorithms, the effects of delegation is consistent with the earlier findings. Message passing cost is considerably reduced with delegation in place and there is the accompanying increase in the average NCCC count. Some reasons for the higher average NCCC count have been listed in Section 4.4, and these are still applicable.

6 Conclusion

We have introduced a new concept of delegation for improving the efficiency of message passing in distributed tree search. With delegation, agents can appoint intermediate neighbours for transmission of their local solutions to other child neighbours. The idea

reduces message passing by de-activating some forward links from agents with delegations, but it can enhance the search by improving the coherency of partial solutions received by agents while still preserving the privacy levels of an underlying algorithm.

We presented two forms of delegation for the prominent tree search strategies in DisCSP i.e. where links between unconnected agents are created prior to a search and where such links are created on the fly as required. We have shown, with empirical experiments, that both forms of delegation reduce message passing in asynchronous algorithms by as much as 50%. But the improvements come at the cost of additional non-concurrent checks, which we attribute to some implicit delays caused by the delegation of messages.

Overall, the results indicate that delegation is effective strategy for distributed tree search where the cost of message passing is significant relative to the cost of constraint checking and where privacy is an issue. For future work, we intend to extend delegation to other algorithms, such as ADOPT [7], and to improve the dynamic delegation algorithm. In particular, we are motivated by problems where the network connections are expensive, of different quality, and unreliable, and we are developing delegation heuristics which make use of this information.

References

1. I. Brito and P. Meseguer. Synchronous, asynchronous and hybrid algorithms for DisCSPs. In *Proc. of the 5th Int'l Workshop on Distributed Constraint Reasoning*, September 2004.
2. Z. Collin, R. Dechter, and S. Katz. On the feasibility of distributed constraint satisfaction. In *Proc. of the 12th Int'l Joint Conference on Artificial Intelligence, IJCAI*, pages 318–324, 1991.
3. Y. Hamadi, C. Bessière, and J. Quinqueton. Backtracking in distributed constraint networks. In *ECAI*, pages 219–223, Aug 1998.
4. Youssef Hamadi. Conflicting agents in distributed search. *Int'l Journal on Artificial Intelligence Tools*, 14(3):459–476, 2005.
5. R. Mailler and V. Lesser. Using cooperative mediation to solve distributed constraint satisfaction problems. In *Proc. of 3rd Int'l Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2004)*, 2004.
6. A Meisels, I. Razgon, E. Kaplansky, and R. Zivan. Comparing performance of distributed constraints processing algorithms., July 2002.
7. P. J. Modi, W. Shen, M. Tambe, and M. Yokoo. An asynchronous complete method for distributed constraint optimization. In *In Proc. The 2nd Int'l Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2003*, pages 161–168. ACM, July 2003.
8. M. Yokoo. *Distributed Constraint Satisfaction:Foundation of Cooperation in Multi-agent Systems*. Springer, 2001.
9. M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *12th Int'l Conference on Distributed Computing Systems (ICDCS-92)*, pages 614–621, 1992.
10. M. Yokoo, T. Ishida, and K. Kubawara. Distributed constraint satisfaction for DAI problems. In M. N. Huhns, editor, *Proc. of the 10th International Workshop on Distributed Artificial Intelligence*, chapter 9. 1990.
11. R. Zivan and A. Meisels. Synchronous vs asynchronous search on DisCSPs. In *Proc. First European Workshop on Multi-Agent Systems (EUMAS)*, December 2003.

Termination Problem of the APO Algorithm

Tal Grinshpoun, Moshe Zazon, Maxim Binshtok, and Amnon Meisels

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

Abstract. Asynchronous Partial Overlay (APO) is a search algorithm that uses cooperative mediation to solve Distributed Constraint Satisfaction Problems (DisCSPs). The algorithm partitions the search into different subproblems of the DisCSP. The proof of completeness of the APO algorithm is based on the growth of the size of the subproblems. The present paper presents an example DisCSP and a detailed run of APO on the example. In the resulting scenario, the run of the algorithm enters an infinite loop. The presented example and scenario contradict the termination and consequently the completeness of the APO algorithm. A correction to the problem that prevents the infinite loop in our example is proposed. A reference to the problematic part in the proof of APO's completeness is also given.

1 Introduction

Asynchronous Partial Overlay (APO) [4, 3] is an algorithm for solving Distributed Constraint Satisfaction Problems (DisCSPs) that uses the concept of mediation to centralize the search procedure in parts of the DisCSP. The APO algorithm belongs to a family of DisCSP search algorithms that is radically different than distributed backtracking algorithms. Distributed backtracking, which forms the majority of DisCSP algorithms, can take many forms. Asynchronous Backtracking (ABT) [7], Asynchronous Forward-Checking (AFC) [5], and Concurrent Dynamic Backtracking (ConcDB) [8] are representative examples of the family of distributed backtracking algorithms. All of these algorithms maintain one or more partial solutions of the DisCSP and attempt to extend the partial solution into a complete one. The extension of partial solutions can be performed asynchronously (as in ABT [7]) or concurrently over multiple solutions (as in ConcDB [8]).

The uniqueness of APO lies in its basic operation of merging partial solutions into a complete one. Merging of different partial solutions is distinct from extending partial solutions. To our best knowledge, Descending Requirement Search (DesRS) [6], is the only search algorithm, beside APO, that merges partial solutions. However, the DesRS algorithm is based on a hierarchical partition of the DisCSP. The imposed hierarchical structure guaranties the correctness of DesRS.

The APO algorithm partitions the agents into groups that attempt to find consistent partial solutions. The partition mechanism is dynamic during search

and enables a dynamic change of groups. The key factor in the termination (and consequently the completeness) of the APO algorithm as presented in the correctness proof in [4] is the growth of initially partitioned group during search. This growth is possible, since the subproblems overlap, allowing agents to increase the size of the subproblems they solve. The proof argues that a subset of agents could cycle infinitely through their allowable values without reaching a solution, if the size of that subset does not increase over time.

The proof of APO's completeness in [4] relies on the assumption that a set of agents $V' \subseteq V$ cannot enter such a state of oscillation. This assumption is considered to be true, since after a mediation session, at least one conflict must be created or must remain. Otherwise, the oscillation would stop and the problem would be solved [4].

The present paper constructs a scenario in which concurrent mediation sessions do not have any remaining conflicts, nor are they aware of any new conflicts created. The solution of these sessions creates new conflicts that become known only after the mediation sessions are over. Thus, the size of the subproblems does not increase. Moreover, the demonstrated scenario leads to an infinite loop of the APO algorithm's run. Such a scenario contradicts the termination and consequently the completeness of APO.

A simple DisCSP with 3 or 4 agents is not enough to achieve such an infinite loop scenario, since we must have at least two concurrent mediation sessions. Consequently, we use a DisCSP with 8 agents and a symmetrical constraint graph. Following the asynchronous nature of the algorithm, some of the messages in our scenario are delayed. All delays are finite.

In the rest of this paper, we briefly describe the APO algorithm accompanied by its pseudo-code as presented in [4] (section 2). We then present our infinite loop scenario in detail (section 3). A proposed correction to the problem is described in section 4. Section 5 presents a discussion.

2 Asynchronous Partial Overlay

Asynchronous Partial Overlay (APO) is an algorithm for solving DisCSPs that applies cooperative mediation. The pseudo-code in figures 1, 2, and 3 follows closely the presentation of APO in [4].

Using mediation, agents can solve subproblems of the DisCSP by conducting an internal Branch and Bound search. For a complete solution of the DisCSP, the solutions of the subproblems must be compatible. When solutions of overlapping subproblems have conflicts, the solving agents increase the size of the subproblems that they work on. The original paper uses the term *preferences* to describe potential conflicts between solutions of overlapping subproblems. The present paper uses the term *external constraints* to describe such conflicts. A detailed description of the APO algorithm can be found in [4], the source of the complete version of APO that is used in the present paper.

procedure initialize
  $d_i \leftarrow random\ d \in D_i$;
  $p_i \leftarrow sizeof(neighbors) + 1$;
  $m_i \leftarrow$ **true**;
  $mediate \leftarrow$ **false**;
  add x_i to the *good_list*;
  send (**init**, $(x_i, p_i, d_i, m_i, D_i, C_i)$) to neighbors;
  $initList \leftarrow$ neighbors;
end initialize;

when received (**init**, $(x_j, p_j, d_j, m_j, D_j, C_j)$) **do**
  add $(x_j, p_j, d_j, m_j, D_j, C_j)$ to *agent_view*;
  if x_j is a neighbor of some $x_k \in$ *good_list* **do**
    add x_j to the *good_list*;
    add all $x_l \in agent\_view \wedge x_l \notin good\_list$
      that can now be connected to the *good_list*;
    $p_i \leftarrow sizeof(good\_list)$;
  end if;
  if $x_j \notin initList$ **do**
    send (**init**, $(x_i, p_i, d_i, m_i, D_i, C_i)$) to x_j;
  else
    remove x_j from *initList*;
  end if;
  check_agent_view;
end do;

when received (**ok?**, (x_j, p_j, d_j, m_j)) **do**
  update *agent_view* with (x_j, p_j, d_j, m_j);
  check_agent_view;
end do;

procedure check_agent_view
  if $initList \neq \emptyset$ **or** $mediate \neq$ **false do**
    return;
  $m'_i \leftarrow hasConflict(x_i)$;
  if m'_i **and** $\neg\exists j(p_j > p_i \wedge m_j ==$ **true**$)$ **do**
    if $\exists(d'_i \in D_i)(d'_i \cup agent\_view$ does not conflict)
      and d_i conflicts exclusively with lower priority neighbors **do**
      $d_i \leftarrow d'_i$;
      send (**ok?**, (x_i, p_i, d_i, m_i)) to all $x_j \in agent\_view$;
    else
      do mediate;
    end if;
  else if $m_i \neq m'_i$ **do**
    $m_i \leftarrow m'_i$;
    send (**ok?**, (x_i, p_i, d_i, m_i)) to all $x_j \in agent\_view$;
  end if;
end check_agent_view;

Fig. 1. APO procedures for initialization and local resolution.

```
procedure mediate
  preferences ← ∅;
  counter ← 0;
  for each x_j ∈ good_list do
    send (evaluate?, (x_i, p_i)) to x_j;
    counter++;
  end do;
  mediate ← true;
end mediate;

when received (wait!, (x_j, p_j)) do
  update agent_view with (x_j, p_j);
  counter--;
  if counter == 0 do choose_solution;
end do;

when received (evaluate!, (x_j, p_j, labeled D_j)) do
  record (x_j, labeled D_j) in preferences;
  update agent_view with (x_j, p_j);
  counter--;
  if counter == 0 do choose_solution;
end do;

procedure choose_solution
  select a solution s using a Branch and Bound search that:
      1. satisfies the constraints between agents in the good_list
      2. minimizes the violations for agents outside of the session
  if ¬∃s that satisfies the constraints do
    broadcast no solution;
  for each x_j ∈ agent_view do
    if x_j ∈ preferences do
      if d'_j ∈ s violates an x_k and x_k ∉ agent_view do
        send (init, (x_i, p_i, d_i, m_i, D_i, C_i)) to x_k;
        add x_k to initList;
      end if;
      send (accept!, (d'_j, x_i, p_i, d_i, m_i)) to x_j;
      update agent_view for x_j;
    else
      send (ok?, (x_i, p_i, d_i, m_i)) to x_j;
    end if;
  end do;
  mediate ← false;
  check_agent_view;
end choose_solution;
```

Fig. 2. Procedures for mediating an APO session and for choosing a solution during an APO mediation.

```
when received (evaluate?, (x_j, p_j)) do
  m_j ← true;
  if mediate == true or ∃k(p_k > p_j ∧ m_k == true) do
    send (wait!, (x_i, p_i));
  else
    mediate ← true;
    label each d ∈ D_i with the names of the agents
      that would be violated by setting d_i ← d;
    send (evaluate!, (x_i, p_i, labeled D_i));
  end if;
end do;

when received (accept!, (d, x_j, p_j, d_j, m_j)) do
  d_i ← d;
  mediate ← false;
  send (ok?, (x_i, p_i, d_i, m_i)) to all x_j ∈ agent_view;
  update agent_view with (x_j, p_j, d_j, m_j);
  check_agent_view;
end do;
```

Fig. 3. Procedures for receiving an APO session.

3 An infinite loop scenario

Consider the 3-coloring problem presented in figure 4 by the solid lines. Each agent can assign one of the three available colors Red, Green, or Blue. To the standard inequality constraints that the solid lines represent, we add four weaker constraints (diagonal dashed lines) that do not allow only the combinations (Green,Green) and (Blue,Blue) to be assigned by the agents.

The initial selection of values by all agents is depicted in figure 4. In the initial state, two constraints are violated – (A1,A2) and (A5,A6). Assume that agents A3, A4, A7, and A8 are the first to complete their initialization phase by exchanging init messages with all their neighbors (procedure **initialize** in figure 1). These agents do not have conflicts, therefore they set $m_i \leftarrow$*false* and send *ok?* messages to their neighbors when each of them runs the **check_agent_view** procedure (see figure 1). After the arrival of the *ok?* messages, agents A1, A2, A5, and A6 accept *init* messages from all of their neighbors and complete the initialization phase. Agents A2 and A6 have conflicts, however they complete the **check_agent_view** procedure without mediating or changing their state. This is true, because in the agent views of A2 and A6, $m_1 = true$ and $m_5 = true$, respectively. These neighbors have higher priority over agents A2 and A6. We denote by *configuration 1* the states of all the agents at this point of the processing and present the configuration in Table 1.

After all agents complete their initializations, agents A1 and A5 detect that they have conflicts, and that they have no neighbor with a higher priority that

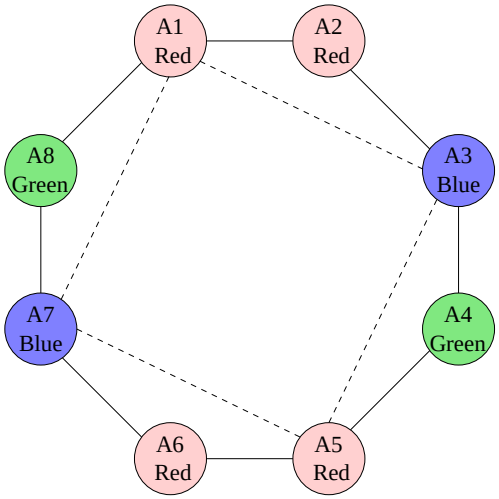


Fig. 4. The constraints graph with the initial setting.

Agent	Color	m_i	d_j values	m_j values
A1	R	$m_1 = t$	d_2 =R, d_3 =B, d_7 =B, d_8 =G	$m_2 = t$, $m_3 = f$, $m_7 = f$, $m_8 = f$
A2	R	$m_2 = t$	d_1 =R, d_3 =B	$m_1 = t$, $m_3 = f$
A3	B	$m_3 = f$	d_1 =R, d_2 =R, d_4 =G, d_5 =R	$m_1 = t$, $m_2 = t$, $m_4 = f$, $m_5 = t$
A4	G	$m_4 = f$	d_3 =B, d_5 =R	$m_3 = f$, $m_5 = t$
A5	R	$m_5 = t$	d_3 =B, d_4 =G, d_6 =R, d_7 =B	$m_3 = f$, $m_4 = f$, $m_6 = t$, $m_7 = f$
A6	R	$m_6 = t$	d_5 =R, d_7 =B	$m_5 = t$, $m_7 = f$
A7	B	$m_7 = f$	d_1 =R, d_5 =R, d_6 =R, d_8 =G	$m_1 = t$, $m_5 = t$, $m_6 = t$, $m_8 = f$
A8	G	$m_8 = f$	d_1 =R, d_7 =B	$m_1 = t$, $m_7 = f$

Table 1. Configuration 1.

wants to mediate. Consequently, agents A1 and A5 start mediation sessions, since they cannot change their own color to a consistent state with their neighbors.

We will first observe A1's mediation session. A1 sends *evaluate?* messages to its neighbors A2, A3, A7, and A8 (procedure **mediate** in figure 2). All these agents reply with *evaluate!* messages (see code in figure 3). A1 conducts a Branch and Bound search to find a solution that satisfies all the constraints between A1, A2, A3, A7, and A8, and also minimizes external constraints (procedure **choose_solution** in figure 2). In our example, A1 finds the solution (A1←Green, A2←Blue, A3←Red, A7←Blue, A8←Red), which satisfies the internal constraints, and minimizes to zero the external constraints. A1 sends *accept!* messages to its neighbors, informing them of its solution. A2, A3, A7, and A8 receive the *accept!* messages and send *ok?* messages with their new states to their neighbors (see code in figure 3). However, the *ok?* messages from A8 to A7 and from A3 to A4 and to A5 are delayed.

Agent	Color	m_i	d_j values	m_j values
A1	G	$m_1 = f$	d_2 =B, d_3 =R, d_7 =B, d_8 =R	$m_2 = f$, $m_3 = f$, $m_7 = f$, $m_8 = f$
A2	B	$m_2 = f$	d_1 =G, d_3 =R	$m_1 = f$, $m_3 = f$
A3	R	$m_3 = f$	d_1 =G, d_2 =B, d_4 =G, d_5 =R	$m_1 = f$, $m_2 = f$, $m_4 = f$, $m_5 = t$
A4	G	$m_4 = f$	$\boldsymbol{d_3}$ =**B**, d_5 =R	$m_3 = f$, $m_5 = t$
A5	R	$m_5 = t$	$\boldsymbol{d_3}$ =**B**, d_4 =G, d_6 =R, d_7 =B	$m_3 = f$, $m_4 = f$, $m_6 = t$, $m_7 = f$
A6	R	$m_6 = t$	d_5 =R, d_7 =B	$m_5 = t$, $m_7 = f$
A7	B	$m_7 = f$	d_1 =G, d_5 =R, d_6 =R, $\boldsymbol{d_8}$ =**G**	$m_1 = f$, $m_5 = t$, $m_6 = t$, $m_8 = f$
A8	R	$m_8 = f$	d_1 =G, d_7 =B	$m_1 = f$, $m_7 = f$

Table 2. Configuration 2 – obsolete data in *agent_views* is in bold face.

Concurrently with the above mediation session of A1, agent A5 starts its own mediation session. A5 sends *evaluate?* messages to its neighbors A3, A4, A6, and A7. Let us assume that the message to A7 is delayed. A4 and A6 receive the *evaluate?* messages and reply with *evaluate!*, since they do not know any agents of higher priority than A5 that want to mediate. A3, is in A1's mediation session, so it replies with *wait!*. We denote by *configuration 2* the states of all the agents at this point of the processing (see Table 2).

Only then, after A1's mediation session is over, A7 receives the delayed *evaluate?* message from A5. Since A7 is no longer in a mediation session, nor does it expect a mediation session from a node of higher priority than A5 (see A7's view in Table 2), agent A7 replies with *evaluate!*. Notice that A7's view of d_8 is obsolete (the *ok?* message from A8 to A7 is still delayed). When agent A5 receives the *evaluate!* message from A7, it can continue the mediation session involving agents A4, A5, A6, and A7. Since the *ok?* messages from A3 to A4 and A5 are also delayed, agent A5 starts its mediation session with knowledge about agents A3 and A8 that is not updated (see bold-faced data in Table 2).

Agent A5 conducts a Branch and Bound search to find a solution that satisfies all the constraints between A4, A5, A6, and A7, that also minimizes external constraints. In our example, A5 finds the solution (A4←Red, A5←Green, A6←Blue, A7←Red), which satisfies the internal constraints, and minimizes to zero the external constraints (remember that A5 has wrong data about the assignments of A3 and A8). A5 sends *accept!* messages to A4, A6, and A7, informing them of its solution. The agents receive these messages and send *ok?* messages with their new states to their neighbors. By now, all the delayed messages get to their destinations, and two constraints are violated – (A3,A4) and (A7,A8). Consequently, agents A3, A4, A7, and A8 want to mediate, whereas agents A1, A2, A5, and A6 do not wish to mediate, since they do not have any conflicts. We denote by *configuration 3* the states of all the agents after A5's solution has been assigned and all delayed messages arrived at their destinations (see figure 5 and Table 3).

Up until now, we have shown a series of steps that led from *configuration 1* to *configuration 3*. Next, we will show a very similar series of steps that will lead us right back to *configuration 1*.

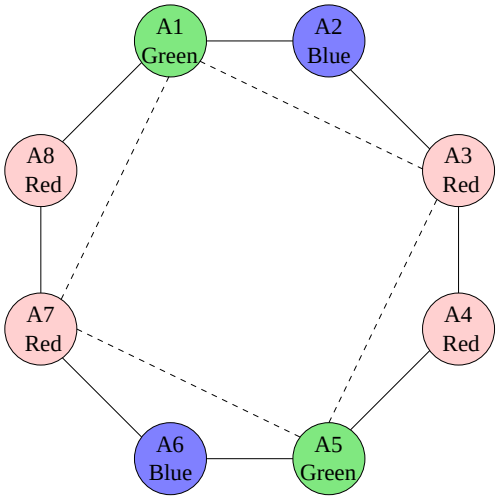


Fig. 5. The graph in configuration 3.

Agent	Color	m_i	d_j values	m_j values
A1	G	$m_1 = f$	d_2 =B, d_3 =R, d_7 =R, d_8 =R	$m_2 = f$, $m_3 = t$, $m_7 = t$, $m_8 = t$
A2	B	$m_2 = f$	d_1 =G, d_3 =R	$m_1 = f$, $m_3 = t$
A3	R	$m_3 = t$	d_1 =G, d_2 =B, d_4 =R, d_5 =G	$m_1 = f$, $m_2 = f$, $m_4 = t$, $m_5 = f$
A4	R	$m_4 = t$	d_3 =R, d_5 =G	$m_3 = t$, $m_5 = f$
A5	G	$m_5 = f$	d_3 =R, d_4 =R, d_6 =B, d_7 =R	$m_3 = t$, $m_4 = t$, $m_6 = f$, $m_7 = t$
A6	B	$m_6 = f$	d_5 =G, d_7 =R	$m_5 = f$, $m_7 = t$
A7	R	$m_7 = t$	d_1 =G, d_5 =G, d_6 =B, d_8 =R	$m_1 = f$, $m_5 = f$, $m_6 = f$, $m_8 = t$
A8	R	$m_8 = t$	d_1 =G, d_7 =R	$m_1 = f$, $m_7 = t$

Table 3. Configuration 3.

Agents A3 and A7 detect that they have conflicts and that they have no neighbor with a higher priority that wants to mediate. Consequently, agents A3 and A7 start mediation sessions, since they cannot change their own color to a consistent state with their neighbors.

We will first observe A3's mediation session. A3 sends *evaluate?* messages to its neighbors A1, A2, A4, and A5. All these agents reply with *evaluate!* messages. A3 conducts a Branch and Bound search to find a solution that satisfies all the constraints between A1, A2, A3, A4, and A5, and also minimizes external constraints. Agent A3 finds the solution (A1←Green, A2←Red, A3←Blue, A4←Green, A5←Red), which satisfies the internal constraints, and minimizes to zero the external constraints. A3 sends *accept!* messages to its neighbors, informing them of its solution. A1, A2, A4, and A5 receive the *accept!* messages and send *ok?* messages with their new states to their neighbors. However, the *ok?* messages from A2 to A1 and from A5 to A6 and to A7 are delayed.

Agent	Color	m_i	d_j values	m_j values
A1	G	$m_1 = f$	$\boldsymbol{d_2}$ =**B**, d_3 =B, d_7 =R, d_8 =R	$m_2 = f$, $m_3 = f$, $m_7 = t$, $m_8 = t$
A2	R	$m_2 = f$	d_1 =G, d_3 =B	$m_1 = f$, $m_3 = f$
A3	B	$m_3 = f$	d_1 =G, d_2 =R, d_4 =G, d_5 =R	$m_1 = f$, $m_2 = f$, $m_4 = f$, $m_5 = f$
A4	G	$m_4 = f$	d_3 =B, d_5 =R	$m_3 = f$, $m_5 = f$
A5	R	$m_5 = f$	d_3 =B, d_4 =G, d_6 =B, d_7 =R	$m_3 = f$, $m_4 = f$, $m_6 = f$, $m_7 = t$
A6	B	$m_6 = f$	$\boldsymbol{d_5}$ =**G**, d_7 =R	$m_5 = f$, $m_7 = t$
A7	B	$m_7 = t$	d_1 =G, $\boldsymbol{d_5}$ =**G**, d_6 =B, d_8 =R	$m_1 = f$, $m_5 = f$, $m_6 = f$, $m_8 = t$
A8	R	$m_8 = t$	d_1 =G, d_7 =R	$m_1 = f$, $m_7 = t$

Table 4. Configuration 4 – obsolete data in *agent_views* is in bold face.

Concurrently with the above mediation session of A3, agent A7 starts its own mediation session. A7 sends *evaluate?* messages to its neighbors A1, A5, A6, and A8. Let us assume that the message to A1 is delayed. A6 and A8 receive the *evaluate?* messages and reply with *evaluate!*, since they do not know any agents of higher priority than A7 that want to mediate. A5, is in A3's mediation session, so it replies with *wait!*. We denote by *configuration 4* the states of all the agents at this point of the processing (see Table 4).

Only after A3's mediation session is over, A1 receives the delayed *evaluate?* message from A7. Since A1 is no longer in a mediation session, nor does it expect a mediation session from a node of higher priority than A7 (see A1's view in Table 4), agent A1 replies with *evaluate!*. Notice that A1's view of d_2 is obsolete (the *ok?* message from A2 to A1 is still delayed). When agent A7 receives the *evaluate!* message from A1, it can continue the mediation session involving agents A1, A6, A7, and A8. Since the *ok?* messages from A5 to A6 and A7 are also delayed, agent A7 starts its mediation session with knowledge about agents A2 and A5 that is not updated (see bold-faced data in Table 4).

Agent A7 conducts a Branch and Bound search to find a solution that satisfies all the constraints between A1, A6, A7, and A8, that also minimizes external constraints. In our example, A7 finds the solution (A1←Red, A6←Red, A7←Blue, A8←Green), which satisfies the internal constraints, and minimizes to zero the external constraints (remember that A7 has wrong data about A2 and A5). A7 sends *accept!* messages to A1, A6, and A8, informing them of its solution. The agents receive these messages and send *ok?* messages with their new states to their neighbors. By now, all the delayed messages get to their destination, and two constraints are violated – (A1,A2) and (A5,A6). Consequently, agents A1, A2, A5, and A6 want to mediate, whereas agents A3, A4, A7, and A8 do not wish to mediate, since they do not have any conflicts. Notice that all the agents have returned to the exact states they were in *configuration 1* (see figure 4 and Table 1).

The cycle that we have just shown between *configuration 1* and *configuration 3* can continue indefinitely. This example contradicts the termination and completeness of the APO algorithm.

It should be noted that we did not mention all the messages passed in the running of our example. We mentioned only those messages that were important for the understanding of the examples, since the example was complicated enough. For instance, after agent A5 completes its mediation session (before *configuration 2*), there is some straightforward exchange of messages between agents, before the m_j values of all the agents are correct (as presented in Table 2).

4 Proposed correction to APO

The scenario presented in the previous section becomes possible, due to delayed updating of *agent_views* after completion of a mediation session. Agent A7 in *configuration 2* and agent A1 in *configuration 4* are the key players in our scenario, but we will focus on agent A7. Since agent A7 (in *configuration 2*) receives A1's **accept!** message before the arrival of A5's **evaluate?** message, agent A7 is ready to engage in a mediation session with agent A5. However, A7's *agent_view* is not yet updated, since the **ok?** message from agent A8 was delayed. This situation allows A5 to conduct a local search with wrong knowledge of external constraints, leading to a new conflict between agents A7 and A8. Since agent A5 finds a solution with no conflicts, it does not add any new agents to its neighborhood (procedure **choose_solution** in figure 2). This enables the infinite reoccurrence of our scenario.

A solution to this problem could be obtained if agent A7 would agree to participate in a new mediation session only when its *agent_view* is updated with all the changes of the previous mediation session. This can be achieved by the mediator sending its entire solution s in the **accept!** messages, instead of just particular d'_j's. The sending of **accept!** in procedure **choose_solution** can be changed to the following:

$$\text{send } (\textbf{accept!}, (s, x_i, p_i, d_i, m_i)) \text{ to } x_j;$$

Upon receiving the new **accept!** message, agent i now updates all the d_k's in the received solution s (accept for d_k's that are not in i's *agent_view*). Notice that agent i still has to send **ok?** messages to its neighbors, since not all of its neighbors were necessarily involved in the mediation session. The new code for receiving an **accept!** message is in figure 6.

Looking back at *configuration 2* from the previous section, we can observe that after the proposed correction, agent A7 will join A5's mediation session only with updated knowledge of A8's color. In such a case, A5 finds in its search a different solution, for example (A4←Green, A5←Red, A6←Green, A7←Blue) that has no external conflicts. The original solution of this step in our scenario (A4←Red, A5←Green, A6←Blue, A7←Red) now imposes a conflict between agents A7 and A8, since A7 is aware of A8's real color. The infinite cycle cannot occur in this scenario, because it relies on the possibility that after A5's mediation session, agents A7 and A8 will share the same color. This possibility is prevented by our proposed correction.

```
when received (accept!, (s, x_j, p_j, d_j, m_j)) do
  for each x_k ∈ s do
    if x_k ∈ agent_view do
      update agent_view with (x_k, d_k);
    end if;
  end do;
  mediate ← false;
  send (ok?, (x_i, p_i, d_i, m_i)) to all x_j ∈ agent_view;
  update agent_view with (x_j, p_j, d_j, m_j);
  check_agent_view;
end do;
```

Fig. 6. The proposed procedure for receiving an **accept!** message.

The above correction involves no additional exchange of messages. It only adds some data (the complete solution s instead of just d'_j) to the **accept!** messages of the original algorithm. This addition has negligible effect on the overall communication scope of the algorithm.

5 Discussion

The APO search algorithm is designed as an asynchronous distributed algorithm on DisCSPs. The proof of APO's completeness as presented in [4] relies on the incorrect assumption that the mediator always adds an agent to its *good_list* in case an external constraint is violated during a mediation session. This paper has presented a detailed example that uses delays in the delivery of messages. In the example, we show how a mediator can find a solution to its subproblem with no external conflicts created according to its *agent_view*, due to an obsolete view of external constraints in the time of mediation. This solution does however lead to violation of external constraints contrary to the assumption in [4].

In order to focus on the correctness problem discovered, we presented a scenario in which the APO algorithm enters an infinite loop. To solve the specific problem discovered we proposed a correction to the algorithm that prevents this scenario from happening. Our proposed correction adds necessary data synchronization at the end of a mediation session. Although our proposed correction is shown to be essential, it does not help in proving the correctness of the assumption that was mentioned before. Consequently, the termination and completeness of the APO algorithm remain unclear.

Benisch and Sadeh [1, 2] have proposed several new versions of the APO algorithm. In [1] they suggest an alternative way for selecting the mediators by changing the metric used in calculating the priority p of an agent. It is important to mention this proposed flexibility to the priority, since the proof of soundness of the APO algorithm in [4] relies on the fact that priorities only increase over time. This is not valid when changing the metric used in calculating the priority

as proposed in [1]. Consequently, the proof of soundness of the APO algorithm must be re-considered for different policies of priorities.

In addition to the mediation procedure originally proposed with APO (procedure **choose_solution** in figure 2), Benisch and Sadeh [2] propose two new mediation procedures – APO-BT and APO-ABT. Both of these versions of the APO algorithm do not include conflict minimization with agents outside of the mediation session. The scenario presented in section 3 can be easily simplified to generate an infinite loop, when conflicts with external agents are not minimized. According to the original publication of the APO algorithm, these versions of the algorithm do not affect the proofs of soundness and completeness. In fact, the proofs in [4] do not refer to the external conflict minimization.

The present paper presents a termination problem of the APO algorithm. The proposed correction solves the specific problem that was presented. However, there still lies a question mark over the completeness of the algorithm. Moreover, the proof of soundness of the algorithm does not seem to be robust enough to deal with different versions of the APO algorithm.

References

1. Michael Benisch and Norman Sadeh. How (not) to choose mediators for distributed constraint satisfaction. In *Proceedings of LSMAS at AAMAS'05*, 2005.
2. Michael Benisch and Norman Sadeh. Examining dcsp coordination tradeoffs. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'06)*, pages 1405–1412. ACM, 2006.
3. Roger Mailler and Victor Lesser. Using cooperative mediation to solve distributed constraint satisfaction problems. In *Proceedings of the Third International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS'04)*, pages 446–453. ACM, 2004.
4. Roger Mailler and Victor Lesser. Asynchronous partial overlay: A new algorithm for solving distributed constraint satisfaction problems. *Journal of Artificial Intelligence Research (JAIR)*, 25:529–576, 2006.
5. Amnon Meisels and Roie Zivan. Asynchronous forward-checking for distributed csps. In *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.
6. Michael Orlov and Amnon Meisels. Descending requirements search for discsps. In *Proceedings of Distributed Constraint Satisfaction Workshop at ECAI-2006*, Riva del Garda, August 2006.
7. Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *ICDCS*, pages 614–621, 1992.
8. Roie Zivan and Amnon Meisels. Concurrent dynamic backtracking for distributed csps. In *CP-2004*, pages 782–787, Toronto, 2004.