Workshop 18

The Twelfth International Workshop on Distributed Constraint Reasoning

DCR 2010

Editors:
Wiebe van der Hoek
Gal A. Kaminka
Yves Lespérance
Michael Luck
Sandip Sen
Distributed Constraint Reasoning Workshop

Held at AAMAS, Toronto, Canada

May 11th, 2010

Introduction

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.

The new challenges posed by solving Distributed Constraint Reasoning Problems include dealing with resource restrictions (such as limits on time and communication), privacy requirements, exploiting opportunities for cooperation, and designing conflict resolution strategies.

The workshop addresses modeling, formulation and solution of Distributed Constraint Reasoning problems, including both Distributed Constraint Satisfaction and Optimization Problems. The workshop encourages submissions on all topics related to Distributed Constraint Reasoning, including (but not limited to) the following:

- unified frameworks for DCR;
- complete and incomplete algorithms for solving DCR problems;
- privacy issues in DCR;
- problem solving in systems with self-interested agents;
- negotiation among self-interested agents;
- distributed constraint propagation and consistency;
- generation and formulation/modeling of DCR; and
- applications of distributed constraint reasoning.

These proceedings consist of one short paper and nine full papers and include applications of DCR, search and extensions to DCR, DCR under uncertainty and new approaches in DCR.

We are also excited to have this year’s workshop include a demonstration of DCR running on a sensor network by Alessandro Farinelli (University of Verona) and Alex Rogers (University of Southampton).

Finally, we extend our gratitude to the members of the program committee and the authors, without whom this workshop would not be possible.

Robert N. Lass & Evan A. Sultanik
Workshop co-chairs
May 2010
Organization

Workshop Co-Chairs

Robert N. Lass and Evan A. Sultanik
Department of Computer Science
College of Engineering
3141 Chestnut St
Philadelphia, PA 19104
{urlass,eas28}@cs.drexel.edu
http://www.cs.drexel.edu/~{urlass,eas28}

Program Committee

– Christian Bessiere (bessiere@lirmm.fr), LIRMM-CNRS
– Boi Faltings (boi.faltings@epfl.ch), Swiss Federal Institute of Technology
  Lausanne (EPFL)
– Rachel Greenstadt (greenie@eecs.harvard.edu), Drexel University
– Katsutoshi Hirayama (hirayama@maritime.kobe-u.ac.jp), Kobe University
– Robert Lass (urlass@cs.drexel.edu), Drexel University
– Amnon Meisels (am@cs.bgu.ac.il), Ben-Gurion University
– Pedro Meseguer (pedro@iiiia.csic.es), Consejo Superior de Investigaciones
  Científicas (CSIC)
– Adrian Petcu (adrian.petcu@epfl.ch), Swiss Federal Institute of Technology
  Lausanne (EPFL)
– Marius Silaghi (msilaghi@fit.edu), Florida Institute of Technology
– Evan Sultanik (eas28@cs.drexel.edu), Drexel University
– William Yeoh (wyeoh@usc.edu), University of Southern California
– Makoto Yokoo (yokoo@is.kyushu-u.ac.jp), Kyushu University
– Roie Zivan (zivanr@cs.bgu.ac.il), Carnegie Mellon University
## Table of Contents

Application of Max-Sum Algorithm to Radar Coordination and Scheduling  . 5

Distributed Scheduling Using Constraint Optimization and Multiagent Path Planning.  ................................................................. 20

AdCSP — Cooperative Ad Auctions with Privacy Guarantees  . . . . . . . 33

SymDPOP: Adapting DPOP to exploit partial symmetries  . . . . . . . 38

Saving Messages in ADOPT-based Algorithms  . . . . . . . . . . . . . .  53

Concurrent Forward Bounding for DCOPs  . . . . . . . . . . . . . . . . . . . 65

Cooperation between search and surveillance agents in DCOP-MST  . . . 80

Towards a Theoretic Understanding of DCEE  . . . . . . . . . . . . . . . 95

Asynchronous Partitioning Framework  . . . . . . . . . . . . . . . . . . . . 110

Effect of DisCSP Variable-Ordering Heuristics in Scale-free Networks . 125
Application of Max-Sum Algorithm to Radar Coordination and Scheduling

Yoonheui Kim¹, Michael Krainin², and Victor Lesser¹

¹ University of Massachusetts at Amherst, MA 01003, USA, ykim@cs.umass.edu
² University of Washington, Seattle, WA 98195

Abstract. The Max-Sum algorithm is a constraint optimization algorithm using message passing that can be configured to work in an approximate mode. It has been shown that in this mode it compares well with other approximate distributed optimization approaches. We used the approximate Max-Sum algorithm to coordinate adaptive radars for real-time weather sensing. When applied in various settings of the system with different types weather phenomena scenarios and radar configurations focusing on the utilization of structures in a distributed clustered organization, the Max-Sum algorithm performs efficiently in terms of resource utilization and communication for most of them in comparison to a negotiation-based algorithm specifically designed for this domain structure. We also modified the Max-Sum algorithm to utilize the domain structure and have found that this utilization further improves the algorithm performance. Finally, we show that the Max-Sum is robust to initial policies.

1 Introduction

The Max-Sum algorithm is a message-passing based algorithm that maximizes a utility function of a constraint graph and is proven to work well for solving a certain class of constraint optimization problems [1]. It is also suitable for a distributed setting as the algorithm computes an optimization function by communicating the partial results with direct neighbors. It has been proven to work well in an approximate mode where the solution is not guaranteed to be optimal due to cycles in the constraint graph.

This paper explores the distributed Max-Sum algorithm for radar scheduling in the NetRad system [3], which is designed for detecting and monitoring hazardous weather phenomena in real time. The NetRad system at the highest level is organized as a collection of controllers, each responsible for scheduling a cluster of radars based on the evolving weather scenario. Each radar’s scanning strategy can be dynamically configured every 60 seconds. The radars are geographically located to overlap so that multiple radars can concurrently scan phenomena to obtain dual-Doppler velocity vector measurements where desired.

There is a limit in the amount of weather volume that each radar can scan per cycle of 60 seconds and the radars have to coordinate to choose a highest-utility
phenomena to scan each cycle so as to optimize the overall utility of the system. Modeling the coordination and scheduling of radars as a constraint optimization problem and applying the Max-Sum algorithm is suitable for this domain for two main reasons.

First, the Max-Sum algorithm can be used to generate a real-time radar scheduling policy. Since it quickly reaches a close-to-optimal solution exhibiting an anytime algorithm property. In this domain, weather phenomena change dynamically, and therefore the scanning strategy needs to be repeatedly recomputed.

Second, the radar scan scheduling problem can be naturally modeled as a constraint optimization problem. In the system, each weather phenomenon is regarded as a task with an associated utility. The quality of a radar scan depends on the scope of space covered of the weather phenomena. A single radar is often not able to scan all the phenomena in its range and thus benefits from collaborating with neighboring radars that can potentially scan the desired region. Coordination of radars are necessary due to pinpointing tasks in addition to the lack of resources e.g. time for scanning. Between two types of tasks, pinpointing and non-pinpointing tasks, pinpointing tasks require more than one radar to scan the phenomena, therefore tighter coordination among radars is necessary.

The main purpose of the work is to study the behavior of the Max-Sum algorithm in the problem domain so as to understand the performance of the Max-Sum algorithm in comparison to other algorithms including a previously developed negotiation algorithm [7] designed for the application. A key issue that we want to study is whether the optimization should be broken along the lines of the grouping of radars into clusters, which leads to a two-stage optimization process where we first optimize scheduling within a cluster and then modify that optimization based on interactions with other clusters. This two-stage process exploits the low communication overhead possible when optimization is done in the first stage. We contrast this with an approach that sees the optimization as a single integrated process that does not explicitly take advantage of the grouping. To understand this issue, we have tried evaluating the performance of two different algorithms: one which directly exploits the decomposed organizational structure and the other which does not. Additionally, we tried to improve the Max-Sum algorithm by providing a better starting point for the optimization process based on the results of a local optimization algorithm specific to each radar cluster.

The paper is organized as the following: In the next section, we introduce the NetRad system and formulate the constraint optimization problem we are trying to solve. The third section briefly overviews the Max-Sum algorithm and describes two approaches to formulate the optimization problem and also describes a modified Max-Sum algorithm which uses an initial policy. Then, we present the experimental results. Finally, we summarize the major conclusion of the paper and discuss future work.
Fig. 1. Example weather scenario and radar settings (a). Example configuration of Radars (b). All radar ranges and phenomena are assumed circular shaped. All phenomena locations and sizes are randomly selected. In (b), Radar 1 (R1) can choose to scan event 1 (Ev1), event 2 (Ev2) or to scan both depending on the utility. Scanning all phenomena in range may not be possible given the time limit to scan.

2 NetRad System and Problem Formulation of Radar Scanning Policy

2.1 The NetRad system

The NetRad System is a system of radars specially designed for the purpose of quick detection of low-lying meteorological phenomena such as tornadoes. They are short-range radars used in dense networks, thereby alleviating blind-spots caused by the curvature of the Earth. NetRad radars additionally do not just use the traditional sit-and-spin strategy; rather, they can be focused to scan in a particular volume of space. By exploiting the collected information on weather phenomena, scanning strategies can be dynamically created for specific weather phenomena in the current environment.

The NetRad system consists of multiple MCCs (meteorological command and control) each of which controls a set of radars. The MCC system is a closed-loop control system in that it responds to the emerging weather events based on detected features in the radar data and end-user concerns that may vary over time. End-users such as forecasters, emergency managers, and researchers can provide information as to what sort of data they are looking for and how frequently. Consequently, the MCC ranks the importance of tasks so as to give preference to the data users want.

The MCC gathers moment data from the radars and then runs detection algorithms on this weather data. The result of this analysis leads to a set of weather-scanning tasks of interest for the next radar scanning cycle. The MCC then determines the best scanning strategy for the available radars that will maximize the sum of the utility associated with the chosen tasks according to a utility function based on the end-user priorities. This scan strategy is used by the NetRad radars on the next cycle.
Tasks may also be either pinpointing or non-pinpointing, meaning either there is, or is not, a significant gain by scanning the associated volume of space with multiple radars at once. The utility gained from scanning a pinpointing task increases with the number of radars scanning the task; the utility for a non-pinpointing task is the maximum utility among the individual radars that scan the same phenomenon. The global utility is simply the sum of utilities of all tasks. For a more in-depth description of the MCC system, see [2], or see [3] for more details on the utility function.

2.2 Problem Formulation of Generating Radar Scan Policy

The goal of the system is to maximize the overall utility by summing the utility calculated for each scanned task. Each radar can choose where to scan by choosing a subset of phenomena in its range. The possible scanning strategies of a radar are discretized. For each task that a radar can scan, there is a preferred scan: one which covers the task region most tightly. The preferred scan for each task is in the scan set as well as mergers of each pair of preferred scans into new, combined sweeps.

For each phenomenon $t_i$, the utility function

$$u_i : t_i \rightarrow r \in \mathbb{R}, \text{ where } 0 \leq r \leq 1$$  \hspace{1cm} (1)

is determined by the priority of the requesting user or the weather pattern.

Also, there is a function for the quality of scan for each phenomenon $t_i$

$$q_i : t_i \times (r_1, \ldots, r_n) \rightarrow r \in \mathbb{R}, \text{ where } r_j \text{ denotes the scanning policy of radar } j.$$  \hspace{1cm} (2)

The radars in the quality function argument are limited to the radars which have the phenomena in range.

The system utility $U$ is defined as

$$U = \sum_i u_i \times q_i.$$  \hspace{1cm} (3)

and thus the goal of the system at each scan cycle is to find the configuration of radars $r_1, \ldots, r_n$ which maximizes the system utility $U$. This problem easily maps into a constraint optimization problem where there is a variable corresponding to each radar whose value specifies which of the possible sweeps the radar should execute on the next scan cycle.

3 Max-Sum algorithm and Modeling on Radar Scanning Problem

3.1 The Max-Sum algorithm

Max-Sum is a distributed message-passing optimization algorithm belonging to the class known as Generalized Distributive Law (GDL) [4]. Max-Sum is a variation of Sum-Product algorithm but tries to maximize the global utility function.
In the Max-Sum algorithm, there is a set of variables $x = \{x_1, x_2, \ldots, x_m\}$ on which a set of functions $F = \{F_1, F_2, \ldots, F_n\}$ depend. Each function $F_i = F_i(x_i)$, $x_i \subset x$. The goal is to find $x^*$ which satisfies the following:

$$x^* = \arg \max_x \sum_{i=1}^{n} F_i(x_i)$$ (4)

Therefore, the Max-Sum algorithm can be viewed as a constraint optimization algorithm and we are looking for the settings of variables which maximize the sum of a set of local utility functions. To achieve this, the Max-Sum algorithm defines a factor graph by creating a node for each variable and for each function. The graph is bipartite, and a function node is connected to a variable node if the corresponding function is dependent upon that variable. The bulk of the algorithm is in the messages passed between nodes, which are:

**Variable $i$ to Function $j$:**

$$q_{i\rightarrow j}(x_i) = \alpha_{ij} + \sum_{k \in M_i \setminus j} r_{k\rightarrow i}(x_i)$$ (5)

Here $\alpha_{ij}$ is a scalar set such that $\sum_{x_i} q_{i\rightarrow j}(x_i) = 0$, and $M_i$ contains the indices of function nodes connected to variable node $i$.

**Function $j$ to Variable $i$:**

$$r_{j\rightarrow i}(x_i) = \max_{x_j \setminus i} [F_j(x_j) + \sum_{k \in N_j \setminus i} q_{k\rightarrow j}(x_k)]$$ (6)

where $N_j$ contains the indices of variable nodes connected to function node $j$ in the factor graph.

If the factor graph is cycle-free, then the messages are guaranteed to converge, and the resulting solution will maximize $\sum_{i=1}^{n} F_i(x_i)$. When the graph contains cycles, the messages may not converge, and even if they do the resulting solution may be sub-optimal. Empirical results show that even in this case, the algorithm frequently provides good solutions [1].

### 3.2 Modeling Max-Sum in the NetRad system

We explored two formalizations to implement Max-Sum in the NetRad system. The first of two formalizations (which we call coarse-grained) is based on the one proposed in [5] where a variable and a function node is created for each MCC. The variable node represents the joint scanning strategy of all radars controlled by the MCC and the domain size of this single variable is quite large. The function node represents the utility for all tasks which can be performed by the MCC. A function node is connected to a variable node if the function could conceivably depend on the variable, that is, if the MCCs are the same or neighbor each other. As some of the tasks are shared by multiple MCCs, each function node considers the same tasks multiple times, so we divide the utility of a task by the number of function nodes counting it. The function node considers
configurations of directly connected radars in the neighboring MCCs to minimize the size of configurations considered in each function node.

The second formalization (which we will call fine-grained) disregards the MCC organization and works at a finer granularity of individual radar rather than at the MCC level. Each radar is regarded as its own variable whose domain is the set of allowable scans it may perform. Each task is its own function node whose value is the single-task utility discussed in Section 2. A function node is connected to a variable node if the radar is capable of scanning the weather phenomenon associated to the task.

The two formalizations have their own advantages. The factor graph for the first formalization is independent of the number of tasks in the system and even to some extent how the radars are assigned to MCCs. Additional tasks only increase the size of the variable domains in the first formalization, not the number of function nodes to which variable nodes are attached.

The first formalization has much bigger domains for the values each variable and function node can take. Without well-designed heuristics, the computational complexity is too high to be applied directly. With 10 values for each variable nodes, the common size of the domain for function nodes is $12^{10}$ where 4 radars belong to one MCC (configuration of 4 radars in an MCC and 8 neighboring radars). This makes the straightforward application of the first formalization undesirable in the on-line settings and a heuristic to cut down the complexity should be used. Therefore, we mainly experimented with the fine-grained formulation of the Max-Sum algorithm for this application domain.

### 3.3 The Max-Sum algorithm with Initial Policy

In the Max-Sum algorithm, a node’s outgoing messages are dependent upon the incoming messages it received in the prior cycles. At the start of the algorithm, no messages have been sent, so certain initial values must be used. The obvious choice is to set all values for messages not received to zero. In this way, initial variable node messages are uniform functions, indicating no preference for any specific variable assignment. A function node $j$’s initial message to a variable node $i$ indicates only the maximum local utility $F_j$ given $x_i$.

If we were to stop here, each variable $i$ would take on the value

$$\hat{x}_i = \arg\max_{x_i} \sum_{j \in N_i \setminus x_j \setminus i} \max_{x_j} F_j(x_j)$$

In other words, each variable would set itself in order to maximize the local utilities of neighboring functions assuming those functions get best-case settings of other variables for the local utility. This initial state is not only determined by a fairly local optimization, but it does so by assuming maximizations which may be neither mutually compatible nor even close to optimal for other parts of the factor graph.

As the algorithm proceeds, more global information begins to become incorporated in the messages; however, it is unclear whether encoding more global
information in the initial messages might lead to convergence to better solutions, faster convergence, or even convergence when the algorithm would otherwise fail to do so in loopy factor graphs.

We would like to be able to take some global variable assignment produced by some optimization technique and introduce it somehow as a starting configuration to replace that in Equation 7. To do so, we modified the algorithm to always start with function nodes sending messages at the beginning of the algorithm.

Once the optimization algorithm of choice has been used to construct a variable assignment \( \hat{x} \), function nodes send the following messages to the connected variable nodes. Here \( j \) is the index of the function node and \( i \) that of the variable node.

\[
r_{j\rightarrow i}(x_i) = F_j((\hat{x}_j \setminus i) \cup x_i) \quad (8)
\]

This is the value of \( F_j \) when \( i \) has state \( x_i \) and all other variables use their states from the solution \( \hat{x} \).

After receiving these messages, if a variable node were to take on a value, it would be:

\[
\tilde{x}_i = \arg \max_{\tilde{x}_i} \sum_{j \in N_i} F_j((\hat{x}_j \setminus i) \cup x_i) \quad (9)
\]

**Proposition 1** If the assignment \( \hat{x} \) is such that no individual variable can by itself change its value to increase the global utility, then \( \hat{x} \) is a solution to the assignment constraints imposed by Equation 9. If changing any individual variable’s value will strictly decrease the global utility, then \( \hat{x} \) is the unique solution for Equation 9.

**Proof.** From the perspective of an arbitrary variable node \( i \), all other nodes are fixed to the configuration specified by \( \hat{x} \). Maximizing \( \sum_{j \in N_i} F_j((\hat{x}_j \setminus i) \cup x_i) \) leads to maximize the global utility given the values of other variables. This is because only the functions for nodes \( j \in N_i \) are affected by \( x_i \).

If \( \hat{x}_i \) were not a solution to this, then the algorithm which selected \( \hat{x}_i \) to be part of \( \hat{x} \) could have instead selected \( \tilde{x}_i \) to receive a higher utility. Since by supposition, no individual variable can change its value to increase the global utility, \( \hat{x}_i \) is a solution to Equation 9. If changing any variable’s value in \( \hat{x} \) will decrease the global utility, then there can be only one solution to Equation 9. Since \( \hat{x}_i \) is a solution, it must be the unique solution.

Thus, in the sense of the above property, we can insert a variable assignment into a factor graph as a starting solution. The property requires that no single variable can change its value to increase the utility. This is a desirable property for an optimization algorithm to have, and a fairly lax one. Any algorithm which does not satisfy this constraint can be followed by a hill-climbing procedure in order to meet the requirement of Property 1.

In addition to what Property 1 can tell us, Equation 9 by itself looks quite a bit better than Equation 7. While the assignment still only considers directly neighboring function nodes, it does so using better assumptions. For nodes other
than itself, it assumes a configuration that is known to exist rather than a separate maximization for each function node. The assumed variable assignments are also known to be consistent with a good global utility, and \( x_i \) will fit itself into this assignment.

After the messages from function nodes to variable nodes, we allow the variable nodes to send one set of messages before proceeding with the regular algorithm. This is so the next set of messages from function nodes will have a starting point other than assuming uniform functions in variable node messages.

In this paper, we focus on the performance of Max-Sum for a single snapshot of weather and do not deal with dynamics of system; it is interesting to know how in a dynamic setting the previous result could be used in the next cycle.

4 Experimental Results on the Performance of Max-Sum on NetRad

We experimented with the Max-Sum algorithm described in Section 3 in various settings. The purpose of our experiments were twofolds. We are particularly interested in understanding how the algorithms perform in the cluster-organizational setting where the whole network of radars is decomposed into several clusters and each cluster is managed by an MCC. We also ran tests for the Max-Sum algorithm with an initial policy and studied how the initial policy affects performance.

4.1 Experimental Setting

To run our experiments, we used an abstract simulation environment of the NetRad radar system developed in the Farm simulator framework [6]. In this simulator, weather tasks are abstracted as circular areas as shown in Figure 1(a), and time is discretized into system heartbeats. Aspects such as the utility function, the effective range of radars, and the separation between radars, however, are the same as in the operational testbed. For more information on the simulation environment, see [7].

We compared the results of several optimization techniques over a number of trials; what varies is the number of radars, number of weather phenomena in the environment and spatial extent of weather phenomena. To make the results more easily interpretable, each trial is run for only one scan cycle. By only being concerned with one scan cycle in isolation, the difference among approaches can be seen more clearly; if a trial is involved multiple scan cycles the result of previous scan cycles decisions will affect the set of tasks available in the next scan cycle; thus making a direct comparison among approaches more difficult.

4.2 Performance of Max-Sum in Two Modeling

We first experimented with the two modelings of Max-Sum described in Section 3.2 on the 12-radars network. The problem with 12 radars is trivial in a sense
that different algorithms found the same solution with the same utility, but the coarse-grained version takes too much time because of the computation in large function nodes. The straightforward application to the cluster-organizational structure in the coarse-grained model leads to difficult subproblems with large domains in function nodes. This is a known problem with the Max-Sum algorithm: the complexity of function nodes increases exponentially very easily with the number and domain size of connected variable nodes and heuristics should be used to reduce the complexity. It is better to avoid such exponential explosion.

Although the coarse-grained model is formulated along the decomposed models with two steps of local and global optimization aligning the system structure, the model does not exploit the full benefit of simple local computation of Max-Sum whereas the fine-grained formulation does.

In addition, the performance of the fine-grained Max-Sum algorithm in the 48-radar setting shows its anytime property where it converges to a close to optimal solution very quickly.

4.3 Performance of the Max-Sum algorithm on NetRad System in Comparison to Other Algorithms

In this section, we evaluate the performance of several alternative optimization algorithms, while varying the number of radars and the number of phenomena to show the performance of the Max-Sum algorithm in the domain with the cluster-organizational structure. We compare the performance of fine-grained implementation of Max-Sum decentralized message-passing algorithm to a decentralized negotiation algorithm [7] and a centralized optimization algorithm based on a genetic algorithm that is currently used for local optimization in the negotiation algorithm in each MCC.

The negotiation algorithm, specifically developed for the NetRad problem domain, is an iterative two step process performed concurrently at each MCC. In the first step each MCC performs a local optimization based on its local tasks
and knowledge of how neighboring MCCs scan schedules are related to its local tasks. In the second step, the MCC negotiates with their neighbors so as to make adjustments to their scheduling based on the strategy of other MCCs. This two step process for performing the distributed optimization tries to maximize the parallelism at the MCC level and to minimize communication among MCCs.

In contrast, the Max-Sum algorithm does not consider such an organizational structure and is completely decentralized. The Max-Sum algorithm does not explicitly take into account that certain links are within an MCC cluster and others are between MCC clusters. The genetic algorithm uses a centralized approach; no communication is required and it only utilizes one processor.

**Performance for Different Network Sizes** In order to evaluate the general performance and the scalability of the algorithms, we compare the performance on different sized networks. In the scenarios, there are the same number of phenomena as the number of radars in the network; the size and location of the phenomena are randomly chosen. The result is shown in Figure 3. The performance quality of Negotiation and Max-Sum is close for all network sizes whereas the performance of the centralized genetic algorithm is always inferior to these algorithms. This result show that the Max-Sum algorithm is able to handle the problem well without explicitly exploiting the clustering of radar controllers.

Each bar in Figure 3(b) shows the total time taken to run the algorithm in a single processor and shows that the Max-Sum algorithm quickly converges to a solution as good as the result of Negotiation algorithm. However, the result does not consider any benefit of decentralized computation of both algorithms and the estimated time for the decentralized setting is given as well.

The lower stacks in Figure 3(b) represents the estimated computation time when the algorithms are run in the decentralized setting. The estimated time is computed as the sum of the longest local computation time for each round. In both negotiation algorithm and the Max-Sum algorithm, each MCC, when the local computation is done, waits for other MCCs to finish the computation and then they exchange messages. Therefore, the time complexity in the decentralized setting results from the sum of the longest time taken in the local computation for each round. The Max-Sum algorithm benefits in terms of decreased computational time and higher quality from the conceptually more decentralized computational structure though it pays some cost in terms of additional communication among MCCs and still have a comparable performance to the negotiation algorithm.

In addition, in order to assess the communication burden of Max-Sum, we measured the number of messages exchanged across MCCs and compare it with that of negotiation algorithm as in Figure 3(d). When only messages exchanged across MCCs are counted, Max-Sum needs more than twice the communication than the negotiation algorithm. In Section 4.4, we will discuss a variant of the Max-Sum algorithm (MS2L) that has comparable communication efficiency as the negotiation algorithm without sacrificing its computational efficiency.
Fig. 3. Gen: Genetic MS: Max-Sum Neg: Negotiation MS2L: Max-Sum-2-Level MS-MCC: Max-Sum across MCCs only, and MS2L-MCC: Max-Sum-2-Level across MCCs only. The performance (a) and time complexity (b) and estimated computation time in a decentralized setting (c) of algorithms on different sizes of the network given the same number of tasks (weather phenomena) as the number of radars. The lower stacks in (b) represent the estimated computation time in a decentralized setting which are also separately shown in (c). It shows that the fine-grained version of the Max-Sum algorithm is able to solve the problem with a similar quality to the negotiation algorithm more quickly. (d) The number of messages of negotiation and Max-Sum per MCCs. Max-Sum across MCCs only represents the number of messages sent across MCCs in Max-Sum. (e) Total amount of communication in terms of the average size of messages × the number of messages. Message size is measured in terms of number of utility values in the messages.
Performance varying with the ratio of the number of phenomena to the number of radars In the next experiment, we increase the number of phenomena in a 48-radar network, thereby requiring more coordination between radars and studied how the algorithms perform. While the quality of solution of Max-Sum is slightly better, the time complexity of the Max-Sum algorithm sharply increases because the number of function nodes in Max-Sum increases as more weather phenomena are added.

![Figure 4](image_url)

(a) Performance Quality  (b) Time Complexity  (c) Number of Messages

Fig. 4. The performance quality (a), time complexity (b) and number of messages (c) of algorithms on different number of phenomena. The basis is 48 weather phenomena and this is increased to 120 phenomena.

When the number of phenomena increases, the maximum number of variable nodes per each function nodes increases. This leads to a computational complexity hike in Max-Sum as shown in Figure 4(b). Also, the number of messages across MCCs increases as there are more tasks shared by multiple MCCs in the environment. In contrast, number of messages in the negotiation algorithm decreases due to the failed negotiations resulting in early termination. It is observed throughout the experiments that the negotiation algorithm terminates early with a suboptimal solution after one or two negotiations when the problem gets difficult as shown in this experiment.

4.4 Performance of Max-Sum in a Two-Level Hierarchy

In the previous results, the fine-grained version of the Max-Sum algorithm shows good performance in the domain without exploiting the cluster-organizational structure. We then modified the algorithm to exploit this system structure and experimented on the Max-Sum algorithm with a two-level hierarchy. That is, we ran the Max-Sum algorithm on the MCC-level factor graphs consisting of only the nodes in each MCC for three rounds and then optimized on the global-level factor graph as normal.

The result in Figure 3 shows that Max-Sum in two-level actually beats the performance of the regular Max-Sum algorithm in every aspect we experimented on. The performance quality remains similar to Max-Sum and the time complexity decreases. The computation on the factor graph with local nodes only is much simpler than on the global-level factor graph and also the result of this computation leads to a quicker convergence on the global level.
As messages are exchanged only within MCCs for three rounds, the number and size of messages also decreases. The result in Figure 3(d) shows that the number of messages of Max-Sum in two level is smaller than negotiation algorithm. This result indicates that in this domain Max-Sum does not necessarily need to exploit the cluster-organizational setting, but the use of the setting benefits the algorithm.

4.5 Performance using Initial Policy

As a final set of experiments, here we present the result on an extension of the Max-Sum algorithm described in Section 3.3. As in Section 3.3 we tried to augment the algorithm with an initial policy for guiding the optimization process. For testing the algorithm with an initial policy, we tested two kinds of initial policy, one random and the other generated by a centralized optimization - a genetic algorithm. As explained, this initial policy is given to function nodes, which send messages that lead the variable nodes to have preference for a scan policy maximizing the utility given the initial policy of other variable nodes.

To show the effect of the initial policy on the Max-Sum algorithm, we started the algorithm with initial random policies for two specific weather scenarios. As in Figure 5(a), the final solution quality does not vary even with different initial policies. This shows that the Max-Sum algorithm is resilient to initial policies even with cycles in the factor graph.

Additionally, we initialized the Max-Sum algorithm with a better initial policy since randomly generated initial policy can be very ineffective. We took the solutions from the genetic algorithm for each MCC whose value is suboptimal in comparison to other algorithms but generally better than random policies and give them as initial configurations. As shown in Figure 5, the performance does not improve except for a reduction in time. However, including the time taken to compute the policy negates the gain in time. This result is interesting in that...
Max-Sum is very resilient to the initial policy and even when it is given a skewed value during the computation, the algorithm still converges to a similar solution.

5 Conclusion

The Max-Sum algorithm is an approximate constraint optimization algorithm using message passing. We applied the algorithm in the NetRad system for coordinating weather-sensing radars. In this system, fine-grained modeling of Max-Sum worked well even without utilizing the cluster-organizational structure while not requiring much more communication than negotiation algorithm. Also, the coarse-grained modeling of Max-Sum has shown the limitation of utilizing the decomposed structure as the local computational burden is too high. We also applied Max-Sum in the two-level hierarchy which works with the local nodes within the MCC at the beginning and later expand the scope to the whole network. This version of the Max-Sum algorithm proved the benefit of using the organizational structure, spending less computation time and less number of messages than all other algorithms. Finally, we applied initial policies designed to improve the algorithm. However, the Max-Sum algorithm showed that it is very resilient to a transient preference towards a given configuration.

The Max-Sum algorithm has a well-known limitation of computational explosion for function nodes and our experiment with the fine-grained version also suffers from such explosion when the the number of phenomena and radar radius are increased. This work on the Max-Sum algorithm suggests some directions for future research. One is to study ways for decomposing the function nodes connected to many variable nodes. In a series of experiments on various settings not shown in this paper due to space limitations, the algorithm suffer from computation explosion as the number of phenomena and radar radius increases. Although many heuristics were proposed to guide the search in the function node itself, it would be beneficial to minimize the size of the function nodes by decomposing the search space into smaller ones. For instance, the function nodes in the domain can be decomposed into a set of smaller function nodes for each radar for non-pinpointing tasks as the final utility of such tasks comes entirely from a single radar.

Another possible direction is a decomposition of factor graph where the connection is loose. Temporarily restricting the connections to a subset of variable nodes based on the factor graph will greatly reduce the complexity of the function nodes. This will also reduce the communication burden given on the Max-Sum algorithm as well as the computational complexity.

Finally, as shown in the experiment on initial policies, Max-Sum is very resilient to a transient preference for a specific configuration. This might also mean that in a dynamic environment when tasks are changing, Max-Sum could adapt to the change well as it can find the same final solution from any starting point. Also, in a sense that the problem is loosely connected across the whole network, a local change in the problem would not affect the solution globally and only partial solution should be recomputed.
Acknowledgement

This work was supported in part by the Engineering Research Centers Program of the National Science Foundation under NSF Cooperative Agreement No. EEC-0313747. Any Opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the National Science Foundation.

We would like to thank Alessandro Farinelli, Alex Rogers, and Ruben Stranders of the University of Southampton for their assistance in this project. Their suggestions such as branch and bound search helped us to efficiently implement the Max-Sum algorithm. They also suggested the fine-grained formalization as an alternative to our initial approach.

References

Distributed Scheduling Using Constraint Optimization and Multiagent Path Planning

Christopher T. Cannon\(^1\), Robert N. Lass\(^1\), Evan A. Sultanik\(^1\), William C. Regli\(^1\), David Šislák\(^2\), and Michal Pechouček\(^2\)

\(^1\) Department of Computer Science, CoE, Drexel University
{ctc82, urlass, eas28, regli}@drexel.edu
\(^2\) Agent Technology Center, FEE, Czech Technical University in Prague
{david.sislak, michal.pechoucek}@agents.felk.cvut.cz

Abstract. The goal of the distributed scheduling problem is to minimize the global cost of assigning \(n\) decentralized workers to \(m\) tasks at time points. This problem is further complicated in continuous environments because the entire state space cannot be searched. This paper presents a decentralized approach of dividing the distributed scheduling in continuous environments problem into two subproblems: distributed set covering and distributed multiagent path planning. First, we represent the problem of assigning workers (i.e., covers) to tasks (i.e., sets) as a Distributed Constraint Optimization Problem (DisCOP). Then, the DisCOP solver passes its solution to the distributed multiagent path planner who creates a conflict-free path for each worker to its assigned tasks. By first representing the problem as a DisCOP, it restricts the plan space to only a set of feasible plans. We apply this approach to the scenario of distributed scheduling for unmanned aerial vehicle surveillance. This approach is shown to relieve the strain on the distributed multiagent path planner by reducing the plan space more so than other distributed approaches.

1 Introduction

The goal of the distributed scheduling problem is to minimize the global cost of assigning \(n\) decentralized workers to \(m\) tasks at time points. This problem is further complicated in three-dimensional continuous environments because the physical state space cannot be completely searched.

As an example of a distributed scheduling problem, consider the real-world domain of distributed sensor networks [11, 15]. The distributed sensor network target tracking problem, shown in Figure 1, consists of four agents each equipped with Doppler radar sensors whose goal is to track moving targets. Each agent’s sensor has three sectors that only one of which can be active at any given time. To accurately track a target, the target must lie within at least three agent’s active sensors. In this example, the agents are the workers and the sensor sectors are the tasks. The agents schedule which sector of their sensor to activate
This paper presents a decentralized approach of dividing the distributed scheduling in continuous environments problem into two subproblems: distributed set covering and distributed multiagent planning. To solve these subproblems, we utilize previous research in these areas and select the best performing techniques. First, we represent the distributed set covering problem as a Distributed Constraint Optimization Problem (DisCOP). Then, we represent the distributed multiagent planning problem as a distributed multiagent path planning problem. As shown in Figure 2, the approach consists of four steps:

1. The set covering problem elements are represented as workers and tasks (e.g., a cover and set are equal to a worker and task, respectively) and costs for assignment are computed, which form the constraint graph of the DisCOP;
2. The DisCOP solver passes its solution to the distributed multiagent path planner;
3. The distributed multiagent path planner creates a conflict-free path for the workers to execute their assigned tasks; and
4. If an element of the problem changes (e.g., a new worker, task, or cost), the distributed multiagent path planner passes the new set covering problem to the DisCOP solver and the process repeats.

The contributions of this paper are:
A formalization of using DisCOP and multiagent planning for distributed scheduling in continuous environments;
- An example of mapping this formalization to the problem of unmanned aerial vehicle (UAV) surveillance; and
- A comparison of this approach to other fully distributed techniques.

The remainder of this paper is organized as follows: first we formalize the approach to solving the distributed scheduling problem. Then we map this formalization to the distributed scheduling problem of UAV surveillance. Using a flight simulation testbed, we analyze this approach against other fully distributed techniques. Finally, a discussion of future work and concluding remarks.

2 Formalization

This section presents a formalization of using DisCOP and multiagent planning to solve the distributed scheduling problem.

2.1 DisCOP

There are four components to a Distributed Constraint Optimization Problem (DisCOP): a set of agents $A = \{a_1, a_2, \ldots, a_n\}$, a set of variables that are to be assigned values by the agents $V = \{v_1, v_2, \ldots, v_{|V|}\}$, a set of domains that contain the values that may be assigned to said variables $D = \{D_1, D_2, \ldots, D_{|V|}\}$, and a set of constraints over the variables’ assignments. A constraint function for a pair of variables $v_i, v_j$ is defined as $f_{ij} : D_i \times D_j \rightarrow |V|$. The objective is to have the agents assign values (taken from the domains) to their variables such that some metric over the resulting constraints’ values is either minimized or maximized.

2.2 Multiagent Planning

There are seven components to the domain of a multiagent planning problem: a set of agents $A = \{a_1, a_2, \ldots, a_n\}$, a set of propositions that are the agents beliefs

---

3 Adapted from Modi et al. [16].
4 Adapted from Bowling et al. [4].
$\Phi = \{ \phi_1, \phi_2, \ldots, \phi_{|\Phi|} \}$, a set of valid states that are all possible combinations of said propositions, $S = \{s_1, s_2, \ldots, s_{|S|}\}$ where $S \subseteq 2^\Phi$, a set of initial states $\Theta = \{\theta_1, \theta_2, \ldots, \theta_{|\Theta|}\}$, a set of actions for $a_i \in A$, $\Gamma_i = \{\gamma_{i,1}, \gamma_{i,2}, \ldots, \gamma_{i,|\Gamma_i|}\}$, a transition relation of effects that are applied after said actions are executed, and a set of goal states $\Psi_i = \{\psi_1, \psi_2, \ldots, \psi_{|\Psi_i|}\}$.

2.3 DisCOP and Multiagent Planning Integration Formalization

Using the two previous formalisms, we formally show how the solution produced by the DisCOP solver prunes infeasible states (and associated actions) from the multiagent planning domain.

First, the DisCOP assigns intermediate goal states to agents: there is one DisCOP variable for each agent/state pair, the domain of which is a boolean indicator for whether or not that agent will set that state as a goal. Constraints in the DisCOP are domain specific, but generally dictate the cost of an agent pursuing a state (which may be based upon a heuristic). See Section 3.3 for an example of how costs are calculated for the UAV surveillance scenario. The solution to the DisCOP is then used to prune irrelevant states from the planning domain (i.e., states that are unreachable).

We first define the function $\text{associate}$ which prunes the multiagent planning problem’s domain based on the DisCOP’s domain. Formally,

$$\text{associate} : \Phi \times S \times \Gamma_i \times D_i \rightarrow D_i' \subseteq D_i.$$  

With the $\text{associate}$ function, we can define the pruned set of valid plan states. Formally, 

$$S' = \{s \in S : \text{associate}(s, D_i) \neq \emptyset\}.$$  

The function $\text{relevant}$ maps the solution context of the DisCOP and the set of actions from the planning domain to a set of pruned actions. Formally,

$$\text{relevant} : \bigcup_{Q \subseteq 2^V} \prod_{v_i \in Q} (\{v_i\} \times D_i) \times 2^{\Gamma_i} \rightarrow 2^{\Gamma_i},$$  

with the mapping explicitly defined as,

$$\text{relevant}(t, \Gamma_i) \mapsto \{\gamma \in \Gamma_i : \langle\gamma, \text{TRUE}\rangle \in t\}.$$  

In summary, we prune the plan’s domain, states, and actions based upon the DisCOP’s task assignments. We first accomplish this with the $\text{associate}$ function, which prunes the plan’s domain to that of the DisCOP’s domain to create a new plan domain. Then, we prune the plan states to only the states that are valid based on the new plan domain. Finally, with the $\text{relevant}$ function, we prune the plan’s actions based on the DisCOP’s variables and domain.
3 Case Study: UAV Surveillance

As an application of this technique, we present the problem of multiple UAVs monitoring a set of targets. This section provides an informal and formal description of the problem and maps it to the DisCOP and multiagent planning formalization described earlier. We also describe the algorithms used for each solver and how they interact.

3.1 Scenario

Consider a set of four UAVs equipped with cameras (or any other type of sensor) that are monitoring a set of twelve enemy targets. The main goal of this scenario is to minimize the amount of time between a UAV surveilling a target. A concern of this scenario is that an enemy will capture a UAV, read its on-board data, and determine which targets are of most interest to its opposition. To negate this action, each UAV's target assignments only consist of a subset of the total targets to add an element of privacy to the locations of targets. A trusted centralized target assignment source is not feasible because assignments must be rapidly generated and a communications link to a centralized source in this scenario would likely be low bandwidth and therefore high latency. The four UAVs are equally divided into Group A and Group B, and the twelve targets are equally divided into Group A Exclusive, Group B Exclusive, and Shared. To prevent the disclosure of all targets if an enemy captures a UAV, each UAV group is only assigned to monitor its own exclusive group and the shared group of targets (e.g., UAV Group A would monitor target Group A Exclusive and Shared).

3.2 Formalization

This problem is formally described as follows. Each UAV has a local agent $a \in A$, an entry position (where the UAV takes flight) $e \in E$, and an exit position (where the UAV lands) $x \in X$. Each UAV agent $a \in A$ surveils $k$ targets $T = \{t_1, t_2, \ldots, t_k\}$ throughout a period of time. Time is divided into $q$ rounds $R = \{r_1, r_2, \ldots, r_q\}$. The size of each round $r \in R$ is denoted as $\delta_R$. In the experiments conducted, the time quanta for $\delta_R$ is one minute. An agent $a$ has covered target $t$ if it is there for the entire duration of that round. If at least one agent $a$ covers target $t$ during round $r$ that target-round pair $(t, r)$ is assigned the value 1, otherwise 0. The domain for each agent $a_i$ consists of all pairs of targets and rounds: $D_i = T \times R$.

The function

$$empty(t_i) = \sum_{j=1}^{q}(r_j - t_{i,j})/\delta_R$$

outputs the total amount of uncovered time for target $t_i$. In the equation, $t_{i,j}$ is the last visited time an aerial vehicle was covering $t_i$ during round $r_j$. The goal of the scenario is to minimize the number of rounds that all targets remain
uncovered based upon a threshold value. Formally:

\[
\begin{align*}
\text{Minimize} & \quad \max_{i=1, \ldots, k, j=1, \ldots, q} \ell_{i,j} \\
\text{Subject to} & \quad \sum_{i=1}^{k} \text{empty}(t_i) \leq \text{Threshold Value}
\end{align*}
\]

### 3.3 UAV Scenario to DisCOP and Multiagent Planning Mapping

Mapping the UAV surveillance problem to the formalization of DisCOP and multiagent planning is as follows:

- **Agents**: Each UAV agent \( a_i \in A \) corresponds to the DisCOP and multiagent planning agent \( a_i \in A \).
- **Propositions**: Each proposition \( \phi_i \in \Phi \) corresponds to a target \( t_i \in T \).
- **States**: Each state \( s_i \in S' \), where \( S' \) is the pruned set of valid states from the `associate` function, corresponds to every combination of targets who are in the domain \( D_i' \) with values `TRUE(Covered)` or `FALSE(Not Covered)`.
- **Variables**: Each agent \( a_i \in A \) has a set of variables \( v_i \in V \) which contains a single variable \( v_j \) for each constrained target. These variables correspond to the DisCOP formalization of \( V \) and must be one of the valid combinations \( s_i \in S' \).
- **Domain**: Every \( D_i \in D \) is the same for each variable \( v_{i,j} \) is boolean, \( D_i = \{ \text{Covered}, \text{Not Covered} \} \).
- **Actions**: The actions \( \gamma_{i,j} \in \Gamma_i \) correspond to the combinations of all possible sequences of targets selected based upon the `relevant` function which specifies all targets set to `TRUE(Covered)`.
- **Initial State**: The initial state set \( \theta_i \in \Theta \) corresponds to the entry position set \( e_i \in E \).
- **Goal State**: The goal state set \( \psi_{i,j} \in \Psi_i \) corresponds to the exit position set \( x_i \in X \).
- **Constraints**: The cost of each variable assignment, where \( v_i \) and \( v_k \) represent two agents, and \( t_j \) is a single target is defined by the function

\[
\text{cost}(v_i, v_k, t_j) = \begin{cases} 
\text{Low Cost} & \text{if } v_{i,j} = v_{k,j} = \text{Covered}, \\
\text{High Cost} & \text{if } v_{i,j} = v_{k,j} = \text{Not Covered}, \\
\text{No Cost} & \text{if } v_{i,j} \neq v_{k,j}.
\end{cases}
\]

Each of the costs in the cost function have a value based upon the current environment. **Low Cost** is equal to the number of UAVs constrained with the target (e.g., if five UAVs are constrained with target \( t_j \), \( \text{Low Cost} = 5 \)). In this situation, multiple UAVs are monitoring the target, but ideally there should only be one UAV associated per target. **High Cost** is equal to twice the number of UAVs in the scenario (e.g., if their are 10 UAVs in the scenario, \( \text{High Cost} = 20 \)). Lastly, **No Cost** is always equal to zero, as it is the ideal situation when only one UAV is monitoring a target.
3.4 Distributed Stochastic Search

The distributed stochastic search (DSA) family of algorithms, specifically DSA-B variant, is the DisCOP algorithm used in the UAV experiments. As shown by Zhang et al. [23], the DSA-B algorithm has the best balance of solution quality, communication cost, and any-time properties compared to the other variants in the DSA family. The DSA-B algorithm first selects random values for variables in its context. A context is an assignment of values to variables for a DisCOP, which essentially is a (possibly partial) solution to a problem. Next, it enters a loop which continues until a solution is requested or a terminating condition is met. In the experimentation, the terminating condition was met when an agent did not update its variable after a period of time elapsed. In the loop, variables that were updated with a new value get sent as messages to its neighbors. Then the agent collects the messages received from its neighbors containing their variable assignments. After collection, the agent stochastically decides whether or not to update its value based upon whether there is a conflict between its currently assigned value and the value it received from its neighbors.

There are two reasons for choosing DSA-B over other complete algorithms: computational and memory cost, and fault tolerance [16, 18, 5, 13]. Because of the real-time computing restraints of this scenario and the limited hardware that would be available on an actual UAV, minimizing the computational and memory cost is critical. At any time DSA-B can provide its current context solution. During experimentation, a solution was requested on average every thirty seconds. If a complete algorithm were used, this would probably not be enough time to produce the optimal solution. DSA-B is also fault tolerant in the sense that an agent can drop out at any time, and the algorithm will still terminate and generate solutions.

3.5 Accelerated A*

In current research, there is a lack of three-dimensional plus time path planning algorithms. The best existing solution is an A* based algorithm, Accelerated A* (AA*) [21], which is the multiagent path planning algorithm used for the UAV scenario. Due to its superb computational efficiency, AA* is used in decentralized manner. Each UAV performs individual planning that provides a smooth trajectory through the waypoints. The interaction between the individual plans is achieved either at the time of plan execution or by means of multiagent flight simulation. In either case, the AA* planner hosted by the individual UAV is executed each time a collision is detected and the trajectory needs to be re-planned. As opposed to A*, AA* significantly reduces the number of samples (states) in a three-dimensional continuous space through adaptive sampling. The samples are generated by parameterized path construction elements. The sampling size varies based on the closest distance to the nearest operating space boundary. The configurations of states are not known prior to the search because they are given by the initial configuration and the algorithm execution. The number of states is higher (denser state space) around the obstacles and lower (sparser state...
space) further away from the obstacles. The adaptive parametrization defines the sampling length of straight elements and sampling angles for turn elements.

3.6 Interaction

The interaction between the different components in the approach is shown in Figure 3. First, the targets are passed to the DisCOP agent as variables. Using the DSA-B algorithm, the DisCOP agent determines its assigned targets. Before the targets are passed to AA*, they need to be ordered in the shortest path possible. One can generalize finding the shortest path between a set of targets problem as having a complete graph $G = (V, E)$, $V = \{v_1, v_2, \ldots, v_n\}$ that is isometrically embeddable in the Euclidean plane. In graph $G$ the goal is to find the Hamiltonian path where $v_1$ is a known vertex, this is also known as the Traveling Salesman Problem (TSP), which is NP-complete [2]. We use the 2-Opt exchange algorithm [19] to determine an approximate shortest path solution, which it does by removing the crossing edges in a graph. After the 2-Opt exchange algorithm creates an ordered sequence of targets, they are passed to the local UAV AA* multiagent path planner, which outputs a smooth path for the UAV to follow. This process can be altered if the AA* multiagent path planner reports that it cannot generate a flight path based upon the targets assigned. If this event occurs, process repeats.

![Fig. 3. State diagram of the local UAV agent view of the interaction between the DSA-B DisCOP agent, 2-Opt TSP approximation, and AA* multiagent path planner.](image)

4 Experiments

This section provides empirical results of this paper’s approach applied to the UAV surveillance scenario.
4.1 Related Work

Although there has been a significant amount of work in the area of UAV surveillance algorithms, most of this work relies upon a central decision maker and therefore is not fully distributed. For example, the Contract-Net Protocol [22] and other auction-based algorithms [12, 6] have been used to solve this problem. Although these solvers produce efficient results, they have no concept of privacy and allow one central authority to create plans for each UAV. Also, the Hungarian algorithm could be used to produce an optimal assignment if the problem reduced to a bipartite graph matching problem [17]. However, the bipartite graph matching problem assumes that a worker can only be assigned to one task, but in this problem a worker can be assigned to multiple tasks.

4.2 Experimental Parameters

As a comparison of our approach, we analyze its performance against two other distributed algorithms: distributed greedy set cover and random selection. The greedy set cover algorithm has been previously used over distributed nodes for a similar predator/prey problem [1]. In the algorithm, each UAV is assigned a unique identification number and sends out its current position and identification number to all other UAVs. Using this data, each UAV locally decides which targets it should monitor by selecting the lowest distance to each target that it was assigned to monitor. If a conflict exists where two or more UAVs are equal distance away from a target, then the UAV with the lowest identification number is assigned to that target. In the random selection algorithm, each UAV randomly selects whether it will monitor each target within its target group. For both of these algorithms, after a UAV completes its assigned targets, the algorithm is run again. This process continues until the scenario is terminated.

4.3 Experimental Setup

The AGENTFLY flight simulation testbed [20] was used for all the experimentation in this paper. Designed to test and compare collision avoidance algorithms in the domain of air-traffic control, AGENTFLY provides a three-dimensional space, support for limited communication, and real time 2D/3D visualization. Available air space can be restricted by terrain surface and no-flight zones.

To test the effectiveness of each algorithm, we mimic previous research in the area of measuring UAV surveillance techniques to define mathematically distinct target placements [7]. Three target arrangements were used: uniform, two-cluster, and random. In the uniform arrangement, targets are evenly placed throughout the environment. In the two-cluster arrangement, targets are paired and placed evenly around the environment. In the random arrangement, targets were randomly assigned locations. Each UAV is assigned to group A or group B to determine which targets they should monitor. For each of these arrangements, four targets were assigned to Group A Exclusive, four targets assigned to Group B Exclusive, and four targets assigned to Shared. These three target assignments are shown in Figure 4.
4.4 Metric Selection

To derive goal-based metrics for the scenario, we use the Goal Question Metric (GQM) approach by Basili et al. [3]. Metric selection using the GQM approach begins extracting the goals or high-level objectives from the specifications of the system. In the GQM approach, there are four parts to a goal: purpose, issue, object, and viewpoint. The purpose describes how the goal is effected. The issue is measurement of the event(s). The object is the item that the goal is centered around. The viewpoint is the perspective of the entity that is affected by the goal. An example given in [3] is to "Improve (purpose) the timeliness of (issue) change request processing (object) from the managers viewpoint (viewpoint)."

Next, the evaluator identifies questions—usually with quantifiable answers—that when answered, decide whether or not the system meets the goal. Each goal may necessitate multiple questions. Finally, the metric is a set of data associated with the questions that can be subjective (depending on the point of view, such as ease of use of a user interface) or objective (which is independent of the point of view, such as program size).

Although one could identify many goals for this scenario, the most important goal is to minimize the number of rounds that targets remain unwatched. We apply the Goal Question Metric approach as follows:

**Goal:** Minimize the number of rounds (purpose) between surveillance (issue) of targets (object) from the view of each UAV (viewpoint).

− **Question 1:** What is the number of rounds between a UAV monitoring each target?
  - **Metric 1.1:** Measure the average gap in rounds between UAVs monitoring each target.
- **Question 2:** How well are targets globally assigned after each iteration of the algorithm?
  
  - **Metric 2.1:** Measure the average number of overlapping targets for each plan instance.
  
  - **Metric 2.2:** Measure the average number of excluded targets for each plan instance.

### 4.5 Analysis

The results for each algorithm and scenario are shown in Table 1 and Table 2. This paper’s algorithm is referred to as DSA-B hereafter. Comparing only the average gap duration of unwatched target rounds, the distributed greedy set cover algorithm performs the best. However, on average there is only a 19% difference between it and DSA-B for all target types. The random algorithm performs comparably well on average, but also has a very high variance as one would expect. Comparing the results for overlapping and excluded assignments shows a true distinction between each algorithm and helps explain the results of Table 1. For each scenario, DSA-B produced plans on average with 38% fewer overlapping target assignments than distributed greedy set cover. This is important because overlapping target assignments mean that AA* must deconflict the airspace above the target for each UAV. Therefore these results show that selecting this paper’s approach does in fact reduce the strain on the AA* multiagent path planner. However, the number of excluded targets explains why DSA-B does not dominate the performance of average gap duration of unwatched target rounds. Compared to the distributed greedy set cover and sometimes random algorithms, DSA-B excludes more targets on average. It is only logical to assume that the more plan instances which exclude targets will lead to an increased number of unwatched target rounds. The random algorithm again performs well, but its variance does not make it an attractive solution. Overall, the DSA-B algorithm has similar results to distributed greedy set cover of average gap duration of unwatched targets and consistently produces plans with few overlapping targets and therefore produces the best solution for this problem.

<table>
<thead>
<tr>
<th>Uniform</th>
<th>Two-Cluster</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Exclusive</td>
<td>Shared</td>
</tr>
<tr>
<td>DSA-B</td>
<td>3.8 ± 1.1</td>
<td>1.9 ± 0.2</td>
</tr>
<tr>
<td>Greedy</td>
<td>2.7 ± 0.6</td>
<td>1.7 ± 0.2</td>
</tr>
<tr>
<td>Random</td>
<td>4.2 ± 1.4</td>
<td>2.0 ± 0.5</td>
</tr>
</tbody>
</table>

**Table 1.** Average gap duration of unwatched target rounds in minutes.

### 5 Conclusions and Future Work

This paper introduced a method of using DisCOP and distributed multiagent path planning to solve the distributed scheduling problem in continuous environments. We have formalized this method and applied it to the scenario of
Table 2. Average number of targets with overlapping and excluded assignments.

<table>
<thead>
<tr>
<th>Method</th>
<th>Uniform Overlap</th>
<th>Uniform Excluded</th>
<th>Two-Cluster Overlap</th>
<th>Two-Cluster Excluded</th>
<th>Random Overlap</th>
<th>Random Excluded</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSA-B</td>
<td>4.03 ± 1.56</td>
<td>2.13 ± 1.31</td>
<td>4.82 ± 1.86</td>
<td>4.82 ± 1.86</td>
<td>2.13 ± 1.04</td>
<td>5.87 ± 0.92</td>
</tr>
<tr>
<td>Greedy</td>
<td>6.87 ± 3.99</td>
<td>0</td>
<td>8.16 ± 3.55</td>
<td>0</td>
<td>8.46 ± 4.77</td>
<td>0</td>
</tr>
<tr>
<td>Random</td>
<td>4.82 ± 1.39</td>
<td>1.88 ± 0.88</td>
<td>4.94 ± 1.18</td>
<td>2.50 ± 1.34</td>
<td>5.62 ± 0.66</td>
<td>1.39 ± 0.87</td>
</tr>
</tbody>
</table>

distributed scheduling for UAV surveillance. We have shown that this method does in fact reduce the amount of plan deconfliction by initially forming the problem as a DisCOP which restricts the plan space to only a set of feasible plans.

Future work will consist of experimenting with different DisCOP algorithms and altering the DisCOP constraints. An extension of DSA which remembers the best global solution explored [24] or an extension of the ADOPT algorithm altered to handle resource constraints [14] may produce solutions with less target exclusion which is ideal for this DisCOP application. The constraints of the DisCOP could be altered to also include the distance from the UAV’s current position to its targets. This alteration would favor closer targets which may decrease the flight path distance, thereby decreasing plan execution time and the time between target surveillance.

References


AdCSP — Cooperative Ad Auctions with Privacy Guarantees

Roussi Roussev and Marius Silaghi
Florida Institute of Technology

Abstract. This paper discusses a cooperative model for advertisement bidding with privacy guarantees. In our application, a set of participants need to agree on an allocation of advertisements to be shown on an Internet stream. The participants have preferences that pose constraints to what should be streamed, and define the value of the ads. Streaming an ad to a specific audience yields larger value than streaming it to a random audience. We model the problem using distributed constraint optimization problems (DCOPs), with privacy. Each user has a set of shared variables, and private constraints on those variables. In this context, a local CSP describes a set of interests and constraints on those interests; all participants have stake in the ads being useful, not simply ignored.

1 Introduction

Historically, advertisement providers have used every known trick to get the Internet users’ attention: flashing ads, pop-ups, moving windows, multimedia, hogging browser resources. In the most extreme cases, they have resorted to intrusive techniques that go as far as exploiting a vulnerability and installing malware to collect data, or circumventing browser protection (such as pop-up blocking) to show their ads [9]. We address a type of advertising problems, for adds delivered using distributed games. These problems have the properties that (a) participants are motivated to cooperate; (b) they are concerned about privacy; and (c) the constraint defining preferences are naturally distributed.

A distributed constraint optimization problem (DCOP) is the perfect model for such problems. A general DCOP formalism uses: X – the set of variables, D – the domain of those variables, C – the set of constraints, A – the set of agents, I – inputs from those agents defining the constraints and O – the respective outputs for each agent as a function of the solution. Constraints can be either public or private to some of these agents. When agents input the constraint directly, and are informed of the full solution, then the I and O elements are optional, being the identity functions. The case of a single variable is practically a voting problem that can be modeled with DCOPs. More variables require the full DCOP power.

The solution of the DCOP provides incentive for both the clients and the ad providers: clients want their ads to be useful to them, ad providers want their ads to be useful to the client and not simply ignored. If constraints cannot be satisfied, the ad has lower value to the participants - proportional to the number unsatisfied participants and constraints.
The outline of this paper is as follows: Section 2 provides a background. In Section 3, we discuss the bidding process. Section 4 provides discussion and future work and Section 5 concludes.

2 Background

Distributed Constraint Optimization [4] is a general framework to model problems where a group of participants want to find an agreement on an assignment of values to a set of variables, such that the sum of a set of functions (aka weighted constraints) on those variables is maximized. Some of the weighted constraints are private to various participants.

Several versions of this framework are available, with different capabilities for modeling privacy. Among the versions that allow for fine specification of privacy requirements are defined in [3, 6, 2, 1]. We next provide the version in [6], which is used in one the algorithms proposed later and is general enough to encompass the other cases that we study.

**Definition 1 (DCOP).** A Distributed Constraint Optimization Problem (DCOP) is defined by six sets \( (A, X, D, C, I, O) \) and an algebraic structure \( F \). \( A=\{A_1,...,A_n\} \) is a set of agents, \( X \) is a set of variables, and \( D \) is a set of domains (one for each variable). \( I=\{I_1,...,I_n\} \) is a set of secret inputs. \( I_i \) is a tuple of \( \alpha_i \) secret inputs (defined on \( F \)) from the agent \( A_i \). Each input \( I_i \) belongs to \( F^{\alpha_i} \). \( C \) is a set of weighted constraints. There may exist a public weighted constraint in \( C \), \( \phi_0 \), defined by a function \( \phi_0(\epsilon) \) on tuples of assignments \( \epsilon \), known to everybody. However, each constraint \( \phi_i, i>0 \), in \( C \) is defined as a set of known functions \( \phi_i(\epsilon, I) \) over the secret inputs \( I \), and the tuples \( \epsilon \) of assignments to all the variables in a set of variables \( X_i, X_i \subseteq X \). \( O=\{o_1,...,o_n\} \) is the set of outputs to the different agents. Let \( m \) be the number of variables, \( o_i : D_1 \times ... \times D_m \rightarrow F^{\omega_i} \) is a function receiving as parameter a solution and returning \( \omega_i \) secret outputs (from \( F \)) that will be revealed only to the agent \( A_i \).

A wide range of solvers are available for problems modeled as DCOPs. Some of them are based on constructive distributed search, such as ADOPT [4] and BnB-Adopt [12]. Others are based on dynamic programming, such as DPOP [5]. These algorithms define various trade-offs between efficiency and privacy guarantees. Another family of algorithms strive to achieve maximal privacy at any cost. Among them we mention the Branch&Bound MPC-DisWCSP [8, 6, 7]. These later algorithms can handle the full privacy specifications of the version of the DCOP framework given above, and are used for the experiments described in this paper.

3 DCOP model

We address the application where a set of \( n \) users employ an application (such as a networked game) and the provider of the game streams advertisements to a set of \( m \) positions in the GUI of the game. The video ads may involve significant bandwidth and therefore we assume that the provider wants to maximize the
utilization of his network by multicasting the ads such that all users see the same
ads in the same position.

We address first the case of several users negotiating for a stream of advertisements coming from one provider. The provider may sell/resell adds and tries to stream them in such a way as to maximize their impact. Remarkably, the users share the same purpose as the provider, preferring ads that will be useful to themselves. For simplicity, we will first assume that to achieve their purpose, the users act truthfully in declaring their true preferences on ads, whenever needed. However, if possible, they prefer to maintain the privacy of their preferences. The obtained problem fits very well into the mold of a standard DCOP framework, where all the participants collaborate to maximize the sum of the weights of their constraints. It should be noted that when the ad stream is shared among multiple agents, the number of constraints that can be satisfied can be lower.

We model the problem as a DCOP with a set $A$ of $n$ agents (one per user). The provider acts as a broker, that announces a description of the set $R$ of $d$ ads competing for the $m$ slots (positions in the GUI). Those ads are prescreened to match the context of the stream to be viewed. The slots are modeled by $m$ variables in the set $X = x_1, \ldots, x_m$. Each variable can take values from the set $D = [1..d]$, where each value corresponds to one of the ad descriptions. Each agent holds a set of private weighted constraints on the possible values of subsets of the variables. The accumulated weight given to an assignment by all the constraints of an agent specifies the value/interest that the user would have in viewing the corresponding combination. Such constraints can be specified by the user, or can be computed from the profile of the user and from the description of the ads (generating a high weight if there is a match and a small weight if there is no match between the profile and the description). There also exists a globally know public constraint, specifying that the $n$ ads should be all different.

The DCOP problem can be solved using any existing solver, based on constructive search or on secure cryptographic protocols. While maximal privacy is achieved by secure multi-party protocols such as MPCDisWCSP4, among the most efficient distributed solvers we mention DPOP. Since DPOP and MPCDisWCSP4 are deterministic protocols, their efficiency can be predicted without experimentation. As such, the efficiency of DPOP for this problem is $2^ndm$ local additions, $2n$ message latencies for messages of size $d^m$ weights. Since one can expect that $m$, the number of ads simultaneously displayed by the GUI, is small, the computation of DPOP will be reasonable. The efficiency of a secure multi-party computation is experimentally detailed in the experiments section. Algorithms such as ADOPT could be employed to offer a trade-off between the efficiency of DPOP and the security of MPC-DisCSP4 and they should constitute the subject of further work.

In the second case, the provider bids for a single client. The ad stream is private to the client. The ad provider distributes a set of descriptors, one for each ad that may be streamed and displayed in the users’ GUI. Each descriptor consist of a set of possible users profiles, and a score. The score estimates how much a user with such a profile will value this ad. A user profile in an ad descriptor can contain regular expressions or a Turing machine that will match a set of possible description. The agent of each user compares the private user profile with the ad descriptors and establishes the value to be associated by the user to each
possible configuration of ad displays. The computation occurs in a sandbox [11] and the result of the computation is thrown away.

Below is an example of clients interests and their constraints. The ad provider bids against those clients on the shared stream. The common client interests are A and D. We have performed initial end-to-end experiments with ad optimization for 4 agents. While our initial implementation took 20 seconds, we are confident that improvements can be made.

<table>
<thead>
<tr>
<th>Client 1 (Age = 21):</th>
<th>Ad1 constraints:</th>
</tr>
</thead>
<tbody>
<tr>
<td>interests[a] = 1</td>
<td>20 ≤ Age ≤ 25</td>
</tr>
<tr>
<td>interests[b] = 4</td>
<td>interests[a] = 1</td>
</tr>
<tr>
<td>interests[d] = 9</td>
<td>interests[d] = 7</td>
</tr>
<tr>
<td>occupation := student</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Client 2 (Age = 25):</th>
<th>Ad2 constrains:</th>
</tr>
</thead>
<tbody>
<tr>
<td>occupation := engineer</td>
<td></td>
</tr>
<tr>
<td>interests[a] = 3</td>
<td>35 ≤ Age ≤ 40</td>
</tr>
<tr>
<td>interests[d] = 4</td>
<td>occupation := engineer</td>
</tr>
<tr>
<td>interests[f] = 6</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 1. An example of two clients, ad providers and their preferences.

4 Discussion and future work

In the single client model, we ignore cases of multiple conflicting constraints within the same client (for example, a shared family browser), but those limitations can partially be addressed through multiple browser profiles. Temporal constraints are also not discussed. In the multi-client scenario, the worst case is all conflicting interests- therefore, no optimum allocation can be made.

To minimize latency, the ad computations should be done asynchronously so as not to obstruct with the viewing experience. The display of the actual content should be started before the ads are negotiated. Therefore the ads are negotiated and streamed in the background.

Given the malicious history of some advertisers [10], some may still choose to abuse the AdCSP system. In addition, malicious clients can still ignore or skew the bidding process with incorrect preferences.

5 Conclusions

We introduce and evaluate a new application of Distributed Constraint Optimization Problems (DCOPs), namely for modeling and solving the problem of agreeing on the content of a stream of advertisements. In this problem, a set of users of a networked application have agents using their profile to find the preferred combination of ads out of a set announced by the provider(s) of the application. This has to be done while keeping the profiles private to the users. We discuss two alternative ways for modeling the problem. In one alternative,
where the variables stand for slots of ads, the solution of the problem specifies directly the solution. In the second alternative the variables model profile items of the users, and the (secure multi-party) solver finds the solution that satisfies the constraints.

An experiment with the second approach shows that the solution can be found with guaranteed privacy within 20 seconds of distributed computation, using our current implementation of MPC-DisWCSP4 (which is currently based on Shamir secret shares, but could use any other multiparty computation technique, such as homomorphic encryptions for other efficiency/security trade-offs). If the ads are made of long movies, the first 20 seconds of the ads may not be optimized, but any subsequent ads will be obtained as the result of the optimization process, improving both the experience of the users and the efficiency of the advertisement process.

We have shown that DCOPs can be successfully used to improve user experience and ad efficiency, and opened a new problem for the future DCOP research.

References

SymDPOP: Adapting DPOP to exploit partial symmetries

Xavier Olive and Hiroshi Nakashima
Graduate School of Informatics, Kyoto University

Abstract. This work proposes a new approach to dealing with symmetries in Distributed Constraint Optimisation Problems by adapting the DPOP algorithm [1]. In contrast to an already proposed distributed preprocessing method leading to a problem redefinition [2], we present here a method that exploits the structure of DFS trees, with no explicit redefinition of the problem.

We exhibit the flexibility in the symmetry detection that this algorithm offers, then compare its performance with DPOP and the aforementioned preprocessing method. We demonstrate that SymDPOP significantly cuts down the total volume of communication spent on symmetric and partially symmetric problems, the latter being problems containing symmetric subproblems. We also stress the fact that SymDPOP minimises the overhead of symmetry detection by keeping the total volume of communication below that of DPOP.

Keywords. distributed constraint optimisation, symmetry breaking

1 Introduction

Distributed Constraint Optimisation is an efficient framework suitable for modelling naturally distributed problems. Those consist in different agents cooperating to solve an optimisation problem. Only a subset of agents knows about each constraint, which leads to the idea of natural distribution. Keeping the definition of a problem distributed may be relevant when computational and communication power is limited, or when privacy concerns are raised.

After Asynchronous Backtracking by Yokoo [3], several algorithms have been proposed to solve Distributed Constraint Satisfaction Problems (DCSP), including ADOPT [4] or DPOP [1]. As symmetry processing can be successful in traditional constraint programming [5, 6], an attempt [2] has been made to exploit them in a distributed context as a preprocessing step to any solving technique. Yet, this procedure requires that all the agents agree on how to reformulate the symmetric problem into a smaller one.

We present here SymDPOP, a new method for exploiting symmetries which is specific to DPOP algorithm and does not require any preprocessing step. We show that by adding extra information to the messages used by DPOP, we can reduce their number, their volume, as well as the workload of each agent.
Fig. 1. A constraint graph and its DFS tree

We also show that unlike the preprocessed DPOP presented in [2], SymDPOP cuts down on the total volume (rather than number) of messages sent. Moreover, the work and communication saved in the symmetry detection process are not lost even if the detection process cannot find any symmetry.

2 Preliminaries

Definition 1 (DCSP). A distributed constraint satisfaction problem (DCSP) is a constraint satisfaction problem (CSP) where variables are distributed over different agents. It consists of a finite set of variables $x_1 \cdots x_n$, a set of domains $d_1 \cdots d_n$, a set of agents $a_1 \cdots a_n$ not necessarily all different, and a set of constraints $c_1 \cdots c_t$.

Definition 2 (Neighbourhood). Each constraint has a scope of variables, thus a scope of agents. Two agents are neighbours if they share at least a constraint.

Constraints fall into two categories: local (private) and global (distributed) constraints. Each agent owns a local subproblem, which is a partial view of the global problem. This global problem is the union of all the local subproblems.

2.1 DFS trees

Whereas the first algorithms designed to solve DCSP kept increasing the size of the neighbourhoods of their variables, a more reasonable approach used in DPOP or ADOPT groups variables according to the constraints they share.

Definition 3 (DFS tree). A DFS tree is a rooted and directed spanning pseudo-tree of the constraint graph such that any two neighbours in the original graph are both in the same branch.

From a constraint graph, with variables as nodes and constraint as edges, as shown in Figure 1, we build a DFS tree. The edges of the DFS tree only represent some of the constraints. Thus, a back edge is an edge present in the constraint graph as well as in the DFS pseudo-tree but not in the tree part of the DFS tree. An agent connected to an ancestor via a back edge is called a pseudo-child and the ancestor is called a pseudo-parent. In the figure, the dashed line is a back edge, and 1 and 3 are resp. pseudo-parent and pseudo-child of each other.
Definition 4 (Separator). Each variable $x$ has a separator $sep(x)$, defined as the set of its ancestors connected with an edge or a back edge to $x$ or to any descendent of $x$.

Definition 5 (Induced width). The induced width of a graph is the size of the largest separator of the nodes in the generated DFS tree.

In other words, the separator of a variable $x$ is the set of ancestor variables of $x$ constrained with $x$ or with any descendant of $x$. DPOP bounds the number of sent messages. However its weakness lies in the size of those messages which grows exponentially with the induced width of the DFS tree.

In order to generate a DFS tree in a distribute manner [7], all the agents label their variables as non-visited. Then, using a variable election algorithm, one of the agents is designated as root. Choosing the variable with the biggest neighbourhood as root favours well-balanced DFS trees, as the most linked variables are more likely to be around the centre of the constraint graph.

The root then initiates the propagation of a token, which will visit all the variables of the graph. It starts sending it to the first neighbour and waits for it to come back, before sending it to the next neighbour. When a variable receives a token, it marks the sender as parent. When it sends a token, it marks the destination variable as child. The token can return either from the variable to whom the token was sent, or from another neighbour, in which case this neighbour is marked as pseudo-child. Again, choosing to send the token to the neighbour known to be the most connected is a heuristic likely to create a tree with a lower induced width, hence offering better performance for DPOP.

2.2 DPOP

Two categories of algorithms using DFS trees stand out: ADOPT is an asynchronous search algorithm using DFS trees to perform a top-down search procedure; DPOP uses a different approach: it aggregates solutions from bottom to top of DFS trees and does not search.

DPOP is a distributed version of the bucket elimination algorithm [8]. It has three phases: a DFS tree creation, a $\text{UTIL}$ propagation and a $\text{VALUE}$ propagation. At the end of the tree creation, all the variables consistently label each other as parent/child and pseudo-parent/pseudo-child. This serves as a communication structure for the two other phases.

Definition 6 (UTIL message). The $\text{UTIL}$ message sent by agent $i$ to agent $j$ is a multidimensional matrix, with a dimension per variable in the separator of $i$. The matrix is filled with the costs of each assignation.

The $\text{UTIL}$ propagation phase starts from the leaves and goes up to the root. Each node aggregates and optimises constraints, namely joins and projects $\text{UTIL}$ messages coming from its children and sends to its parent a representation of relations with its ancestors via a new $\text{UTIL}$ message.
Definition 7 (Junction). Let \( f_1 \cdots f_k \) be functions defined over \( d_1 \cdots d_k \), the junction \( \sum f_i \) is defined over \( u = \bigcup d_i \), such that for each variable \( x \) in \( u \), \((\sum f_i)(x) = \sum (f_i(x))\).

The junction is an aggregation operation which a node applies on the resulting constraints coming from its different subtrees. If node variable \( x \) has \( k \) children \( y_1, \cdots y_k \), it receives from each of them a UTIL message with cost functions \( f_1, \cdots f_k \) defined on \( \text{sep}(y_1), \cdots \text{sep}(y_k) \). Variable \( x \) then sums up all these constraints in \((f_1 + \cdots + f_k)\) defined on \( \text{sep}(y_1) \cup \cdots \cup \text{sep}(y_k) = \text{sep}(x) \cup \{x\} \).

Definition 8 (Projection). Let \( f \) be a function of a set of variables \( s = \{x, y_1, \cdots, y_n\} \). We define the projection of \( f \) on \( x \) as the function \( \tilde{f}_x \) of \( s - \{x\} \) such that for every assignation \( \{y_1 = u_1, \cdots, y_n = u_n\} \) of \( s - \{x\} \), \( \tilde{f}_x(u_1, \cdots, u_n) = \min_x f(x, u_1, \cdots, u_n) \).

The projection is an optimisation operation where the current node picks its optimal value for each possible assignation of the variables in its separator. With the same example, \( x \) will minimise the function \((f_1 + \cdots + f_k)\) and create \( \tilde{f}_x \) defined on \( \text{sep}(x) \), to be sent to its parent. The optimal value of the tree root will thus be the result of the join/project process operated by the root node. From this optimal value, all children can choose their optimum according to the value given to each variable in their separator and to the projection they operated.

This VALUE propagation process is initiated by the root of the tree. Each node determines its optimal value based on the values attributed to its separator variables and propagates this value to its children with a VALUE message.

3 Symmetries

Symmetries are omnipresent in nature, thus widely used in physics or engineering, making problems easier hence faster to solve. As well, we can profit from symmetries in constraint programming [6, 9] in order to avoid revisiting equivalent assignations. As the induced width of DFS trees directly affects DPOP’s performance, deriving implied constraints [5] will not make the resolution faster. Therefore, we assume the problem reformulation approach is the only acceptable way to deal with symmetries: [2] reformulates the problem after a preprocessing step (and breaks symmetries), but we propose here a way to reformulate the problem as we solve it (and exploit symmetries without breaking them).

3.1 Definitions

Definition 9 (Symmetry). A variable symmetry over a CSP is a mapping that permutes the variables of the problem by pair, while leaving the constraints unchanged.

For example, if \( (x_1, y_1, x_2, y_2, z) \) are all defined on \( \{0, 1\} \) and constrained by \( x_1 + 2y_1 = z \), \( \sigma = \{(x_1 \leftrightarrow x_2), (y_1 \leftarrow y_2)\} \) is a symmetry: indeed, as \( \sigma(x_1 + 2y_1 =
\( z \} = \{ x_2 + 2y_2 = z \} \) and \( \sigma(\{ x_2 + 2y_2 = z \}) = \{ x_1 + 2y_1 = z \} \), \( \sigma \) leaves the set of constraints globally unchanged. Therefore, we can reduce the search effort by solving only \( x_1 + 2y_1 = z \) and applying the symmetry to the solution in order to find the solution of the whole original problem.

The detection of symmetries over centralised CSPs has been studied in [10] with the help of group theory. In a distributed context, all the agents only have a partial view over the problem, and are unable to find symmetries by themselves. Henceforth, the distribution of the definition requires a different approach.

**Definition 10 (Partial representation).** A partial representation of a problem \( p \) is the restriction of \( p \) to a subset of variables \( \mathcal{V} \), their neighbour variables, and the constraints involving any variable in \( \mathcal{V} \). Each agent naturally owns a partial representation of the problem restricted to its variables.

**Definition 11 (Partial symmetry).** A partial symmetry over a CSP is a symmetry over a partial representation of the CSP. The symmetries detected by an agent which has a partial view over the problem are partial symmetries.

**Proposition 1.** If \( \sigma \) is a partial symmetry over \( p_1 \) and \( p_2 \), two partial representations of the same CSP, then \( \sigma \) is a partial symmetry over \( p_1 \cup p_2 \).

**Proof.** Let \( c \) be a constraint of \( p_1 \cup p_2 \). If \( c \) is a constraint of \( p_1 \), then \( \sigma(c) \in p_1 \subset p_1 \cup p_2 \). As well, if \( c \) is a constraint of \( p_2 \), then \( \sigma(c) \in p_2 \subset p_1 \cup p_2 \). Therefore, \( \sigma \) is a symmetry over the union of the problems.

**Corollary 1.** If \( \sigma \) is a partial symmetry for each agent involved in a distributed CSP, then it is a symmetry for the global problem.

**Theorem 1.** If \( \sigma \) is a partial symmetry for all agents owning a variable in \( \sigma \) and for all their neighbouring agents, \( \sigma \) is a symmetry of the whole problem.

**Proof.** Let \( \sigma \) be a partial symmetry for an agent \( a \). Let \( b \neq a \) be an agent of the DCSP. If no variable of \( b \) is subject to a permutation in \( \sigma \), then the problem definition according to \( b \) is unchanged through \( \sigma \). If not, \( \sigma \) is a partial symmetry of \( b \) by hypothesis. Thus, according to corollary 1, \( \sigma \) is a symmetry.

Computing in a distributed way the intersection of all the partial symmetries of a given problem as a preprocessing method is a basic method to exploit the symmetries [2]. Hereunder, we propose an algorithm to compute this intersection simultaneously with the UTIL propagation process of the DPOP algorithm.

### 3.2 Symmetries in DFS trees

We present in this paper SymDPOP, an algorithm inspired from DPOP, which is adapted for detecting symmetries in the DFS tree structures and not in the constraint graph. A symmetry in the DFS tree is a symmetry of the constraint graph, but the reverse is not necessarily true.
DPOP appears to be particularly efficient when, during the tree generation, the token is passed in priority to the neighbour which has the largest neighbourhood. In case of equal number of neighbours, the choice among them can be done arbitrarily by, for example, the alphabetical order of the variable names.

However, as symmetry detection focuses on permutations of variables owned by the same agent, we introduce a simple heuristic to choose the next variable in a total order of the agent owning it, e.g., the alphabetical order of agent names, before the final tie-breaking with variable names. This SymMostConnected heuristic is more likely to keep the symmetry of the constraint graph in the DFS tree, as shown on figure 2 where colours represent agents owning the variables.

### 3.3 SymDPOP

Detecting symmetries in a distributed context is based on the idea of theorem 1. Each agent is able to detect its partial symmetries and if those are also partial symmetries for the neighbours of agents owning variables involved in those symmetries, they are symmetries of the whole problem. While the preprocessing method used in [2] communicates at the agent level, SymDPOP uses the DFS structure produced by DPOP algorithm and involves communication between variables, drawing comparisons between variables owned by the same agent.

Like DPOP, the symmetry detection process starts from the leaves, builds and propagates partial symmetries permuting subtrees, until it reaches the top of the tree. However, each node $n$ only keeps a representation of this partial symmetry $\sigma$ restricted to $\{n\} \cup sep(n)$, and propagates the representation of $\sigma$ restricted to $sep(n)$ only.

**Definition 12 (SymUTIL message).** A SymUTIL message is a UTIL message to which we attach a set $\mathcal{S}$ of partial symmetries defined as images of each variable in the separator of the UTIL message. For each symmetry $\sigma_i$ of the set $\mathcal{S}$, and for each variable $x$ of the separator, if $x$ is owned by agent $a$, then $\sigma_i(x)$ is also owned by the same agent $a$.

**Proposition 2.** It is possible to rebuild all the symmetric UTIL messages from a SymUTIL message.
Algorithm 1 Once agents know about their DFS tree

```plaintext
for all agent owning leaf variables do
    consider partial symmetries involving those leaves;
group leaves subject to partial symmetries;
for all source leaf do
    build SymUTIL and start propagation;
end for
for all leaf not involved in any symmetry do
    start UTIL propagation;
end for
end for
```

**Fig. 3.** Consistency

After generating the DFS tree, each agent having leaf variables examines them with their constraints to find those subject to a partial symmetry (algorithm 1). Among those symmetric leaves, it labels one as source leaf, and builds a set of partial symmetries $\sigma_1 \cdots \sigma_n$ associating each source leaf to its symmetric leaves, and the separator of the source leaf to its symmetric separators. The source leaves then build a SymUTIL message and sends it to their parent.

**Definition 13 (Consistency).** Let $\sigma$ and $\tau$ be two partial symmetries of two subproblems $p$ and $q$. We say that $\sigma$ and $\tau$ are consistent if for each variable $x$ in $D(p) \cap D(q)$, $\sigma(x) = \tau(x)$.

**Proposition 3.** Let $\sigma$ and $\tau$ be two partial symmetries on two subproblems $p$ and $q$. If $\sigma$ is consistent with $\tau$, there exists a partial symmetry on $p \cup q$.

This proposition will determine whether and how to propagate the symmetry detection. Intuitively, referring to Figure 3, both nodes $x_1$ and $x_2$ are roots of a subtree, for which there exists a partial symmetry $\sigma$ (resp. $\tau$) associating the subtree to another subtree. If they match on their intersection, which is reduced to $\{y\}$ on the figure, and if we can also swap the other neighbours (here the parents) of $y$ with $\sigma(y) = \tau(y)$, then we can build a symmetry $\omega$ associating the
Algorithm 2 On reception of all UTIL/SymUTIL messages

if all the received messages are UTIL then
    continue UTIL propagation;
else
    if the problem is still partially symmetric then
        continue SymUTIL propagation;
    else
        regenerate();
    end if
end if

Algorithm 3 regenerate()

for all SymUTIL received do
    rebuild the UTIL messages from the SymUTIL;
    for all rebuilt UTIL messages do
        continue UTIL propagation;
    end for
end for

subtree rooted in $y$ to the subtree rooted in $\omega(y) = \sigma(y) = \tau(y)$. We formalise this idea and extend it to the case of multiple symmetries in the following corollary.

Corollary 2 (SymUTIL propagation). Let $(x_1, \ldots, x_n)$ be sibling variables of the same DFS tree, and $S_i = (\sigma_{i1}, \ldots, \sigma_{ik})$ be the set of partial symmetries in the SymUTIL message sent by $x_i$ to $y$, parent variable of all the $(x_i)$. Let $\upsilon$ be a partial symmetry on the constraints involving $y$ (and $\upsilon(y)$). If by picking one $\sigma_i$ in each $S_i$, all those selected $\sigma_i$ are consistent together and with $\upsilon$, then there exists a partial symmetry $\omega$ on the subproblem represented by $y$-rooted subtree.

Proof. Since $\upsilon$ and all $\sigma_i$ are consistent, there exist a partial symmetry $\omega$ on the union of their associated subproblems. $\omega$ involves variables which are either descendents of $y$ or in the separator of $y$. For each variable $z$ in $D(\omega)$, if $z$ is a variable or linked to a variable from one of the $x_i$-rooted subtrees, then $\omega(z) = \sigma_i(z)$. Otherwise, it is a neighbour of $y$ and thus in $D(\upsilon)$, in which case $\omega(z) = \upsilon(z)$.

At the reception of all the SymUTIL messages (algorithm 2), destination variable $y$ cross-checks the consistency of the partial symmetries of its children, and considers a partial symmetry on its separator, building de facto a potential $\upsilon$ and $\omega$. If this $\omega$ is a partial symmetry, the agent computes only one junction/projection operation for $y$ and sends the appropriate SymUTIL message to its parent. If the partial symmetries are not consistent, the symmetry detection process stops (algorithm 3), regenerates as many UTIL messages as necessary, attributes them to each variable and lets them continue the regular UTIL propagation of DPOP algorithm.

When the UTIL/SymUTIL propagation is finished, each variable, starting from the top of the tree, gets attributed a value. If the top of the tree is in a symmetric
Algorithm 4 On reception of a SymVALUE message
assign current variable x and all the $\sigma_i(x)$.
continue SymVALUE propagation.

Algorithm 5 On reception of a VALUE message
if symmetric separators get the same assignation then
start a unique SymVALUE propagation;
else
continue VALUE propagation;
end if

state, it propagates the value down with SymVALUE messages, otherwise it uses VALUE messages. The content of VALUE and SymVALUE messages is the same; only the name differs.

Proposition 4. If variable $x$ sends a SymUTIL message, including $n$ symmetries $\sigma_i$, and if the values attributed to $\text{sep}(x)$ are equal to the ones attributed to $\text{sep}(\sigma_i(x))$, then the values attributed to each variable of $x$-rooted subtree are equal to the values attributed to the image of this subtree through $\sigma_i$.

Indeed, whether the whole problem is symmetric or not, if $x$ sends a SymUTIL message, it is the head of a subtree which is partially symmetric to the subtree rooted in $\sigma_i(x)$. If $y$ is a descendent of $x$, it will choose its value according to the value of its separator. The variables in $y$’s separator are either descendants of $x$ or part of $x$’s separator. Consequently, when a variable $x$ receives a SymVALUE message, it assigns the same value to the images of $x$ through all the partial symmetries $\sigma_i$ (algorithm 4).

When it receives a VALUE message (algorithm 5), it keeps propagating unless the variable previously sent a SymUTIL message. In that case, the agent starts only one SymVALUE propagation for each group of partially symmetric variables which has got the same separator assignation.

3.4 Example on a symmetric problem

We consider the problem between the following agents: $c$ (●) owns $c_0$, $x$ (●) owns $\{x_1, x_2\}$, $y$ (●) owns $\{y_1, y_2\}$, $z$ (●) owns $\{z_1, z_2\}$. All the variables are defined on $\{0, 1, 2\}$ and are subject to the following constraints: $c_0 \neq x_1, c_0 \neq x_2, x_1 \neq y_1, x_2 \neq y_2, z_1 \neq y_1, z_2 \neq y_2, x_1 \neq z_1, x_2 \neq z_2$.

With the preprocessing method, agent $x$ will detect a partial symmetry $\sigma = (x_1 \leftrightarrow x_2, y_1 \leftrightarrow y_2, z_1 \leftrightarrow z_2)$, and suggest it in order to agents $y$, $z$ and $c$, which will agree on the symmetry, and on a reformulation leading to the DFS tree on the left side of Figure 4.

In contrast, SymDPOP starts at the end of the generation of the DFS tree. Agent $z$ finds that $\sigma$ is a partial symmetry, and sends its constraint in a SymUTIL message (double line in the figure), together with $\sigma$, to its parent, variable $y_1$.
When agent $y$ gets the message, it considers the permutation of the parents of $y_1$ and $y_2$ ($x_1$ and $x_2$) which is a partial symmetry consistent with $\sigma$. Then, it computes only one junction and one projection operation on $y_1$ and sends the SymUTIL message to $x_1$. $x_1$ considers the permutation of the parents of $x_1$ and of $x_2$, which are here the same variable $c_0$: the problem is symmetric.

Agent $c$ receives the SymUTIL message and, having no parents, sends a SymVALUE message back. When agents receive a SymVALUE message, they attribute the value to their variable, and to the image of the variable through $\sigma$.

### 3.5 Example on a non-symmetric problem

We consider the following problem: variables $a$, $b$, $c$ and $d$ are all owned by a different agent. Agent $t$ (●) owns variables $t_1$ and $t_2$, agent $x$ (●) owns variables $x_1$ and $x_2$, agent $y$ (●) owns variables $y_1$ and $y_2$ and agent $z$ (●) owns variables $z_1$ and $z_2$. Those variables are all defined on $\{0, 1, 2\}$ and are subject to the following constraints: $\forall i, x_i \neq y_i, x_i \neq z_i, x_i \neq t_i$, and $t_1 \neq d, t_1 \neq b, a \neq b, a \neq c$ and $a \neq t_2$. We will refer to Figure 5 as we discuss further about this problem.

First of all, this problem is not globally symmetric: we cannot permute any pair of variables and leave the problem unchanged. However, if agents $x$, $y$ or $z$ consider their definition of the problem, they can find the following partial symmetries: $\sigma_x = (x_1 \rightleftharpoons x_2, y_1 \rightleftharpoons y_2, z_1 \rightleftharpoons z_2, t_1 \rightleftharpoons t_2)$ for agent $x$, $\sigma_y = (x_1 \rightleftharpoons x_2, y_1 \rightleftharpoons y_2)$ for agent $y$, and $\sigma_z = (x_1 \rightleftharpoons x_2, z_1 \rightleftharpoons z_2)$ for agent $z$. Using DPOP, the UTIL message that $y_1$ would send to $x_1$ and the one that $y_2$ would send to $x_2$ would be identical, and would both be sent between the same agents $x$ and $y$. Even though the problem is not symmetric, SymDPOP will be able to gather those messages into one SymUTIL message.

At the end of the generation of the DFS tree, variables $c$ and $d$ are not subject to partial symmetries, so they follow the regular DPOP procedure and send a UTIL message to their parent variable. Then, agent $y$ (resp. $z$) detects its partial symmetry $\sigma_y$ (resp. $\sigma_z$). As a consequence, variable $y_1$ (resp. $z_1$) sends a SymUTIL (double line in the figure) message to variable $x_1$, containing the constraint between $x_1$ and $y_1$ (resp. $z_1$) and the partial symmetry $\sigma_y$ (resp. $\sigma_z$).
When agent $x$ receives the messages, it considers swapping the parent of $x_1$ ($t_1$) and of $x_2$ ($t_2$) and builds a symmetry consistent with the one received from $y$ and from $z$. Therefore, it joins the constraints, projects $x_1$ out and sends the SymUTIL message together with $\sigma_x$ to variable $t_1$. When it receives the messages, it also receives an UTIL message constraining $d$ and $t_1 (\neq t_2)$, hence stopping the SymUTIL propagation. Variables $t_1$ and $t_2$ are then parents of the root of symmetric subtrees, and the SymUTIL message received from agent $x$ is transformed into two UTIL messages directed to $t_1$ and to $t_2$ (dashed line) who continue the regular UTIL propagation process from now on.

On the way down, when agent $t$ gets the two VALUE messages, it compares the value assigned to $t_1$ and $t_2$. If $t_1 = t_2$ then $t_1$ sends a SymVALUE to $x_1$, which will assign a value to $x_1$ and $x_2$ then sends another SymVALUE to agents $y$ and $z$. If $t_1 \neq t_2$, then $t_1$ and $t_2$ both send a VALUE message to agent $x$ which will compare the value attributed to $x_1$ and $x_2$ and consider as well the choice between a VALUE propagation and a SymVALUE propagation.

4 Evaluation

4.1 Examples

In this section, we measured the number and volume of messages sent during the process of solving the problems in the past sections. We implemented SymDPOP on FRODO framework [11] for our measurements. We measured the number and volume of messages exchanged for solving the symmetric problem of Figure 4 in Table 1. The biggest part of the total number of messages is used to generate a DFS tree, and both methods work on the same tree, so SymDPOP and DPOP are equal on that aspect. On the UTIL/VALUE propagation part, however, SymDPOP reformulates the problem dynamically, and cuts the number of messages by 2.
Table 1. Number (up) and volume (down, in bytes) of messages sent to solve problem of Fig. 4 (left) and problem of Fig. 5 (right)

<table>
<thead>
<tr>
<th></th>
<th>DPOP</th>
<th>SymDPOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS generation</td>
<td>2281</td>
<td>2281</td>
</tr>
<tr>
<td>UTIL/VALUE</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>total</td>
<td>2293</td>
<td>2287</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>DPOP</th>
<th>SymDPOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS generation</td>
<td>15607</td>
<td>15607</td>
</tr>
<tr>
<td>UTIL/VALUE</td>
<td>4278</td>
<td>2794</td>
</tr>
<tr>
<td>total</td>
<td>19885</td>
<td>18401</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>DPOP</th>
<th>SymDPOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS generation</td>
<td>24052</td>
<td>24052</td>
</tr>
<tr>
<td>UTIL/VALUE</td>
<td>7000</td>
<td>6746</td>
</tr>
<tr>
<td>total</td>
<td>31052</td>
<td>30846</td>
</tr>
</tbody>
</table>

Be that as it may, in a distributed context, the communication volume has a substantial impact on the performance. Even though lots of messages are sent for generating this DFS tree, those are rather small compared to the messages including a constraint. SymDPOP also cuts the volume of those constraint and value messages. To put the comparison with the DFS generation phase, SymDPOP cuts almost 10% of the total communication volume here.

Furthermore, one of the main concerns of symmetry treatment is the overhead of symmetry detection. One of the main advantages of SymDPOP is that it does not generate extra messages even if the problem is not symmetric. It uses the regular UTIL messages to detect the symmetry and the information added to make a SymUTIL messages (a list of variables) is smaller than a whole UTIL message (a multidimensional matrix of costs).

As we solved the non-symmetric problem from Figure 5, the values given to $t_1$ and $t_2$ were different, and only VALUE messages were sent. Table 1 shows the statistics that FRODO gathered. Even though the problem was not symmetric, we saved 3 UTIL messages (and few bytes), instead of wasting messages trying to detect a symmetry from the partial symmetries detected by some agents. In contrast, the preprocessing method would use extra messages for the suspected partial symmetries, $\sigma_x$, $\sigma_y$ and $\sigma_z$ which are not symmetries of the whole problem, and switch back to the regular DPOP method eventually.

4.2 SensorDCSP benchmark

For this evaluation, we used the problem SensorDCSP presented in [12] to compare DPOP, SymDPOP, and the preprocessing method of [2]. This problem consists of sensors tracking mobiles moving over a map. At all time, each mobile has to be tracked by exactly three sensors within range. The distribution of the sensors makes this problem naturally distributed. Each variable is a boolean associated to a sensor/mobile couple, and owned by the corresponding sensor. Each constraint over a mobile is distributed over all the sensors detecting it. If two mobiles are close enough to each other, they are detected by the same sensors, thus subject to the same constraints. This makes SensorDCSP a convenient benchmark of symmetric distributed problem, albeit not NP-complete.
50 mobiles & 100 mobiles \\
DPOP & preDPOP & SymDPOP & DPOP & preDPOP & SymDPOP \\
preprocessing & 1042 & 1898 \\
DFSgeneration & 30600 & 17874 & 30600 & 17874 \\
UTIL/VALUE & 5396 & 3212 & 3212 & 5396 & 3212 & 3212 \\
total & 35996 & 22128 & 33812 & 74696 & 37310 & 68946 \\

Table 2. Number of messages for solving a 25-sensor SensorDCSP instance

50 mobiles & 100 mobiles \\
DPOP & preDPOP & SymDPOP & DPOP & preDPOP & SymDPOP \\
preprocessing & 736 & 1742 \\
DFSgeneration & 1696 & 1335 & 1696 & 1335 \\
UTIL/VALUE & 1161 & 900 & 923 & 1161 & 900 & 923 \\
total & 2857 & 2971 & 2619 & 4801 & 4916 & 4230 \\

Table 3. Volume (in kbytes) of messages sent for solving a 25-sensor SensorDCSP instance (average on 100 instances)

We compared the executions of this benchmark with FRODO framework in a shared-memory environment\(^1\) and with mj-express [13] based communication structures on 6 Fujitsu HX600 nodes\(^2\) of the T2K Open Supercomputer [14] for 25 sensors distributed over a map, and with a variable number of mobiles.

Figure 6 shows the computation time averaged on 100 executions. The shared-memory implementation limits the communication effect, therefore SymDPOP improves the computation time by 30 %, versus 50 % for the preprocessing method. However, with a MPI implementation, the preprocessing method hampers the performance, and only SymDPOP manages to yield faster resolution.

We showed in the examples and in the MPI execution that communication volume with SymDPOP was significantly improved. We compared the communication volume in two instances of the problem, with 25 sensors and 50 (and 100) mobiles on Table 3. DPOP and SymDPOP on one hand both generate a complete DFS tree using a large number of small messages. SymDPOP and the preprocessed DPOP use the same number of messages for the UTIL and VALUE propagation, SymUTIL messages being slightly bigger than the UTIL ones. However, the preprocessing messages used to detect the symmetries are extremely heavy. Considering the total communication volume, SymDPOP shows a reduction of more than 10 %, whereas the preprocessed DPOP is an actual regress.

5 Conclusions

In this paper, we presented SymDPOP, a new original DPOP algorithm to exploit strict and partial symmetries dynamically as we explored the DFS tree.

\(^1\) Core2Duo based Linux, 2GB RAM, java-6-sun-1.6
\(^2\) Each node consists of 4 quad-core Opteron 8356, 32GB RAM, java-6-sun-1.6
Fig. 6. Computation time (in ms) for solving a $\{25-x\}$-SensorDCSP instance (average on 100 instances) on a shared-memory FRODO execution.

Fig. 7. Computation time (in ms) for solving a $\{25-x\}$-SensorDCSP instance (average on 100 instances) on a mpj-express FRODO execution on T2K supercomputer.
representation of a problem. The algorithm dynamically reformulates the structure of the problem during UTIL propagation phase of the DPOP algorithm. Implemented on FRODO framework with shared memory (resp. MPI) communication structures, SymDPOP cuts down execution time by 30 % (resp. 20 %) and communication volume by 10 %.

One of the most sophisticated parts of this symmetry breaking method is its behaviour on non-symmetric problems. In contrast with other symmetry detection preprocessing methods, SymDPOP never uses extra messages, eliminating any communication overhead which could hamper the resolution of a non-symmetric problem.

References

Saving Messages in ADOPT-based Algorithms *

Patricia Gutierrez and Pedro Meseguer

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

Abstract. ADOPT and BnB-ADOPT are two related algorithms essential for distributed constraint optimization. They exchange a large number of messages, which is a major drawback for their practical usage. Aiming at increasing their efficiency, we present results showing that some of their messages are redundant so they can be removed without compromising their optimality and termination properties. Removing most of those redundant messages we obtain ADOPT+ and BnB-ADOPT+, which in practice, cause substantial reductions on communication costs with respect to the original algorithms.

1 Introduction

Distributed Constraint Optimization Problems (DCOP) can be found in many real domains for modeling multiagent coordination problems. DCOPs include a finite number of agents, with the usual assumption that each agent holds one variable with a finite and discrete domain. Variables are related by binary cost functions. The cost of a variable assigning a value is the sum of cost functions evaluated on that assignment. The goal is to find a complete assignment of minimum cost by message passing (for details on DCOP definition see [1]).

Considering distributed search for DCOP solving, the first proposed complete algorithm was ADOPT [1]. Later on, the closely related BnB-ADOPT [2] was presented. This algorithm changes the nature of the search from ADOPT best-first search to a depth-first search strategy, obtaining a better performance. Both algorithms are complete, compute the optimum cost and terminate [1, 2]. ADOPT and BnB-ADOPT have a similar communication strategy, using the same set of messages (with small differences in their processes). They also share the same data structures and semantic for store and update their lower and upper bound tables.

ADOPT and –to a lesser extent– BnB-ADOPT exchange a large number of messages. Often, this is a major drawback for their practical application, despite their good theoretical properties (completeness, optimality, termination). Aiming at decreasing the number of exchanged messages without compromising optimality and termination, this paper provides two contributions. First, we present a new version of ADOPT that uses fewer messages than the original algorithm. Some of the proposed modifications come from common knowledge inside the DCOP community, or have been inspired by the

* This work is partially supported by the project TIN2009-13591-C02-02.
BnB-ADOPT algorithm, so we do not claim to be their unique authors. Secondly, we provide two results for detecting redundant messages in our version of ADOPT and BnB-ADOPT. These results allow us to generate new versions of these algorithms, called ADOPT+ and BnB-ADOPT+, which save messages with respect to the original algorithms. Experimentally, we have seen that these new versions cause very significant savings in communication costs (the number of exchanged messages is divided by a factor often larger than 2) on several widely used DCOP benchmarks.

This paper is structured as follows. First we describe our ADOPT version in section 2, which saves some messages with respect to the original ADOPT. We present the communication structure of our ADOPT version, that is closely related with the one of BnB-ADOPT, in section 3. Then, we introduce the main contribution of this paper in section 4, that consist of the results that allow us to detect redundant messages. These results are illustrated with some examples in section 5. Using these results but aiming also at efficiency, we propose new versions of these algorithms, called ADOPT+ and BnB-ADOPT+, in section 6. We report experimental results on the benefits of these new versions with respect to the older ones in section 7. Finally, we conclude in section 8.

2 Reengineering ADOPT

In this section we introduce our ADOPT version. It has some differences with the original ADOPT [1]. We have included these changes for efficiency purposes, but they do not compromise the optimality and termination of ADOPT. Specifically, our version differs from the original ADOPT in the following points:

1. ADOPT sends THRESHOLD messages to children and VALUE messages to children and pseudochildren. Since every time ADOPT sends a THRESHOLD message it also sends a VALUE message, we include all the information in a single VALUE message. The treatment of THRESHOLD and VALUE information is exactly the same, only that it is not splitted in two separate messages. With this simple modification the number of messages is reduced significantly. Obviously, these changes has no effect on optimality and termination of ADOPT. From now on, we consider VALUE and COST messages only.

2. In ADOPT, each agent reads one message of the input queue, processes it and performs backtrack. Executing the backtrack procedure the agent decides if it must change its value. In any case, it sends the corresponding VALUE messages to its children and pseudochildren and a COST message to its parent. This is done for each message until the queue is empty. We modify the algorithm in the following way: on each iteration, the agent reads and processes all messages from the input queue without performing backtrack, and when the queue is empty, it performs backtrack sending the corresponding VALUE and COST messages.

With this modification the algorithm is equally able to find the optimum and terminates. Since all messages are processed, an agent self will update its data structures (context, lb, ub, th) in the same way and order as before, but it will not assign its value or generate messages until the queue is empty. So, there are messages that
would have been sent by self in the original ADOPT that will not be send with this modification. Let us assume an omitted message \(msg_1\) and a final message \(msg_2\) of the same type as \(msg_1\) sent by self. It may happen:

(a) If the omitted \(msg_1\) is a VALUE, not sending it will cause no harm, because if this VALUE is processed, the receiver would updates in its context only self assignment, which will be overwritten anyway when \(msg_2\) arrives.

(b) If the omitted \(msg_1\) is a COST, the same thing happens: the last message \(msg_2\) will overwrite the \(lb\) and \(ub\) tables that might have been updated by \(msg_1\). Also, if the omitted \(msg_1\) would have cause that a new not linked variable would be added to the receiver context, we can assure that this variable will also be added with \(msg_2\), since both messages have the same context variables.

However, when an agent receives VALUE or COST messages, information may be reinitialized if contexts are not compatible. It could be the case that the omitted \(msg_1\) would have caused this effect in the original execution. If this is not caused also by \(msg_2\), then the reinitialization was useless, since the considered obsolete information is now required and will be recalculated. So we can avoid this useless reinitialization. Therefore, these changes has no effect on optimality and termination of ADOPT. This way of processing the input queue reduces a great deal the number of exchanged messages, which is very beneficial for executing this algorithm on difficult instances.\(^1\)

3. Finally, we include a timestamp for every assignment that travels in VALUE messages and in COST messages contexts (one timestamp per value). This timestamp permits to determine which of two assignments is more recent. We allow the receiver context to be updated also by COST messages if they contain more recent assignments. As consequence, an agent self is able to process more updated COSTs instead of discarding them. In the original ADOPT algorithm, COSTs with more recent information are discarded and self would need to wait for delayed VALUE messages until its context is updated with the most recent information. Now, the accepted COST contains the same information as the ones being discarded, so we could have considered them before. Therefore, these changes has no effect on optimality and termination of ADOPT.

In the following, we assume an ADOPT version that includes these changes with respect to the original algorithm [1].

3 Communication Structure

In this section, we summarize the communication structure of our version of ADOPT and BnB-ADOPT. We assume that the reader has some familiarity with ADOPT and BnB-ADOPT code (for a more complete description, the reader should consult the original sources [1, 2]). ADOPT and BnB-ADOPT arrange agents in a DFS tree. An

\(^1\) In his/her review, an anonymous reviewer pointed out that this idea has already been implemented in Jay Modi’s code, and in DCOPolis code of ADOPT (http://dcopolis.sourceforge.net).
agent *self* knows its parent, its children and pseudochildren. Also, *self* holds a context, which is a set of assignations involving *self* ancestors that will be updated with message exchange.

Our ADOPT version uses the following messages:

- **VALUE**(i, j, val, th, context): *i* informs child or pseudochild *j* that it has taken value *val* with threshold *th* in *context*.
- **COST**(k, j, context, lb, ub): *k* informs parent *j* that with *context* its bound are *lb* and *ub*.
- **TERMINATE**(i, j): *i* informs child *j* that *i* terminates.

An agent of our ADOPT version executes the following loop: it reads and processes all incoming messages, and changes value if the lower bound of the current value surpasses the threshold. This strategy allows the agent to change its value whenever it detects a better local assignment. Then, it sends the following messages: a VALUE per child, a VALUE per pseudochild and a COST to its parent. Every time a VALUE or COST message is sent the receiver context is updated. Every time a COST message is sent the receiver lower bound and upper bound are updated. Agents maintain a bounded interval consisting in a lower and upper bound that will be refined during execution. When this interval shrinks to zero (lower bound equals the upper bound) the cost of the optimal solution has been found.

BnB-ADOPT uses the following messages:

- **VALUE**(i, j, val, th): *i* informs child or pseudochild *j* that it has taken value *val* with threshold *th*.
- **COST**(k, j, context, lb, ub): *k* informs parent *j* that with *context* its bound are *lb* and *ub*.
- **TERMINATE**(i, j): *i* informs child *j* that *i* terminates.

A BnB-ADOPT agent executes the following loop: it reads and processes all incoming messages, and changes value if the lower bound of the current value surpasses the upper bound. In that case the value is proven to be suboptimal and it can be discarded as long as the context is not changed. Then, the agent sends the following messages: a VALUE per child, a VALUE per pseudochild and a COST to its parent (for more details, see [2]). Every time a VALUE or COST message is sent the receiver context is updated. Every time a COST message is sent the receiver lower bound and upper bound are updated. As ADOPT, agents maintain a bounded interval that will be refined during the execution. When this interval shrinks to zero (lower bound equals the upper bound) the cost of the optimal solution has been found.

ADOPT and BnB-ADOPT differ in the semantic of threshold values. BnB-ADOPT calculates thresholds in such a way that they represent an estimated upper bound for every agent subtree, and are used for pruning. On the other hand, ADOPT manages thresholds as lower bounds for every agent subtree, that allow agents to efficiently reconstruct partial solutions.

As explained in section 2, our ADOPT version associates with each assignment (either travelling in VALUE or COST messages) a timestamp. This permits COST messages to update the context of receiver, if some value is more recent than the value in
the receiver context. On this respect, our ADOPT version and BnB-ADOPT act in the same way (in BnB-ADOPT timestamps are called counters, referred as ID in [2]).

4 Redundant Messages

In this section we present the results on redundant messages. They are valid for our ADOPT version and for BnB-ADOPT. In the following $i$, $j$ and $k$ are agents, all three executing either our ADOPT version or BnB-ADOPT. Agent $i$, holding variable $x_i$, takes value $v$ when the assignment $x_i \leftarrow v$ is made and $i$ informs of it to its children, pseudochildren and parent. The state of $i$ is defined by (1) its value, (2) its context (as the set of values of agents located before $i$ in its branch, timestamps are not considered part of the context), and (3) for each possible value $v$ and each $j \in children(i)$, the lower and upper bounds $lb(v, j)/ub(v, j)$. A message $msg$ sent from $i$ to $j$ is redundant if at some future time $t$, the collective effect of other messages arriving $j$ between $msg$ and $t$ would cause the same effect, so $msg$ could have been avoided. A message $msg$ sent from $i$ to $j$ containing the assignment $x_i \leftarrow v$ with timestamp $t$ updates $context_j[i]$ with timestamp $t'$ if and only if $t > t'$.

**Lemma 1.** If $i$ takes value $v_1$ with timestamp $t_1$, and the next value it takes is $v_2$ (possibly equal to $v_1$) with timestamp $t_2$, there is no message with timestamp $t$ for $i$ st. $t_1 < t < t_2$.

**Proof.** There is no VALUE sent from $i$ with timestamp between $t_1$ and $t_2$, since $v_1$ and $v_2$ are consecutive values. About COSTs, they build their contexts from the information contained in VALUES. Since no VALUE can include a timestamp between $t_1$ and $t_2$, no COST will contain it for $i$. □

**Theorem 1.** If $i$ sends to $j$ two consecutive VALUES with the same val, the second message is redundant.

**Proof.** Let $V_1$ and $V_2$ be two consecutive VALUES sent from $i$ to $j$ with the same value $val$ with timestamps $t_1$ and $t_2$, $t_1 < t_2$. Between $V_1$ and $V_2$ any messages may arrive to $j$. When $V_1$ reaches $j$, it may happen:

1. $V_1$ does not update $context_j[i]$ ($V_1$ is discarded). When $V_2$ arrives: (a) $V_2$ does not update $context_j[i]$ ($V_2$ is discarded). Future messages will be processed as if $V_2$ would have not been received, so $V_2$ is redundant. (b) $V_2$ updates $context_j[i]$, that has timestamp $t$. There are two options: (i) $t_2 > t > t_1$ and (ii) $t_2 > t = t_1$. Option (i) is impossible because Lemma 1. Option (ii) is possible, but since $t = t_1$ the value contained in $V_2$ is already in $context_j[i]$. About future messages, every message accepted with timestamp $t_2$ of $context_j[i]$ would also be accepted if timestamp of $context_j[i]$ were $t_1$. Since there are no messages with timestamp between $t_1$ and $t_2$ for $i$, we conclude that $V_2$ is redundant.

2. $V_1$ updates $context_j[i] \leftarrow val$, timestamp $t_1$. When $V_2$ arrives: (a) $V_2$ does not update $context_j[i]$; as case (1.a). (b) $V_2$ updates $context_j[i]$; since $V_1$ updated $context_j$ and Lemma 1, the timestamp of $context_j[i]$ must be $t_1$. Updating with
V₂ does not change contextᵢ but its timestamp is put to t₂. Since there are no messages with timestamp between t₁ and t₂ (Lemma 1), any future message that could update contextᵢ with t₂ would also update it with t₁. So V₂ is redundant.

We have not considered the threshold contained in VALUE messages because both algorithms are complete, compute the optimum cost and terminate without the use of thresholds (they are included to increase efficiency). For the original ADOPT algorithm, the threshold is present in the implementation of its termination condition, although it is not essential to use the threshold to detect termination. The ADOPT termination condition is LB = UB, when the size of the bound interval shrinks to zero. Then, the cost of the optimal solution has been determined and agents can safely terminate. Actually, the threshold is always maintained between the LB and UB (using the MaintainThresholdInvariant method, [1]). So when LB = UB, the threshold equals UB.

**Theorem 2.** If k sends to j two consecutive COSTs with the same content (context, lower/upper bound) and k has not detected a context change, the second message is redundant.

**Proof.** Let C₁ and C₂ be two consecutive COSTs sent from k to j with the same content, and contextₖ has not changed between sending them. Any message may arrive to j between C₁ and C₂. Upon reception, the more recent values of C₁ (and later of C₂) are copied in contextⱼ (by PriorityMerge [2]). Copying C₂ more recent values in contextⱼ is not essential. Let us assume that these values are not copied. Then, some messages that would have been ignored between C₁ and C₂ will now be accepted. Since there is no context change between C₁ and C₂, these messages will necessarily include contexts compatible with k context, so they will update timestamps only, generating COSTs with the same bounds. At some point, j will receive all the more recent values of C₂ (necessarily before any context change). After this, j will behave as if it would have copied C₂ more recent values. So if those values are not copied, this will not cause any harm. Because of that, our proof concentrates on bounds. When C₁ arrives, it may happen:

1. C₁ is not compatible with contextⱼ, its bounds are discarded. When C₂ arrives: (a) C₂ is not compatible with contextⱼ, its bounds are discarded. So C₂ is redundant. (b) C₂ is compatible with contextⱼ, its bounds are included in j. Since C₁ was not compatible, there is at least one agent above j that changed its value, received by j between C₁ and C₂. There are one or several VALUEs on its/their way towards k or k descendants. Upon reception, one or several COSTs will be generated. The last of them will be sent from k to j with more updated bounds. C₂ could have been avoided because a more updated COST will arrive to j. So C₂ is redundant.

2. C₁ is compatible with contextⱼ, its bounds are included. When C₂ arrives: (a) C₂ is not compatible with contextⱼ, its bounds are discarded. So C₂ is redundant. (b) C₂ is compatible with contextⱼ, it bounds are included but this causes no change in j bounds, unless bounds are reinitialized. In this case there is at least one agent above j that changed its value, same as case (1.b). So C₂ is redundant.
5 Examples

5.1 Example 1

We present an example to illustrate Theorem 1. Consider agents $i$ and $j$, holding variables $x_i$ and $x_j$ respectively. Agent $i$ sends two consecutive VALUE messages $V_1$ and $V_2$ to agent $j$. As explained in the previous section, if $V_2$ is discarded then $V_2$ is redundant since it does not change agent $j$ state and it has no effect in future message processing. Now, we will consider two possible scenarios in which message $V_2$ is accepted.

Case $V_1$ discarded, and $V_2$ accepted:

Consider agent $i$ sending message $V_1$ to agent $j$ informing the assignment $x_i = b$ with timestamp $t_1$ (Figure 1.a). Agent $j$ has in its context the assignment $x_i = a$ with timestamp $t \geq t_1$, so $V_1$ is discarded. Then, message $V_2$ is sent to agent $j$ with timestamp $t_2$ (Figure 1.b). If $V_2$ is accepted, then timestamp $t_2$ has to be more updated than $t$ ($t_2 > t$). Then, we get $t_2 > t \geq t_1$. The case $t_2 > t > t_1$ is impossible, because from Lemma 1, there is no message with timestamp between $t_1$ and $t_2$ for $i$. The case $t_2 > t = t_1$ is possible, but the value contained in $V_2$ is already in $j$ context (because $t=t_1$). In this case, we explain in the proof of Theorem 1, why $V_2$ is redundant with respect future messages.

Case $V_1$ accepted, and $V_2$ accepted:

Let us consider that agent $i$ sends message $V_1$ to agent $j$ informing the assignment $x_i = b$ with timestamp $t_1$, and is accepted (Figure 2.a). Then $context_j[i]$ is updated and its timestamp is set to $t_1$. Between $V_1$ and $V_2$ many messages may arrive to agent $j$ with different timestamps, and $context_j$ might be updated. But if $V_2$ is accepted, we can assure that timestamp in $context_j[i]$ must be $t_1$ when message $V_2$ arrives (otherwise $V_2$ would not have been accepted). Upon $V_2$ reception, $context_j[i]$ remains the same (because $V_1$ and $V_2$ contains the same assignment) and only timestamp $t_2$ is updated.
Fig. 2. Case $V_1$ accepted, and $V_2$ accepted

(Figure 2.b). Since there is no message with timestamp between $t_1$ and $t_2$ (Lemma 1) is easy to see that any VALUE message $V_3$ or COST message $C_3$ that would update $context_j[i]$ having timestamp $t_2$, will also update it if $context_j[i]$ would have timestamp $t_1$ (Figure 2.c). So updating $t_2$ in $context_j[i]$ makes no difference in future message processing. Therefore, $V_2$ is redundant.

5.2 Example 2

We present a second example to illustrate Theorem 2. Consider agents $k$ and $j$, holding variables $x_k$ and $x_j$ respectively. Agent $k$ sends two consecutive COST messages $C_1$ and $C_2$ to agent $j$. As explained in the previous section, if $C_2$ is discarded then $C_2$ is redundant since bounds $LB$ and $UB$ are not updated and its reception has no effect in future message processing. Now, we will consider two possible scenarios in which message $C_2$ is accepted.

Case $C_1$ discarded, and $C_2$ accepted:

Consider first a higher agent $i$ connected with agents $j$ and $k$. Variable $x_i$ changes its value from $a$ to $b$ and agent $i$ sends the corresponding VALUE messages. The VALUE message sent to agent $j$ is received and $context_j[i]$ is updated, but the VALUE message sent to agent $k$ is delayed (Figure 3.a)). Now, agent $k$ sends a COST message $C_1$ to agent $j$ and this message is discarded because contexts are incompatible on variable $x_i$, since agent $k$ has not received the last VALUE message from agent $i$ yet (Figure 3.a)). Between message $C_1$ and $C_2$ other message may arrive, so let us assume that agent $i$ changes its value again to $a$ and sends the corresponding VALUE messages to agents $j$ and $k$, but message to agent $k$ is delayed once again (Figure 3.b). With these messages delayed, there is no context change in agent $k$ and message $C_2$ is sent. Message $C_2$ is accepted (since context are now compatible) and it will update agent $j$ bounds (Figure 3.c). However, as there are still 2 delayed VALUE messages from agent $i$ to agent $k$, we can assure that when they arrive to agent $k$ there will be a context change, so a new
COST message $C_3$ will be generated with more updated information (Figure 3.c). As we can see, message $C_2$ can be ignored since a message $C_3$ will eventually arrive to agent $j$ and update its bounds with more recent information.

**Case $C_1$ accepted, and $C_2$ accepted:**

Consider that agent $k$ sends message $C_1$ to agent $j$ and it is accepted because contexts are compatible. In this case bounds $LB$ and $UB$ are updated in agent $j$. If message $C_2$ arrives right after $C_1$, it is easy to realize that is redundant, since it would copy the same information in agent $j$. However, between $C_1$ and $C_2$ some messages may arrive, and as result of this the bounds informed by $C_1$ might be reinitialized. This could only happen if a higher agent changes its value. So if a higher agent $i$ changes its value from $a$ to $b$, when the corresponding VALUE message arrives to agent $j$ the bounds are reinitialized. After this agent $i$ must change again its value to $a$ if we want a scenario where $C_2$ message could be accepted. In this case, as there is no context change in agent $k$, we can assure that there is one or more delayed VALUE messages sent to agent $k$ (same case as Figure 3.b). When those delayed messages arrive to agent $k$ there will be a context change, generating a new COST message $C_3$ with more updated information (same case as Figure 3.c). So message $C_2$ can be ignored since a message $C_3$ will eventually arrive to agent $j$ and update its bounds.

**6 New Versions**

Temporarily, we define ADOPT$^+$ as our ADOPT version (see section 2) with the following changes: (1) the second of two consecutive VALUEs with the same $i$, $j$ and $val$ is not sent, (2) the second of two consecutive COSTs with the same $k$, $j$, context, $lb$ and $ub$ when $k$ detects no context change is not sent. In the same sense, we temporarily define BnB-ADOPT$^+$ as BnB-ADOPT with the following changes: (1) the second of two consecutive VALUEs with the same $i$, $j$ and $val$ is not sent, (2) the second of two
consecutive COSTs with the same $k$, $j$, context, $lb$ and $ub$ when $k$ detects no context change is not sent.

**Theorem 3.** $\text{ADOPT}^+$ (respectively $\text{BnB-ADOPT}^+$) terminates with the cost of a cost-minimal solution.

**Proof.** By Theorems 1 and 2, messages not sent by $\text{ADOPT}^+$ (respect. $\text{BnB-ADOPT}^+$) are redundant so they can be eliminated. $\text{ADOPT}$ (respec. $\text{BnB-ADOPT}$) terminates with the cost of a cost-minimal solution [1] (respec. [2]), so $\text{ADOPT}^+$ (respec. $\text{BnB-ADOPT}^+$) also terminates with the cost of a cost-minimal solution. \( \square \)

But the new algorithm is not efficient because we have ignored thresholds. Looking for an adequate threshold management, we define $\text{ADOPT}^+$ (respec. $\text{BnB-ADOPT}^+$) as our ADOPT version (respec. BnB-ADOPT) algorithm with the following changes:

1. Agent $i$ remembers for each neighbor agent $j$ the last message sent.
2. A COST from $j$ to $i$ includes a boolean $ThReq$, set to true when (i) the threshold contained in the last VALUE message received could not be copied into $j$ threshold (ADOPT case) or (ii) $j$ threshold was reinitialized (ADOPT and BnB-ADOPT cases).
3. If $j$ has to send $i$ a COST equal to (ignoring timestamps) the last COST sent, the new COST is sent if and only if $j$ has detected a context change between them.
4. If $i$ has to send $j$ a VALUE equal to (ignoring timestamps) the last VALUE sent, the new VALUE is sent if and only if the last COST that $i$ received from $j$ had $ThReq = true$.

### 7 Experimental Results

Performance is evaluated in terms of communication cost (messages exchanged) and computation effort (non-concurrent constraint checks). We consider also the cycles as the number of iteration the simulator must perform until the solution is found.

We tested our algorithms on binary random DCOPs. In addition, BnB-ADOPT was also tested on meeting scheduling and sensor networks. Binary random DCOP are characterized by $\langle n, d, p_1 \rangle$, where $n$ is the number of variables, $d$ is the domain size and $p_1$ is the network connectivity. We have generated random DCOP instances: $\langle n = 10, d = 10, p_1 = 0.2, ..., 0.8 \rangle$. Costs are selected randomly from the set $\{0, ..., 100\}$. Results appear in Table 1.a and Table 2.a, averaged over 50 instances.

In the meeting scheduling formulation, variables represent meetings, domain represent the time slot assigned for each meeting, and there are constraints between meetings that share participants [3]. We present 4 cases with different hierarchical scenarios. Results of the execution appear in Table 2.b, averaged over 30 instances.

In the sensor network formulation, variables represent targets, domain represent the time slots in which they might be tracked, and there are constraints between adjacent targets [3]. We present 4 cases with different topologies scenarios. Results of the execution appear in Table 2.c, averaged over 30 instances.

On random DCOPs, $\text{ADOPT}^+$ showed clear benefits on communication costs with respect to our ADOPT version. It divided the number of exchanged messages by a factor from 1.1 to almost 3, maintaining the number of cycles practically constant.
### (a) Random DCOPs.

<table>
<thead>
<tr>
<th>p1</th>
<th>#Messages</th>
<th>#NCCC</th>
<th>#Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>468</td>
<td>2,827</td>
<td>29</td>
</tr>
<tr>
<td>0.3</td>
<td>1,063,300</td>
<td>36,330,792</td>
<td>82,604</td>
</tr>
<tr>
<td>0.4</td>
<td>16,810,366</td>
<td>851,171,565</td>
<td>1,461,910</td>
</tr>
<tr>
<td>0.5</td>
<td>410,414,194</td>
<td>12,187,384,497</td>
<td>12,826,157</td>
</tr>
</tbody>
</table>

Table 1. Results of our ADOPT version (first row) compared to ADOPT+ (second row)

In the same sense, experiments with random DCOPs show that our algorithm BnB-ADOPT+ reduces the number of messages by a factor from 2 to 3 when connectivity increases with respect to BnB-ADOPT. For meeting scheduling, messages are reduced by a factor of at least 2, and for sensor networks, by a factor between 3 and 6. We have achieved important savings for all problems tested. BnB-ADOPT+ was able of processing only half of messages (or less) and reach the optimal solution maintaining the number of cycles practically constant. This can be sustained in the empirical fact that there is a large reduction in messages, a slight reduction in the non-concurrent constraint checks, and the number of cycles remains constant.

### 8 Conclusion

We have presented two contributions to increase the performance of distributed constrained optimization algorithms. First, we describe our version of the ADOPT algorithm, which saves some messages with respect to the original algorithm. Secondly, we present theoretical results to detect redundant messages in our ADOPT version and in BnB-ADOPT. Using these results we generate two algorithms, called ADOPT+ and BnB-ADOPT+, which caused substantial savings with respect to our ADOPT version and BnB-ADOPT, when tested on commonly used benchmarks of the DCOP community.
Acknowledgements

Authors sincerely thank reviewers for their useful criticisms.

References

Concurrent Forward Bounding for DCOPs

Arnon Netzer, Amnon Meisels, and Alon Grubshtein
Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, 84-105, Israel
{netzerar,am,alongrub}@cs.bgu.ac.il

Abstract. A new search algorithm for solving distributed constraint optimization (DCOP) problems is presented. The proposed algorithm performs concurrent search on non intersecting parts of the global search space, using multiple search processes. Each Search Process uses synchronous forward bounding in its sub-space, to prune the remaining search space. Concurrent Forward Bounding (ConcFB) dynamically spawns new Search Processes in parts of the search space which have higher chances to produce tighter bounds on the cost of the global solution.

An extensive experimental evaluation is presented comparing ConcFB to the leading DCOP algorithms. ConcFB significantly outperforms other DCOP algorithms in both run time and communication load.

A new measure of concurrency of computation for DCOP algorithms is introduced the Concurrency Factor (CF). Each feature of ConcFB is shown to increase its concurrency. Unlike other DCOP algorithms, the CF of ConcFB increases with problem density and difficulty.

1 Introduction

Distributed Constraint Optimization Problems (DCOPs) form a powerful framework for distributed problem solving that has a wide range of application in Multi Agent Systems, [1] [2]. In a typical DCOP agents need to assign values in a way that would minimize (or maximize) the sum of the resulting constraint costs. A DCOP algorithm searches the combinatorial assignments space, in order to find the combination of assignments that will produce this minimum.

Several DCOP algorithms where introduced over the last year. These include SyncBB [3], ADOPT [2], ODPOP [4], BnB-ADOPT [5] and Asynchronous Forward-Bounding [6].

The present paper proposes a new DCOP algorithm: Concurrent Forward Bounding (ConcFB). The proposed algorithm is based on two main ideas: it performs multiple search processes on disjoint parts of the search space [7], each search process performs Forward Bounding in order to achieve early pruning of the search space[6]. The algorithm incorporates dynamic splitting that focuses the search on parts of the search space that are more likely to produce lower bounds.

ConcFB benefits from the pruning power of forward bounding, and complements it with concurrency of several search processes. When the performance of the proposed
ConcFB algorithm is compared to the leading DCOP algorithms, it turns out to be superior in terms of both measures - run time and network load.

The ConcFB algorithm is described in section 3, and its correctness proof is in section 4. Section 5 presents extensive experimental evaluation of ConcFB and its comparison evaluation with the leading DCOP algorithms. The paper conclude in 6 in which Concurrency Factor (CF), which is an objective quantitative measurement of the degree of concurrency of a DCOP algorithm, is introduced.

2 Distributed Constraints Optimization

A DCOP is a tuple \( \langle A, X, D, R \rangle \). \( A \) is a finite set of agents \( A_1, A_2, \ldots, A_n \). \( X \) is a finite set of variables \( X_1, X_2, \ldots, X_m \). Each variable is held by a single agent (an agent may hold more than one variable). \( D \) is a set of domains \( D_1, D_2, \ldots, D_m \). Each domain \( D_i \) contains a finite set of values which can be assigned to variable \( X_i \). \( R \) is a set of relations (constraints). Each constraint \( C \in R \) defines a non-negative cost for every possible value combination of a set of variables, and is of the form \( C : D_{i_1} \times D_{i_2} \times \ldots \times D_{i_k} \rightarrow R^+ \). A binary constraint refers to exactly two variables and is of the form \( C_{ij} : D_i \times D_j \rightarrow R^+ \). A binary DCOP is a DCOP in which all constraints are binary.

An assignment 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. \( \text{vars}(PA) \) is the set of all variables that appear in PA, \( \text{vars}(PA) = \{X_i \mid \exists a \in D_i \land (X_i, a) \in PA\} \). A constraint \( C \in R \) of the form \( C : D_{i_1} \times D_{i_2} \times \ldots \times D_{i_k} \rightarrow R^+ \) is applicable to PA if \( X_{i_1}, X_{i_2}, \ldots, X_{i_k} \in \text{vars}(PA) \). The cost of a partial assignment PA is the sum of all applicable constraints to PA over the assignments in PA. A full assignment is a partial assignment that includes all the variables (\( \text{vars}(PA) = X \)). A solution is a full assignment of minimal cost.

In the duration of this paper we assume all constraints to depend on exactly two variables (binary constraints), and one variable per agent. The terms variable and agent are used interchangeably.

3 Concurrent Forward Bounding

Concurrent Forward Bounding (ConcFB) performs multiple concurrent search processes on disjoint parts of the search space. A simple example of producing such disjoint subspaces would be to randomly choose a variable \( X_i \), take each of its possible assignments \( d \in D_i \) to be the only allowed assignment for a given search process, and set each search process to scan the entire sub-space that has a specific value \( d \) assigned to \( X_i \). This will result in a number of search processes which equals the domain size of \( X_i \).

We will later show how search processes can be dynamically spawned in “interesting” parts of the global search space.

Each Search Process (SP) maintains a global order of the agents. Each SP also maintains a Current Partial Assignment (CPA), holding all the assignments of variables that are assigned values in the SP. Each agent holds a designated data structure for every SP that passes through it, maintaining information about the specific SP. In particular
each variable holds a current partial domain of values that are not assigned yet, to the specific SP.

When an agent assigns a value to its variable on a CPA that represent a given SP, it sends forward a Comp-LB message to every neighbor (e.g. constrained agent) that is after it in the global order of the specific SP. The Comp-LB message requests the receiving agent to compute a Lower Bound (LB) on the partial assignment on the CPA.

When an agent receives all of the Lower Bound estimations for a given assignment in a given SP, it adds them to the cost of the CPA, and compares the resulting cost to the Upper Bound. If the result is larger or equal to the Upper Bound, the agent tries to assign a different value. If no such assignment exists, it backtracks. If the result is smaller than the Upper Bound, the CPA, with the var assignment and the LBs collected so far, is sent to the next variable in the global order of the specific SP.

When a variable receives a Comp-LB request for a given CPA, it calculates the cost of each of its assignments for the given CPA. The Lower Bound for the Variable is taken to be the minimum over the costs:

\[
LB_{j}^{X} = \min_{d \in \text{Dom}(X_{j})} \sum_{X_{i} \in \text{CPA}} \text{Cost}(X_{i}^{a}, d)
\]

Where \(X_{i}^{a}\) is the assignment of variable \(X_{i}\) and the lower bound is computed by \(X_{j}\).

Search processes can split, dynamically generating new SPs. A split is generated in response to SPLIT message. An agent that receives a SPLIT request checks the amount of assignments left in the current domain of the specific SP. If more than one assignment is left, then the current domain is split in half, one half remain in the original SP, and the other is used as the current domain of a newly created SP. If the amount of assignments left in the current domain is not enough for split, a SPLIT request is sent to the next agent in the global order of the SP, requesting it to split. Agents must keep track of all split SPs and their originating SP. A backtrack can be performed only after all split SPs and originating SP have exhausted their search of the sub space.

A global variables order used by all SPs will result in an uneven work distribution between agents. The present realization uses random order of variables for each SP.

Each time a new global upper bound is found, it is broadcast to all agents. When an agent receives an Upper Bound message that is lower that it’s current value, it disregards the incoming message. Every message carries the best Upper Bound known to its sender.

### 3.1 Algorithm Description

Figures 1 and 2 presents the pseudocode of every agent running ConcFB. There are six types of messages used in ConcFB:

- **CPA** - Contains the SP ID and the Current Partial Assignment for that given SP. The CPA contains an \(LB\_List\) of lower bounds collected so far. An agent receiving a CPA message tries to extend it with its own assignment.
- **BackTrack\_CPA** - Informs of inability to extend the CPA.
- **Comp\_LB** - A copy of a CPA for a given SP. Requesting to compute an estimate of the lower bound for the given CPA.
– Split - An agent receiving a Split message will attempt to split the SP.
– UpperBound - An agent will replace its upper bound with the new one if the new one is smaller.

The algorithm starts with the First Agent creating the initial Search Processes (routine Initialize_SPs() in Figure 1). Each SP has a domain of 1 value out of the First Agent Domain (line 3). The variable order is randomized for each SP (line 4). Each SP calls assign_CPA() to start the Search Process.

Upon receiving a CPA message, an agent creates a local structure for the SP on the received CPA. This data structure includes the current domain for the SP, the CPA received and the LB_list received. If the LB_list contains an LB previously obtained by the agent, this LB is removed from the LB_list (line 4 in Received_CPA()). An origin_SP.splitList is created, in which the newly received SP serves as both the origin_SP and an item in the splitList (line 5). If the number of assignments in the CPA is large enough, a Split message is sent to the SP originator (line 7). The agent then tries to assign its own value on the CPA by calling Assign_CPA() (line 8).

Upon receiving a Backtrack_CPA message, the agent locates its local structure SP by using the SP_ID. It then removes its assignment for the relevant SP (line 2 of Received_BacktrackCPA()), and removes the local_cost from the SP.CPA.cost. If the current domain of the relevant SP is empty, or if the agent has no constraints with any of the unassigned agents of the relevant SP.CPA, the agent tries to backtrack (line 5). Otherwise, it tries to assign a new value by calling Assign_CPA() (line 7).

Any LB received via a Reported LB message, from an agent that already appears in SP.LB_list replaces the existing one. Otherwise the new LB is added to SP.LB_list (Received_ReportedLB() line 2). The agent waits for all Reported_LB messages to arrive. If all needed LB arrived (line 3), then, if the cost of the CPA plus the sum of all contributions in SP.LB_list is lower than the UpperBound (line 4), the CPA is sent to the agent who is next in the global order of the specific SP. Otherwise the assignment is no good and the agent calls the Received_BacktrackCPA() (line 6).

When an agent receives a Split message and the SP cannot be split, the message is sent to the next agent in the global order (line 8 in Received_Split()).

When an agent tries to assign a value, it first checks whether the domain of the relevant SP was not exhausted (line 1 in Assign_CPA()). If it was, the agent Backtracks. Otherwise, the assignment with the lowest cost with respect to the relevant SP.CPA is selected (line 2). If the cost of SP.CPA with the added assignment and the lower bounds collected for this CPA is not smaller than the known Upper Bound, the agent backtracks (line 5). If the cost of SP.CPA is smaller than the known Upper Bound and SP.CPA has a full assignment, a new Upper Bound has been found. The new Upper Bound is sent to all agents and a Backtrack is performed on the relevant SP (lines 8 and 9). If it is not a full assignment, a Collect_LB message is sent to all unassigned neighbors.

When an agent attempts to Backtrack, it first removes the relevant SP from its origin_SP.splitSet (line 2 of Backtrack()) and removes all data structures relevant to this SP. If the origin_SP.splitSet of the relevant SP is not empty then a BackTrack cannot be performed yet, and the routine ends. If origin_SP.splitSet is empty, then it is time to Backtrack. The origin_SP.splitSet is removed, and a Backtrack_CPA message
ConcFB Main:
1. done ← false
2. if (IA)
3. initialize_SPs()
4. while (not done)
5. pop msg
6. if(msg.ub < UB)
7. UB = msg.ub
8. switch msg.type
9. CPA → Received_CPA(msg.CPA, msg.LB.List)
10. Backtrack_CPA → Received_BacktrackCPA(msg.SP_ID)
11. Comp_LB → Received_CompLB(msg.CPA)
12. Reported_LB → Received_ReportedLB(msg.SP_ID, msg.LB, msg.sender)
13. Split → Received_Split(msg.SP_ID)
14. Terminate → done = true

Initialize_SPs():
1. for i ← 1 to domain.size
2. create SP_i
3. SP_i.domain ← domain[i]
4. SP_i.RandomizeAgentsOrder()
5. Add SP_i to rootSp.splitSet
6. Assign_CPA(SP_i)

Received_CPA(CPA, LB_List):
1. Create local SP data structure
2. SP.CPA ← CPA
3. SP.LB_List ← LB_list
4. SP.LB_list.remove(current_var)
5. Add SP to SP.splitSet
6. if (SP.CPA.nofAssignedVars > Split Depth)
7. send Split to SP.initiator
8. Assign_CPA(SP)

Received_BacktrackCPA(SP_ID):
1. retrieve local SP with SP_ID
2. SP.removeLastAssignment
3. SP.CPA.cost← = LocalCost
4. if (SP.currentDomain.isEmpty or no lower priority neighbors)
5. Backtrack()
6. else
7. Assign_CPA(SP)

Received_CompLB(CPA):
1. $LB = \min_{d \in Domain} \sum_{X_i \in SP.CPA} Cost(X_i^*, d)$
2. send Reported_LB to CalcLB originating variable

Fig. 1. Pseudo code of ConcFB.
Received_ReportedLB(SP_ID, LB, neighbor):
1. retrieve local SP with SP_ID
2. SP.LB_LIST[neighbor] = LB
3. if (got ReportedLB from all unassigned neighbors)
   4. if (SP.CPA.cost + SP.LB_LIST.sum < UpperBound)
      5. send CPA(SP.CPA, SP.LB_LIST) to next agent
   6. else
      7. Received_BackTrackCPA(SP_ID)

Received_Split(SP_ID):
1. retrieve local SP with SP_ID
2. if (SP.currentDomain.size > 1)
   3. create newSP
   4. newSP.domain ← SP.splitDomain
   5. Add newSP to SP.splitSet
   6. Assign_CPA(newSP)
3. else
   4. send Split to next agent

Assign_CPA(SP):
1. if (SP.Domain not empty)
   2. SP.CPA ← best local assignment
   3. SP.CPA.cost+ = LocalCost
   4. if (SP.CPA.cost + SP.LB_LIST.sum >= UB)
      5. Backtrack(SP)
   6. else
      7. if (is full(SP.CPA))
         8. Found_new_UB(SP.CPA.cost)
         9. Backtrack(SP)
     10. else
         11. send Collect_LB(SP.CPA) to all unassigned neighbors
12. else
    13. Backtrack(SP)

Backtrack(SP):
1. Origin_SP ← retrieve Origin SP of SP_ID
2. remove SP from origin_SP.splitSet
3. if (origin_SP.splitSet.isEmpty)
   4. remove origin_SP.splitSet
   5. if (ID == FirstAgent)
      6. if (SP.List.isEmpty)
         7. send Terminate to all Agents
         8. done ← true
         9. solution ← UB
     10. send Backtrack_CPA to last assignee

Fig. 2. Pseudo code continue.
is sent to the last assignee of the relevant SP (line 10). If \texttt{origin}\_\texttt{SP.splitSet} is empty, and this is the first global agent, then the algorithm stops. A Terminate message is sent to all agents, and the Upper Bound registered in the first global agent is the algorithm’s result.

### 3.2 Dynamic Splitting

Figure 3 presents an example of the CPA behavior for each of 4 Search Processes, a histogram of the number of assignments of the CPA in each variable, and the total number of CPA assignments, per SP. There are 14 variables with an average of 4 neighbors per variable, and a domain size of 4. Variable order was the same for all search processes, and no dynamic splitting was used.

It is clear from Figure 3 that most of the CPA assignments are in the middle part of the histogram. In other words, in the middle depth of the search space. It is also clear that the distribution of CPA assignments is not uniform among the search processes. In this example SP2 has about half the CPA assignments than any of the other SPs. This will result in SP2 exhausting its subspace much earlier than the other SPs, and lowering the concurrency of the algorithm. Spawning more SPs of the harder working SPs will even the work load between them, and insure that enough SPs are alive throughout the search.

![Figure 3. CPA assignments per SP](image)

Figure 4 presents the histogram of the number of assignments of the CPA in the last 3 variables, and the number of Upper Bounds found per SP. As can be seen, the number of Upper Bounds found can vary significantly between SPs. In fact though SP1 and SP3 has the same total number of CPA assignments (Figure 3) SP1 has 8 times more Upper Bounds occurring. Spawning more SPs that are producing more Upper Bounds, that may enable a faster detection of Upper Bounds and a better pruning of the search space.
3.3 Random Variable Ordering

Figure 5 presents the distribution of constraints checks between the variables, when there is no random reordering of variables for each SP. The figure is the averaged result of 50 ConcFB runs over random problem with 14 variables, an average of 4 neighbors per variable, and domain size of 4. It is clear that the work load is not distributed evenly between variables. In fact most of the work is done by the variables with lower priority. This is due to the fact that in forward bounding algorithms most of the work is done by the computation of lower bounds in look ahead, and not in the CPA assignments.

Figure 6 shows the result of the same run, with random variable ordering for each Search Process. Figure 6 shows an even spread of work load between the variables (except for the first one which is first for all search processes). The highest number of CC per variable is about half in the random order scenario. Clearly a better distribution of work among the agents.
4 Correctness of Conc-FB

To prove the correctness of a search algorithm for DCOP one needs to prove that it is sound, complete and that it terminates.

Theorem 1. ConcFB is sound

Proof. The only place in the algorithm that terminates the search and reports a solution is in Backtrack() line 8. In which the UB is reported as a solution. A new UB is created only in Assign.CPA() line 7, and only for CPAs that are fully assigned (line 6). Therefore, the solution necessarily represents the cost of a full assignment. □

Lemma 1. ConcFB traverses all sub-spaces

Proof. The search space is divided into subspaces in two functions - Initialize.SPs() and Receive.Split(). The domain of the first agent is split in Initialize.SPs() (line 3). Each of the resulting domains is assigned a Search Process that attempts to extend it to a full assignment.

In Receive.Split() the part of the domain of an agent within a given SP, is split. One resulting SP is the original SP. The new SP is searched by the resulting CPA (line 3). Hence all subspaces are being searched in Conc.FB. □

Theorem 2. ConcFB is Complete

Proof. To prove the completeness of ConcFB one must show that only CPAs that cannot be expanded to a full assignment with lower cost than the UB are pruned. In addition one must show that no SP on a subspace search is stopped before it is completed.

Pruning is done in 2 places in the algorithm. The first is in Assign.CPA() after a variable seeking its best assignment for extending a given CPA(line 2) discovers that adding the new assignment causes the CPA cost to be greater than the UB (line 3): this is clear case their where no solution can be found by extending the assignment of the CPA.

The second pruning place in the algorithm is in function Received.ReportedLB(). In this case after receiving lower bounds from all lower priority neighbors CPA.cost +
Accumulated \( LB < UB \). Clearly any full assignment extending the CPA will cost no less, and there is no need to try and extend this CPA.

Finally we insure that no CPA is backtracked before it is fully searched (or pruned) in line 2 of \( \text{Backtrack()} \), by making sure that all SPs that are trying to extend the CPA have completed (e.g. returned) before the backtrack can be performed. □

**Theorem 3. ConcFB Terminates**

**Proof.** Since no assignments on a CPA is visited more than once during the search process, and since each CPA is either extended or backtracked at each step of the algorithm, and since the search space is finite, it is obvious that the algorithm will terminate. □

5 Experimental Evaluation

Two measures of performances are routinely used to evaluate distributed DCOP algorithms. Runtime in the form of non-concurrent constraints checks (NCCCs), and network load in the form of the total number of messages sent.

The present evaluation compares ConcFB to two DCOP algorithms - BnB-ADOPT [5] and ODPOP [4].

For ODPOP the main computational operation is the comparison of combination of assignments, sent to each computing agent by its offspring on the pseudo tree [4]. This operation performed by each agent in order to find an assignment for itself that is compatible with all assignments of its ancestors on the pseudo-tree.

All experimental evaluation of run time, are given for the three algorithms in non concurrent logical operations. For ODPOP these are compatibility checks, and in BnB-ADOPT and ConcFB they are constrain checks.

BnB-Adopt was shown to be superior to ADOPT and to NCBB [5]. The present evaluation use BnB-Adopt with DP2 as a preprocessing phase for h vals [8], for reference the synchronous branch & bound algorithm (SBB) with best val assignment was used.

Experiments were performed on randomly generated DCOPs with 12 agents, domain size 3 and costs randomly assigned between 0 and 10000. The average number of neighbors varied between 2 and 5, all results where averaged over 50 randomly generated problems.

Figure 7 presents run time in terms of Non Concurrent Logic Operations (NCLO) of all 4 algorithms. For 5 neighbors ODPOP performed more then \( 10^7 \) NCLOs, and was left out of Figure 7 (3 orders of magnitude slower than all others).

The result of removing ODPOP from Figure 7 are in Figure 8. It is easy to see that the advantage of ConcFB over BnB-ADOPT grows significantly when the average number of neighbors grows.

The total number of messages for the same experiments follows a similar pattern and are shown in Figure 9. In this case BnB-ADOPT is much worse then all other algorithms. The high message count for BnB-ADOPT is to be expected, since BnB-ADOPT is using the same message passing and communication scheme of ADOPT [5], and ADOPT was shown to have a very high message count [9].
Fig. 7. NCLO - all 4 algorithms

Fig. 8. NCLO - ConcFB and BnB-ADOPT

Fig. 9. Number Of Messages
When BnB-ADOPT is left out, the picture becomes clearer. The total number of sent messages (Figure 10) of both ODPOP and ConcFB stays low as the density of the DCSP increases. ODPOP sends less messages than ConcFB. As the number of neighbors increases, the difference in number of messages between the two algorithms decreases. For 5 neighbors they send the same amount of messages.

Another experiment was done on MaxCSP problems. In this case, we randomly generated MaxCSP problems with 14 variables, domain size equals to 6 and an average of 4 neighbors (P1 = 2.85). Figure 11 shows the run time in NCCCs for each algorithm. ODPOP is not presented since it could not complete even P2=01 problems in reasonable time. The total number of messages for the same experiment are in Figure 12.

6 Discussion

A new algorithm for Distributed Constraints Optimization Problems (DCOPs) was presented. The algorithm performs concurrent search processes on disjoint parts of the DCOP. Each search process uses forward bounding on its search space to enhance its
The proposed algorithm - Concurrent Forward Bounding (ConcFB) was described and its correctness proven.

Two measures of performance were used in the extensive performance evaluation, and ConcFB was compared to the leading DCOP algorithms. ConcFB is four times faster than BnB-ADOPT for randomly generated DCOP in medium density and sends six times less messages. For the same problems it sends twice as many messages as ODPOP but its run time is ten times faster. As the difficulty of the random DCOP grows, the speed up of ConcFB compared to ODPOP grows exponentially. For DCOP with 5 neighbors, on the average, ODPOP does not complete its run in a few hours (Figure 7).

An interesting measure that can demonstrate the efficiency of computation during search of DCOP algorithms can be defined as follows. The Concurrency Factor (CF) is the ratio of the total amount of computation performed by all agents running an algorithm, to the amount of non-concurrent computation. The latter is the common measure of distributed asynchronous run-time [10].

In a completely synchronized algorithm (e.g. SBB) the total number of constraint checks is equal to the NCCC, hence $CF = 1$, on the other hand, the theoretical bound for the $CF$, is when all agents are completely unsynchronized, and there is no dependency between all computation. In this case $CF = N$ where $N$ is the number of agents. It is expected that in most DCOP problems, since agent computation is dependent upon other agents, the real values of $CF$ will be much lower than $N$.

In order to gain insight into the concurrent behavior of ConcFB one can compare the CF of different variants of the algorithm. 4 variant of the ConcFB algorithm were used, 1 - running only a single SP. 2 - running multiple SPs but no random variable ordering and no dynamic splitting. 3 - running multiple SPs with random variable ordering but no dynamic splitting. 4 - full featured ConcFB.

As is evident in Figure 13 the total computation (CC) is only lower for the full featured ConcFB. In contrast, the amount of non-concurrent computation (e.g. run time in NCCC) decreases as the CF increases.

Figure 14 shows the Concurrency Factor of all algorithms. The CF of the synchronous algorithm SBB is exactly 1. ODPOP uses a pseudo tree to achieve concurrency. This deteriorates as the number of neighbors grows. BnB-ADOPT is very concurrent by nature, in addition to the concurrency introduced by working on different
Fig. 13. behavior of ConcFB algorithm

branches of the pseudo tree. This explains the lowering of the CF on BnB-ADOPT as the number of neighbors grows. In contrast to all other algorithms, the concurrency of ConcFB actually benefits from increase in the number of neighbors.

Fig. 14. Concurrency Factor - all algorithms

References

Cooperation between search and surveillance agents in DCOP_MST

Roie Zivan and Katia Sycara,
Robotics Institute,
Carnegie Mellon University,
5000 Forbes Avenue,
Pittsburgh, PA, 15213, USA,
{zivanr,katia}@cs.cmu.edu

Abstract. A team of mobile sensors working together towards a common goal of covering targets in some area, can be modeled by DCOP_MST. DCOP_MST is a model based on distributed constraints optimization that enables representation of the dynamic elements in a mobile sensing agents team problem, e.g., environment changes, changes in the agents’ tasks and technology failures. Agents in DCOP_MST, perform local DCOP algorithms, adjusted to this model, in order to adjust their deployment to the current state of the dynamic environment. In our previous work on DCOP_MST we assumed complete and accurate knowledge on the location and importance of targets in the area. This information might not be available in realistic scenarios. In this paper we consider two teams of agents, (a) sensors with advanced mobility and sensing tasked to detect targets, and (b) sensors with limited mobility tasked with target surveillance. We demonstrate how the DCOP_MST model can be used by these two teams to fulfill their sub-tasks, and how it can be used for cooperation among the two teams. Our experimental study demonstrates how an increased level of cooperation among the two teams improves the performance of both teams and improves the overall result on fulfilling the mutual coverage goal.

1 Introduction

Some of the most challenging applications of multi agent systems include a team of mobile agents with sensing abilities which are required to cover a given area to achieve a common goal. Various examples are a network of sensors tracking enemy targets, rescue teams searching for survivors and teams of unmanned vehicles (UVs) searching an unfamiliar terrain. These applications are often large and complicated and thus, it is reasonable to assume that the agents cooperating to achieve coverage in these applications, reside on mobile sensors of different types and are divided into different teams where each of them has its own subtask. These subtasks are derived from the general common goal.

In a previous paper [12] we presented DCOP_MST, a model based on distributed constraint optimization for representing and solving problems of teams of mobile sensing agents [12]. The DCOP_MST model allows the representation of multiple dynamic
elements, which the applications discussed above include, such as changes in the environment, changes caused by technology failures and changes caused by agents’ movements. We demonstrated in [12] how the DCOP_MST and the novel algorithms we proposed for this model, are successful in maintaining high level coverage in dynamic environments. The innovation of the DCOP_MST model is its ability to maintain dynamic domains and dynamic sets of neighbors. In addition, the environment is modeled as a function (environment requirement function, ER) that assigns to each point in the area its coverage importance, e.g., targets of interest are expected to have very high ER values.

However, in [12] we assumed that the ER function, which agents refer to and perform upon, is accurate and there is no uncertainty. This assumption is limiting when considering military and rescue applications. In this paper we extend the work on DCOP_MST. Here, the model is used to represent two teams of agents which reside on mobile sensors. The mutual goal of these teams performing in the same area, is surveillance coverage of all targets in the area according to their importance. The first team includes agents with advanced mobility and accurate sensing technology. The sub-task of this team of agents is to detect the targets and inform the surveillance agents (agents from the other team) of their location. The second team is the team of agents we have presented [12], which in this paper we refer to as “surveillance agents”. Their task is to maintain coverage according to the targets’ importance [12].

We demonstrate in this paper that the differences between the sub-tasks of the two teams require different search strategies. For the surveillance team, in [12] we found that the most successful algorithm was MGM enhanced with exploration methods which maintain reasonable coverage (prevent from abandoning covered targets). The requirement of maintaining long term coverage encouraged an algorithm in which the coordination between agents is high (an agent moves only if her neighbors do not) and monitored exploration level. On the other hand, search agents are expected to benefit from high level of exploration which will allow them to reach the entire area in minimal time.

To accomplish the different requirements of the search team’s task, we designed a new algorithm based on DSA [9]. The agents use a second ER map/function which initially includes only vague probabilities on the location of the targets. The agents update this function by reducing the probability of areas which they recently visited. Agents select their next position according to this function. Our experiments show that in contrast to standard DSA [9], our algorithm does not suffer from thrashing as the probability for changing location increases.

Although the tasks of the two teams are different and require different algorithms, awareness of the global goal and the subtask of the other team can enable agents to cooperate with agents from the other team [3, 4]. We propose three levels of cooperation:

1. Agents from the search and detection team participate in the surveillance process of targets within their sensing range. This requires agents of the search and detection team to be aware of the location of the targets which were already found and to perform surveillance as long as they are in range.
2. Agents from the surveillance team inform the search and detection agents on elements in their sensor range which they suspect to be targets.
3. Search and detection agents which are covering a target, do not leave it until it is covered by surveillance agents.

In our experimental study we evaluated both the success of the sub-tasks of the two teams and the overall result when these different levels of cooperation are performed. Our empirical evaluation reveals that higher level of cooperation improves both.

The rest of this paper is organized as follows: After presenting the standard DCOP model in Section 2, we present DCOP\_MST in Section 3. Section 4 presents the algorithm based on MGM performed by the surveillance agents. In Section 5, we present the extension to the model and the algorithm based on DSA performed by the search agents team. Section 6 discusses the different levels of cooperation we propose between the two sub-teams. Our empirical study is presented in Section 7 followed by our conclusions.

2 Distributed Constraint Optimization

A DCOP is a tuple \(< A, X, D, R >\). \(A\) is a finite set of agents \(A_1, A_2, \ldots, A_n\). \(X\) is a finite set of variables \(X_1, X_2, \ldots, X_m\). Each variable is held by a single agent (an agent may hold more than one variable). \(D\) is a set of domains \(D_1, D_2, \ldots, D_m\). Each domain \(D_i\) contains the finite set of values which can be assigned to variable \(X_i\). \(R\) is a set of relations (constraints). Each constraint \(C \in R\) defines a non-negative cost for every possible value combination of a set of variables, and is of the form \(C : D_{i_1} \times D_{i_2} \times \ldots \times D_{i_k} \rightarrow \mathbb{R}^+ \cup \{0\}\). A binary constraint refers to exactly two variables and is of the form \(C_{ij} : D_i \times D_j \rightarrow \mathbb{R}^+ \cup \{0\}\). A binary DCOP is a DCOP in which all constraints are binary. 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. \(\text{vars}(PA)\) is the set of all variables that appear in \(PA\). \(\text{vars}(PA) = \{X_i | \exists a \in D_i \land (X_i, a) \in PA\}\). A constraint \(C \in R\) of the form \(C : D_{i_1} \times D_{i_2} \times \ldots \times D_{i_k} \rightarrow \mathbb{R}^+ \cup \{0\}\) is applicable to \(PA\) if \(X_{i_1}, X_{i_2}, \ldots, X_{i_k} \in \text{vars}(PA)\). The cost of a partial assignment \(PA\) is the sum of all applicable constraints to \(PA\) over the assignments in \(PA\). A full assignment is a partial assignment that includes all the variables \((\text{vars}(PA) = X)\). A solution is a full assignment of minimal cost.

3 DCOP\_MST

DCOP\_MST includes a number of concepts which are not used in the standard DCOP model. The first is the position of an agent. We denote the current position of agent \(A_i\) by \(\text{cur_pos}_i\). The position of the agent is its current assignment but in DCOP\_MST it is a physical position in the area (which can be represented by coordinates).

Second, we define two ranges which are essential in DCOP\_MST. The first is the Sensing Range \((SR)\) of agents. The sensing range is the effective coverage range of the agent, i.e., agent \(A_i\) can detect and cover all the targets that are within its sensing range from \(\text{cur_pos}_i\). The Mobility Range \((MR)\) is the range that an agent can move in a
single iteration of the algorithm. We denote the sensing range and the mobility range of
agent \( A_i \) by \( SR_i \) and \( MR_i \) respectively.

After defining these concepts we can redefine the set \( D \). A domain \( D_i \) of agent
\( A_i \) includes all the alternative positions which are within \( MR_i \) from \( \text{cur_pos}_i \). The
resulting domain is a dynamic set \(^1\).

For each agent \( A_i \) a credibility variable \( Cred_i \) is defined. The credibility variable is
a real positive number which is calculated by a reputation model.

We further define an environmental requirement function \( ER \). This function ex-
presses for each point in the area, the required joint credibility amount (the sum of the
credibility variables) of agents with appropriate sensing range so that the given point
can be adequately sensed (i.e. covered). Function \( Cur\_DIFF \) calculates for each point
in the area the difference between the current value of the \( ER \) function and the sum
of the credibility values of the agents which are currently covering it. Formally, if we
denote the set of agents within sensing range from point \( p \) by \( SR^p \) then:

\[
Cur\_DIFF(p) = ER(p) - \sum_{A_i \in SR^p} Cred_i
\]

The global goal of the agents in DCOP\_MST is to cover all the area according to \( ER \)
(i.e. to reduce the largest value of \( Cur\_DIFF \) to zero). Since this goal cannot always
be achieved, we define a more general goal which is to minimize the largest value of
the \( Cur\_DIFF \) function over all points in the area.

A set \( E \) includes all types of events. Each event can have an influence on the credi-
bility of the agents involved in the event and/or on the function \( ER \). A reputation model
is used to define the influence for each event \( e_j \in E \) on the credibility of the agents
involved. An ordered set \( OE \) includes the events which occur according to their chrono-
logical order. Each member of the ordered set \( OE \) includes the type of event, the time
of occurrence (iteration index), and its location (in case of an environmental event) or
the agents involved (in case of an event which affects agents’ credibility).

As in the standard DCOP model, each agent can send a message to each of the
other agents. We assume that each agent is aware of the current position and credi-
bility of each of the other agents. As in standard DCOPs neighboring agents are the
agents that can be influenced by an assignment change (e.g. constrained agents). Thus,
in DCOP\_MST, two agents are considered neighbors, if after they both move towards
each other, their sensing ranges overlap. Formally, the set of neighbors of agent \( A_i \) is
denoted by \( \text{cur\_nei}_i \). An agent \( A_j \) is included in \( \text{cur\_nei}_i \) iff the distance between
\( \text{cur\_pos}_i \) and \( \text{cur\_pos}_j \) is less than \( MR_i + MR_j + SR_i + SR_j \). Like in the case of
the domains, since agents change their current position, the meaning of this definition
is that the set of an agent’s neighbors is dynamic.

4 MGM for DCOP\_MST

The general design of the state of the art local search algorithms for DCOPs is syn-
chronous. In each step of the algorithm an agent sends its assignment to all its neighbors
in the constraint network and receives the assignment of all its neighbors. The MGM

\(^1\) An alternative definition would be that all the possible positions are included in an agent’s
domain but it only considers the ones with in its mobility range. However these two definitions
are equivalent.
algorithm is a simpler version of the DBA algorithm [8, 9]. In every synchronous step, each agent sends its current value assignment to its neighbors and collects their current value assignments. After receiving the assignments of all its neighbors, the agent computes the maximal improvement (reduction in cost) to its local state it can achieve by replacing its assignment and sends this proposed reduction to its neighbors. After collecting the proposed reductions from its neighbors, an agent changes its assignment only if its proposed reduction is greater than the reductions proposed by all of its neighbors.

The adjustments required to apply MGM to solve DCOP_MSTs are as follow: First, as a self adjusting algorithm, the algorithm should run infinitely, i.e. after the algorithm converges to a solution it remains active in order to be sensitive to changes [2]. Second, the most delicate matter is the definition of the quality of each of the positions an agent can reach, so that it would serve the global goal. The global goal, as defined in Section 3 is to minimize the largest value of the $Cur_{DIFF}$ function. The selection of the agents’ positions must serve this goal. An immediate trivial choice would be a position which covers the point with the highest $Cur_{DIFF}$. However, in case there are a number of positions which enable coverage of this point, we would expect the agent to choose the most effective one, i.e., the position which enables coverage of additional points with a smaller $Cur_{DIFF}$. Therefore, an agent selects its position according to the following recursive method (its code is presented in Figure 1, method select_pos):

1. Each time the recursive method is called it is given a set of possible positions and a function that defines a value to each point in the sensing range of all the possible positions. In the first call, the set will include all the positions within the agent’s mobility range $MR_{self}$ and the function $Temp_{DIFF}$ which is the current difference function without the current coverage of the agent performing the calculation ($A_{self}$). Formally, $Temp_{DIFF}$ is defined as follow:

   - For each point not currently covered by $A_{self}$, $Temp_{DIFF} = CUR_{DIFF}$.
   - For each point currently covered by $A_{self}$, $Temp_{DIFF} = CUR_{DIFF} + Cred_{self}$.

2. Next, a set ($target\_set$) which holds the points with the largest function value in the sensing range of all of the agent’s possible positions is generated.

3. Two termination conditions are checked:

   (a) If there is only one possible position, then it is selected.
   (b) If the largest function value is equal to zero (i.e. the $target\_set$ is empty). In this case any possible position can be selected.

4. If none of the termination conditions is met, the agent recalls the recursive method. The new set of possible positions which is passed in the recursive call includes all the positions in the current set of possible positions which are within the sensing range of all points in the target set. The function that is passed to the recursive method is the current function, only without the values of the points in the area which is within the sensing range of all positions in the new generated possible positions set. In other words, only areas which are not covered by the agent from each of the possible positions need to be considered.
MGM_MST
1. cur_pos ← Selected_init_pos()
2. while (true)
3. send cur_pos to each Ai ∈ cur_neiself
4. collect positions of each Ai ∈ cur_neiself
5. LR ← BestPossibleLocalReduction()
6. Send LR to each Ai ∈ cur_neiself
7. Collect LRs from each Ai ∈ cur_neiself
8. if (LR > 0)
9. if (LR > LRs of each Ai ∈ cur_neiself (ties broken using indexes))
10. cur_pos ← the position that gives LR
11. possible_pos ← positions within MR_self from cur_pos
12. Temp_Diff ← Cur_Diff \ self_coverage
13. new_pos ← select_pos(possible_pos, Temp_Diff)
14. cur_cov ← highest Temp_Diff among points within SR_self from cur_pos and not within SR_self from new_pos
15. new_cov ← highest Temp_Diff among points not within SR_self from cur_pos and within SR_self from new_pos
16. return min(cur_cov − new_cov, Cred_self)

select_pos(pos_set, func)
17. if (||pos_set|| = 1)
18. return pos_set.content
19. target_set ← points within SR_self from some pos ∈ pos_set with largest func value (must be larger than zero)
20. if (target_set is empty)
21. return some pos ∈ pos_set
22. if (no pos ∈ pos_set is within SR_self from all the points in target_set)
23. target_set ← largest subset of target_set within SR_self from some pos ∈ pos_set
24. possible_pos ← all positions in pos_set which are within SR_self from all points in target_set
25. intersect_area ← area within SR_self from all pos ∈ possible_pos
26. new_func ← func \ func.intersect_area
27. return select_pos(possible_pos, new_func)

Fig. 1. MGM_MST.

Figure 1 presents the code of the MGM_MST algorithm. The main loop of the algorithm remains almost unchanged from standard MGM [6, 7] (the standard algorithm was left out for lack of space). The agents send their assignments (current positions) to
the agents which are currently their neighbors. Notice that according to the assumptions in Section 3 the agent sends the accurate position \( ^2 \).

Method **BestPossibleLocalReduction** calls method **select_pos** to find the best alternative position. After it is found, the method returns the improvement that would be achieved by changing to the selected alternative position. This improvement (or "reduction") is the difference between the highest \( \text{Cur}\_\text{Diff} \) values, not including the credibility variable of \( A_{\text{self}} \) \( (\text{Temp}\_\text{DIFF}) \), which are covered by the agent when it is located in one of the two positions (the current and the new) and uncovered when it is located in the other (lines 13 - 15). The possible improvement can’t be larger than the agent’s credibility variable, \( C_{\text{cred}\_\text{self}} \), since that is the agent’s maximal contribution to the coverage of any point in the area (line 16).

Method **select_pos** is a recursive method that is first called with the set of all positions within the mobility range of the agent and function \( \text{Temp}\_\text{DIFF} \). First, the two termination conditions (as described above) are checked (lines 17 - 21). In the process, a set called **target_set** is generated which includes all the points with the (same) highest value of the received function \( \text{func} \), which are covered from at least one of the positions in the received set of possible positions (line 19). If all points in **target_set** cannot be covered from a single position, then only the largest subset of **target_set** which can be covered from a single position is left in **target_set** (lines 22,23). Next, a set is generated which includes all the positions in **pos_set** which enable coverage of all the points in **target_set** (line 24). In addition, a new function is generated which is equal to \( \text{func} \) except for the values of the points in the area which is covered from all the positions in the new generated set (intersection area) (line 25,26). Finlay, the recursive method is called with the new generated set and function.

### 4.1 Exploration methods

Classic local search combines exploitation methods in order to converge to local optima and exploration methods in order to escape them [10]. The MGM\_MST algorithm is strictly exploitive (monotone). While it benefits from quick convergence and avoids costly moves by the sensors, once a target is beyond the agent’s range it remains uncovered. Algorithms which implement exploration methods were proposed for standard DCOPs [6, 9, 7, 11]. However, some of the methods which are most effective in standard DCOP are not expected to be effective for DCOP\_MST [12].

In order to explore the area for new targets while maintaining coverage of targets which were previously detected we proposed two simple but powerful exploration methods which can be combined with the MGM\_MST algorithm. These two methods change the parameters of the algorithm temporarily in order to escape local minima. This approach was found successful for local search in DisCSPs [1].

1. **MGM\_WR\_MST** simply allows an agent to consider points within a larger (double) range than their \( MR \) for a small number of iterations (WR represents Wide Range).

\( ^2 \) This is a reasonable assumption considering that GPSs are used. We assume that the technology allows an agent to detect the agents which their ranges overlap with its own as defined in Section 3 and update its set of current neighbors. If not, agents would need to inform all other agents when they change position so they can update their set of neighbors accordingly.
This method assumes that a wider range is possible even though it is slower. Therefore the agents consider a wider range only in a small percentage of the algorithm’s iterations which repeat periodically (in our experiments for example, we allowed two iterations of a wider, double, range every ten regular iterations).

2. The MGM_RC_MST algorithm allows agents in some iterations to move to a position which results in an increase of the $Cur\_Diff$ function up to some number $c$. More specifically, line 8 of the algorithm is changed in these iterations to:

8. if $(LR + c > 0)$

Again, this reduced condition (RC) is only temporary and is applied periodically. This would mean that for a small number of iterations the importance (coverage requirement) of targets in the area is reduced.

In both of these methods, agents are not expected to leave targets with high importance in order to search for new targets. In MGM_WR_MST it is obvious since like in the case of MGM_MST, only moves which result with a gain are performed. In the case of MGM_RC_MST, the $c$ parameter defines the reduced importance of the targets which are already covered. Thus, $c$ is a bound on the increase to the $Cur\_Diff$ function that the method can create.

In [12], the exploration methods described above were compared with standard MGM and two classic exploration methods adjusted to DCOP_MST, DSA and DBA.

5 Search and Detection team in DCOP_MST

The first step in including a search and detection team in DCOP_MST is relaxing the assumption that there exists an ER function which includes the accurate importance of every point in the area. Instead, we assume that the function initially includes some distribution which reflects the probability of the existence of targets in the area. This assumption makes the model compatible for any level of uncertainty from complete entropy (as in [5]) to complete knowledge (as in [12]).

This initial ER function is copied to another (initially identical) function we refer to as the search map (SM). Search agents use the search map SM when they decide on their path in order to detect targets and they update the ER function with the targets they find. The surveillance agents use only the ER function, in the same way as described in Section 4. Thus, the ER function is the device used for communication between the two teams.

The search agents use the SM function to communicate to each other where they have recently visited and therefore the probability of the existence of a new target is low. This is done as follow:

1. The base value of a point in the SM function, $Base_p$, is equal to the value in the initial ER function. Thus initially, all points in the SM function are equal to their base value.

2. A search agent $sa$ located in some point $p$ at iteration $t$, causes a decrease in the value of all points $p'$ within the sensor range of $p$. The new value of these points is $SM(p',t) = Base_p - Cred_{sa}$
DSA_SAT
1. cur_pos_self ← select_init_pos()
2. while (true)
3. foreach (point p within SR from cur_pos_self)
4. SM(p) ← Base_p − Cred_self
5. ER(p) ← importance(p)
6. next_pos ← get_best_pos()
7. rand ← random([0, 1])
8. if (rand < prob)
9. cur_pos_self ← next_pos

get_best_pos()
10. possible_pos ← positions within MR_self from cur_pos
11. max_val ← 0
12. foreach pos ∈ possible_pos
13. psr ← p within SR from pos
14. if (∑p∈psr SM(p) < max_val)
15. next_pos ← pos
16. max_val ← ∑p∈psr SM(p)

Fig. 2. DSA_SAT.

3. At each iteration t in which there is no search agent in sensing range from point p, the SM value of p is incremented as follow:
\[ SM(p, t) = \min\{SM(p, t-1) + 1, Base_p\}. \]

Figure 2 presents the adjusted DSA algorithm for a team of search agents in DCOP_MST, DSA_SAT. In each iteration of the algorithm the SM value of all points within sensor range of the agent are set to a lower value according to the agent’s credibility (lines 3,4). In addition, the agent updates the ER function with the true importance of the points in its sensing range. Then, the agent selects the best position it can move to by calling function get_best_pos(). The agent moves to this position in probability prob (lines 7,8).

Function get_best_pos() selects the point within mobility range, which the sum of the SM values of points within sensing range from it, is maximal.

6 Cooperation across teams

In the previous sections we described two teams performing in the same area, each with its own task and means for communication between them via the ER function. However, it is reasonable to assume that cooperation among agents from the two teams can lead to better results for the following reasons:

1. Although each team of agents has its own task, they are both working towards a common goal.
2. Agents’ efficiency depends on their location. Therefore, it may be the case that an agent from one team is in a position that allows her to serve the task of the other team best. Common practice in multi agent systems include hierarchical plan structures that allow agents to assist others when working towards a common goal [3, 4]. Specifically to the applications at hand, we assume that search agents have superior technology and therefore they can perform surveillance with relatively high credibility while surveillance agents cannot define the importance of a target. Thus, we describe the following possible collaborations among the two teams:

1. Search support (SS): search agents take an active role in the surveillance of targets within their sensing range.
2. Alert: agents from the surveillance team increase the value of points in the search map where they suspect their might be a target. Thus, search agents are encouraged to search at these locations.
3. Avoid Abandoning (AA): search agents do not move to a new location when they are covering targets which are not reasonably covered by surveillance agents, i.e., search agents which locate a target wait for it to be covered by surveillance agents before they continue their search.

All three modes of cooperation described above require agents to be aware of the task of the other team. The alert and AA modes further require that agents communicate with agents in the other team via either the ER or the SM functions.

7 Experimental evaluation

In order to evaluate the performance of the two teams of agents and the effect of the proposed collaborations, we use the same simulator used in [12]. The problem simulated is of an area in which the possible positions are points on a grid. The size of the grid in all our experiments was 150 by 150. Each of the points in the area has an importance value between 0 and 100. The ER function initially had all points equal to 0. The credibility of search agents was set to 50 and the credibility of surveillance agents was set to 30. The sensing ranges of all agents were set to 10. The mobility range of search agents was set to 30 while the mobility range of surveillance agents was set to 10. The algorithm performed by the surveillance agents was MGM_WR_MST and the DSA_SAT was performed by the search agents. The area included 20 targets which are points with importance 100. We assume that these targets are revealed to search agents which are located within sensing range from them. Surveillance agents can perform their task only on targets which were detected by search agents. However, a target which has not yet been detected by a search agent raises the suspicion of a surveillance agent. All our experiments included 10 search agents and 60 surveillance agents. Each reported result is an average over 50 runs of the algorithm on different random generated problems. The random elements in each problem were the location of the targets and the initial location of the agents.

In the first set of experiments we evaluated the success of our proposed search agent algorithm DSA_SAT with respect to the value of the prob parameter. Figure 3 presents
the number of targets detected by the search agents as a function of the number of iterations performed since the search started. The different lines represent different $prob$ values used by the search agents in the DSA.SAT algorithm. It is clear that the algorithm is most successful for high values of $prob$. In contrast to standard DSA, high
level of exploration does not cause thrashing. This is because the search agents are not required to converge to a solution like in standard DCOP but rather keep on searching for additional targets. However, for large prob values (e.g., 0.7 and 0.9), there is no notable difference in performance. Figure 4 presents the same phenomenon when Alert cooperation mode is used. This figure demonstrates the benefit of using this mode of communication between the surveillance agents and the search agents for faster detection of targets.

Figure 5 presents the success of the two teams when performing together in order to reduce the difference of the sum of the ER values of all targets in the area and the credibility of agents which are performing surveillance in sensing range from the targets. In this set of experiments the initial ER function included all targets. Thus, there was no need to perform search. The results in Figure 5 demonstrate the effect of the different levels of communication on the performance of the surveillance team. It is clear from the result that when the search agents participate in the surveillance procedure (SS mode) but are not committed to it, the improvement in performance is minor. However, when the search agents are aware of the level of coverage on targets found and leave targets only if they are reasonably covered (AA mode), the performance in terms of surveillance substantially improves.

Figure 6 presents results of a complete experiment, i.e., target locations are not known in advance and the results are in terms of surveillance coverage (both sub-teams need to perform their sub-task). The results presented demonstrate how each additional level of cooperation among the two sub-teams improves the overall performance of the entire global team of sensing agents. The effect of the Alert mode is more apparent in the first iterations when surveillance agents are waiting for targets to be discovered,
while the AA mode triggers the most substantial improvement in coverage. To emphasize the difference in performance, we focus on Figure 7 and see that after 25 iterations, in the highest level of cooperation, the coverage of the team is improved by a factor larger than two.
8 Summary and Conclusions

Recently, DCOP_MST, a new model based on DCOP for solving dynamic problems, was proposed. It represents a dynamic application of a team of mobile sensing agents which is expected to be robust to changes in the environment in which the sensors operate, changes in the teams tasks and technology failures. The early work, which introduced the DCOP_MST model [12], assumed that agents are aware of the location and importance of targets in the area.

In this work we adjusted the model to include two sub-teams of agents performing towards a common goal. Each team had its own sub-task which serves the common goal. The first team included agents with high mobility and accurate sensing technology. The subtask of this team was detection of targets. The second team’s task was to perform surveillance on the targets which were detected by the first team. The search agents used an additional function, a search map (SM) to represent locations in which there is a higher probability to find targets. This SM was dynamically updated according to the search agents’ movements. Thus, it was also used as a means of coordination among the search agents. The algorithm used by the search agents was based on DSA and our results demonstrated that it was most successful with high level of exploration and concurrency. The information regarding targets that were detected and their importance was transferred to the surveillance agents via the ER function.

Our results demonstrate the success of the DCOP_MST model in representing teams with different tasks. Furthermore, they demonstrate how increased levels of cooperation among the teams improve the performance of both sub-teams in fulfilling their sub-tasks and the result of the global team in achieving the common goal. Clearly, the flexibility of the model in representing and coping with dynamic elements, allows the teams to effectively handle additional information supplied to them by agents from the other team. The most successful results were achieved when all the proposed forms of cooperation were used.

Acknowledgment: This research has been supported by AFOSR FA9550-08-1-0356.

References


Towards a Theoretic Understanding of DCEE

Scott Alfeld, Matthew E. Taylor, Prateek Tandon, and Milind Tambe

http://teamcore.usc.edu/

The University of Southern California

Abstract. Common wisdom says that the greater the level of teamwork, the higher the performance of the team. In teams of cooperative autonomous agents, working together rather than independently can increase the team reward. However, recent results show that in uncertain environments, increasing the level of teamwork can actually decrease overall performance. Coined the team uncertainty penalty, this phenomenon has been shown empirically in simulation, but the underlying mathematics are not yet understood. By understanding the mathematics, we could develop algorithms that reduce or eliminate this penalty of increased teamwork.

In this paper we investigate the team uncertainty penalty on two fronts. First, we provide results of robots exhibiting the same behavior seen in simulations. Second, we present a mathematical foundation by which to analyze the phenomenon. Using this model, we present findings indicating that the team uncertainty penalty is inherent to the level of teamwork allowed, rather than to specific algorithms.

1 Introduction

Recently there has been a rise in autonomous teams of agents working cooperatively. Teams of artificial agents, whether software or robotic are being deployed today in a variety of scenarios, including mobile sensor networks (Cheng et al., 2005; Marden et al., 2007; Jain et al., 2009), and teams of underwater autonomous scouts (Zhang et al., 2005). Many problems are universal across multiagent systems. Even in fully observable environments, centralized solutions scale poorly with the number of agents. Often centralization is not an option, and agents must behave partially independently while still working with each other to facilitate cooperation. The underlying understanding of these issues is that the higher the level of teamwork the better the outcome.

Recently, however, the question of if agents should work together has arisen. As such, we do not concern ourselves with issues of communication in this paper, but instead focus on when and if the level of teamwork amongst agents should be increased. We recently introduced the notion of the team uncertainty penalty (Taylor et al., 2010).

Put informally, this result shows that agents should sometimes act alone, rather than attempt to coordinate, even if communication is free. Previous work points to the density of a graph as the leading culprit, but the underlying cause remains a mystery.

1 This paper focuses on cooperative multi-agent problems where all agents may be considered part of a single team as they share a common reward function. However, we use the term “level of teamwork” to refer the amount of partial centralization among agents, reflected in how much information they share, how they coordinate actions, and how many agents may simultaneously perform a joint action. More precisely, higher level of teamwork will refer to higher values of $k$ in the $k$-optimal and $k$-dependent algorithms that follow.
Empirical evidence has shown manifestations of the team uncertainty penalty in simulation (Taylor et al., 2010). While analysis of these results suggest when the phenomenon will be observed, there has not yet been a way to determine \textit{a priori} when the team reward will or will not suffer from increasing agent cooperation. Outside of simulation, the extent of the team uncertainty has not yet been studied and it is unknown how debilitating it can be in robotic applications. Moreover, as of yet there is no mathematical framework under which the specifics of the phenomenon can be analyzed.

This paper provides experimental evidence of robot teams exhibiting similar results as those found in simulation. We also provide a mathematical analysis used to help pinpoint the contributing factors of this counter-intuitive phenomenon. Additionally, in Section 5.3, we provide evidence showing that the team uncertainty penalty is a part of the DCEE framework rather than an artifact of specific algorithms. We introduce the notion of $L$-Movement and analyze that ramifications of restricting the allowed total movement of an agent team. We also introduce the \textit{configuration hypercube} for specific problem formulations and propose that analysis on the probabilistic structure of this hypercube will lend key insights into the behavior of previously explored algorithms. In particular, we suggest that analysis of the hypercube will provide a heuristic by which one can predict the effects of increasing the level of teamwork for a given problem.

2 Background

The \textit{Distributed Constraint Satisfaction Problem} (DCOP) (Mailler & Lesser, 2004; Modi et al., 2005; Petcu & Faltings, 2005) framework is becoming common for modeling cooperative multiagent scenarios. Because of its ability to model the need for cooperation via the notion of joint rewards, the formulation is being used in several problems, such modeling sensor networks and for multiagent plan coordination (Cox et al., 2005). Solving a DCOP (determining a globally optimal configuration) is NP-Hard (Modi et al., 2005), and thus there is substantial work towards finding fast, locally optimal algorithms (Maheswaran et al., 2004; Pearce et al., 2008; Stranders et al., 2009).

In real world applications, the rewards are not known \textit{a priori}, and discovering them requires exploration. Moreover, agents are concerned with a total, online reward achievable in a limited time frame. We present one example of such an application in Section 4. When working in such environments, agents must strike a balance between exploiting their current knowledge and exploring their environment. While exploration vs. exploitation is a common problem for single agents (c.f., reinforcement learning (Sutton & Barto, 1998) and multi-armed bandits (Robbins, 1952)), having a multiagent system where rewards are determined by the coordination of agents adds a level of complexity. The DCOP framework, however, assumes that all aspects of the environment are known. Allowing uncertainty to be modeled, the notion of a \textit{Distributed Coordination of Exploration and Exploitation} (DCEE) problem was recently introduced (Jain et al., 2009). This extension of DCOP to real world environments accounts for the uncertainty of rewards, as well incorporating the notion of a limited time horizon. For reference, we describe DCEE here.

A DCEE domain is much like a DCOP, with several key differences. Specifically, the rewards in the constraint matrices are not known until they are explored. That is to say that the reward achieved by agents $A_i$ and $A_j$ when they assign their variables to
\[ \lambda_m \text{ and } \lambda_n \] respectively is not known until a point that \( A_i \) assigns value \( \lambda_m \) and \( A_j \) simultaneously assigns value \( \lambda_n \). What is known \textit{a priori}, however, is something about the distribution over rewards for each constraint. In this paper we assume the underlying distribution of rewards is known.

A DCEE consists of a set \( Q \) of \( n \) variables, \( \{x_1, x_2, \ldots, x_n\} \), assigned to a set of agents, where each agent controls one (or more) variable’s assignment. For this paper we concern ourselves with the case when every agent has only one variable. Agents have at most \( T \) rounds to modify their variables \( x_i \), which can take on any value from the finite domain \( D_i \). The goal of such a problem is for agents to choose values for the variables such that the cumulative sum over a set of binary constraints and associated payoff or reward functions, \( f_{ij} : D_i \times D_j \rightarrow \mathbb{R} \), is maximized over time horizon \( T \in \mathbb{N} \). More specifically, the agents attempt to pick a set of assignments (one per time step: \( A_0, \ldots, A_T \)) such that the total reward (the \textit{return}) is maximized: \[ R = \sum_{t=0}^{T} \sum_{x_i, x_j \in V} f_{ij}(d_i, d_j), \text{ where } d_i \in D_i, d_j \in D_j \text{ and } x_i \leftarrow d_i, x_j \leftarrow d_j \in A_k. \]

The following is a list of select properties that DCEEs have, but DCOPs do not: (1) agents initially know the constraint graph but only discover rewards through exploration (i.e., a pair of agents set their values to explicitly discover a reward), (2) problems last a set amount of time, (3) there are more combinations of domain values than can be explored within this time (disallowing exhaustive exploration), and (4) we seek to maximize the online global reward for the team over this time horizon \( T \).

A DCEE can be thought of as being defined in part by a graph \( G = \{V, E\} \). In this framework, every variable corresponds to a vertex in \( V \), and every constraint is an edge in \( E \). Every edge is thus a two dimensional matrix where every element has been drawn i.i.d. from a known distribution. Throughout this paper we refer to an agent changing its variable’s value as that agent \textit{moving}.

### 2.1 k-Optimal

Centralized algorithms for DCOPs scale poorly, as finding a globally optimal reward is NP-Hard (Modi et al., 2005). Because of this, approximate DCOP algorithms are heavily worked investigated (Zhang et al., 2003; Pearce & Tambe, 2007). The notion of \( k \)-optimal configurations expresses the level of a locally optimal solution. The lower the value of \( k \), the more local the \( k \)-optimal solution.

Algorithms previously presented (Taylor et al., 2010) come in different variants based on the value of \( k \). This value indicates the maximum size of a coalition of agents that can move at each step. In a DCOP, a \( k \)-optimal configuration is defined as one where no coalition of up to \( k \) agents would benefit by changing their variable. The notion of \( k \)-optimal does not extend directly to DCEE, as we discuss in Section 5.1.

### 2.2 SE-Optimistic

The DCEE algorithm SE-Optimistic-1, where \( k = 1 \), will behave as follows. A group of agents is considered to be in a neighborhood if the subgraph created by those agents form a connected graph. In each round, every agent calculate their potential gain (how much they could improve their reward) obtained by moving under the assumption that all other agents in their neighborhood will not move. Each agent also queries each of its neighbors for their potential gain. If an agent has a higher potential gain than any of its neighbors in a given round, it will move so as to obtain its maximum gain. In
SE-Optimistic-2, where $k = 2$, a pair of agents (in the same neighborhood) will move if their total potential gain is higher than any other pair of agents in their neighborhood. Intuitively, these algorithms perform a greedy search. In each round, the agents who would most benefit by moving alone do so, while others in the same neighborhood do not move. $k = 2$ algorithms allow more agents to attempt to improve their individual rewards in each round, while in $k = 1$ algorithms only one agent per neighborhood may move per round.

### 2.3 MGM-Omniscient

Omniscient algorithms (Taylor et al., 2010) are artificially provided the values of all rewards of joint actions and thus do not need to explore. This information, causing the algorithms to be “omniscient” in regards to the rewards, modifies a DCEE into a DCOP. The omniscient DCEE algorithms referred to in this paper, namely omni-1 and omni-2, are previously introduced algorithms run on this resulting DCOP. In omni-1, in each round every agent calculates how much it could improve its own reward by being the one in its neighborhood to move. After agents communicate these values amongst others in their neighborhood, the agent who can increase their reward the most moves, and others in the same neighborhood do not. Omni-2 is similar, but allows for agents to form coalitions of two agents in the same neighborhood, and both may move in a single round. These algorithms, called Maximum Gain Message (MGM), are described in detail elsewhere (Pearce & Tambe, 2007).

### 3 Team Uncertainty Penalty

Intuitively, it seems that the more teamwork amongst truthful, cooperative agents the better the overall performance of the team. Increasing the level of teamwork among agents will add to communication overheads and computational costs, but in this work, we ignore both communication and computational costs. Instead, the team uncertainty penalty has to do with decreased total reward (in some circumstances) when teamwork is increased (Taylor et al., 2010).

The prevalence of the team uncertainty penalty has been shown empirically. It has been observed in varying graph topology, and across several algorithms. In Section 5.3 we present experimental results of a more mathematical nature that suggest the team uncertainty penalty in even more general, being an artifact of the level of teamwork rather than the specifics of the algorithms.

Before discussing the penalty in more depth, we first outline the domain used and then discuss results in the domain exhibiting this counter-intuitive behavior.

### 3.1 Simulator Domain

The DCEE mobile wireless network problem was first introduced elsewhere (Jain et al., 2009). To explore this problem, we have built and released a simulator\(^2\) that models a set of mobile robots with wi-fi antennas that must optimize the sum of the inter-agent signal strengths. Value settings correspond to agent (robot) locations, constraints are determined by the network topology, and rewards are based on link quality between robots. As in prior work, we assume that the topology is fixed, that small changes to

---

\(^2\) Available at [http://teamcore.usc.edu/dcop](http://teamcore.usc.edu/dcop).
agent locations causes signal strengths for that agent to become uncorrelated with their previous values (i.e., small scale fading dominates (Molisch, 2005)), and that signal strengths between agents are symmetric.

3.2 Simulator Results
Previous results (Jain et al., 2009; Taylor et al., 2010) demonstrated that multiple DCEE algorithms were effective at improving the online reward in the simulated domain over multiple numbers of agents, experiment lengths, and network topologies. To quantify performance, we consider the total gain of different algorithms, where the gain on round $n$ is defined as the difference between the team’s reward on round $n$ and the team’s reward on round 0.

Figure 1 shows the total gains of algorithms on graphs of 10 agents run for 50 rounds each (averaged over 10 independent trials). First, consider the Omniscient algorithms (Omni, Omni-2, and Omni-3). In the left half of the graph, we see that as the amount of teamwork (i.e., $k$) in the Omniscient algorithm increases, the total reward during experiments on chain graphs increases. The right half of the graph shows the same trend for complete graphs: as $k$ increases, the Omniscient algorithm receives more reward.

Results of the Optimistic algorithms in Figure 1 tell a different story. While increasing $k$ again improves the gain for complete graphs, higher values of $k$ decrease the reward gained in chain graphs (ring graphs with one fewer edge). This is precisely what we mean by the team uncertainty penalty: increasing levels of cooperation may decrease the team reward (relative to lower levels of teamwork). This phenomena is further analyzed and discussed elsewhere (Taylor et al., 2010), but strictly on an empirical basis.

4 Robot Experiments
This section provides novel experimental results on physical robots, corroborating the results from simulation discussed in the previous section. Previous results (Taylor et al., 2010) come from simulations of DCEE. Here we provide evidence that the DCEE formulation holds true to real world scenarios, specifically in the case of mobile robots connected via a wireless network. Moreover, we demonstrate the team uncertainty penalty manifesting on actual hardware.

4.1 Problem Setup
Robots in this section run DCEE algorithms in order to maximize the signal strength between wireless network receivers, analogous to the simulation described in Section 3.
An agent can measure the wireless signal strength between itself and each neighbor, corresponding to measuring the reward for the pair of agent assignments. Agents select from a set of possible physical locations (i.e., a variable assignment). Because the time for movement in physical robots dominates communication and calculation time (Jain et al., 2009), we measure experiment length by the number of rounds, defined by the period in which every agent may decide to move to a new position and then reach that position. All agents may choose to either stay in their current position or explore.

The Create, made by iRobot, is used as the platform for our experiments (see Figure 2). Additionally, an ebox-3854 is used as an on-board computer and the EMP-8602 mini PCI card is used for 802.11 b communication between the robots. More details can be found at http://enl.usc.edu/projects/peg/platform.html.

4.2 Results

This section discusses the results of executing the SE-Optimistic-1 and SE-Optimistic-2 algorithms on physical robots. To corroborate the results in simulation, the two algorithms are tested on both chain and complete graphs of five agents each. Figure 3 shows the average gain per round of the four different setups. The plots average ten trials and error bars show the standard error. In all cases, the algorithms improve the reward of the team, although SE-Optimistic-1 on the complete graph improves much more slowly.

One possible reason for this discrepancy is that the gain achieved by the agents is dependent on the starting configuration — the worse the starting configuration, the more latitude exists for achievement. Unfortunately, the physical agents cannot be returned to exactly the same start state, and thus different trials have different initial signal strengths. The average team initial reward for the complete graphs when run using SE-Optimistic-1 was $574 \pm 55$, whereas the average for SE-Optimistic-2 was $506 \pm 31$, thus SE-Optimistic-1 has a relatively harder time improving the team’s reward.

Figure 4(a) displays the total gain (i.e., the area under the curves in Figure 3). We again see that SE-Optimistic-1 on a complete graph performs worse than all other algorithms. Most important is that the trends from section 3 are seen again. In a chain graph, $k = 1$ outperforms $k = 2$, and in a complete graph, $k = 2$ outperforms $k = 1$. As such, these experiments on five robots confirm trends predicted in simulations of 10–50 virtual agents.

5 Analysis of the Team Uncertainty Penalty

This section provides a mathematical foundation for analyzing the team uncertainty penalty. We extend the notion of a $k$-optimal configuration in a DCOP to the DCEE formulation. We also introduce the notion of $L$-movement which expresses the flexibility
Fig. 3. This plot shows performance of SE-Optimistic on two different graph topologies for \( k = 1 \) and \( k = 2 \). The x-axis shows the round number and the y-axis shows the gain. Results are averaged over 10 trials, each using 5 agents. Standard errors are shown only on \( k = 1 \) plots for readability.

in exploration of a given DCEE algorithm. Finally, we present a mathematical model of DCEE problems based on the interplay of random variables.

Previously introduced algorithms such as SE-Optimistic-1 and SE-Optimistic-2 have behavior defined by the value \( k \), controlling the number of agents that cooperate in each round. We refer to this class of algorithms as \( k \)-dependent. Previous work points to graph density as a strong contributing factor (Taylor et al., 2010), and we extend this analysis by introducing the notion of \( L \)-movement, which ties directly to graph density. In addition, we provide simulation results showing the effects of changing the value of \( k \) in \( k \)-dependent algorithms.

We hope that the DCEE formulation will eventually be understood fully on a theoretic level. This section provides a foundation for such an understanding, as well as some initial results from the proposed mathematical model. We show that the team uncertainty penalty may not be the fault of the individual algorithms, and provide evidence that the phenomenon is an inexorably tied to DCEE formulation and algorithms’ dependence on \( k \).

5.1 Extending \( k \)-Optimality and Introducing \( L \)-Movement

In a DCOP, a configuration is defined as \( k \)-optimal if no connected group of at most \( k \) agents would benefit by jointly moving (Maheswaran et al., 2004; Pearce et al., 2008). In a DCEE, however, the rewards are not known, and thus an agent (or group of agents) cannot know whether or not they would increase their reward by changing their variable(s). Therefore the notion of \( k \)-optimality does not extend directly from DCOP to DCEE. However, if the reward distribution(s) are known in a DCEE, a natural extension of the \( k \)-optimal definition is that a DCEE configuration is \( k \)-optimal if and only if no connected group of up to \( k \) agents has a positive expected gain by choosing an entirely new configuration. However, this does not allow for the possibility of agents reverting to a previously seen (partial-)configuration.
Fig. 4. The chart in (a) shows that the total gain in robot tests follow the trends from simulation. SE-Optimistic-1 outperforms SE-Optimistic-2 on chain graphs, but underperforms it on complete graphs. Standard error over the 10 trials are shown for each setting. (b) depicts two topologies used throughout the paper: a five node ring graph (top), and a five node complete graph (bottom).

In a starting configuration of a DCEE, where only one configuration (the current one) has been explored, this definition is reasonable, and connects well with the notion of $k$-optimality of a DCOP configuration. After even one round, however, groups of agents have the option to jointly move back to a configuration where some or all of the constraint reward values have been observed. As such, the $k$-optimality of a configuration would depend on the set of configurations already observed. We therefore provide a definition of $k$-optimal in DCEE as follows.

**Definition 1.** A DCEE configuration, combined with the history of explored configurations, is $k$-optimal if and only if no group of up to $k$ agents has a positive expected gain by selecting another configuration with zero or more constraint values already explored.

This says a configuration is only $k$-optimal if no coalition of up to $k$ agents could move so as to increase their total reward by each agent either (i) selecting previously explored values and obtaining the known rewards, or (ii) selecting unexplored actions leading to a positive expected gain in total reward. Because of the dependence on explored and unexplored configurations, the set of potentially $k$-optimal configurations of a DCEE is extremely large. Consider the following example.

**Example 1.** A DCEE with two agents $A_1$ and $A_2$ and one constraint. When each variable takes on the first value, the first reward is known – suppose it is 120. $A_1$ then moves, selecting a new value, and the constraint yields a reward of 110. At this point two rewards are known. Suppose either $A_1$ or $A_2$ can select an action that will yield a reward of 110 with probability .5, and a reward of 100 with probability .5. Selecting a new action would yield an expected gain of $110 - 105 = -5$. However, if $A_1$ reverts to its previous action it can increase the total reward by $120 - 110 = 10$. Thus this configuration, combined with the history of explored configurations, is not 1-optimal.

We now focus on a different aspect of $k$-dependent algorithms. First we introduce $L$-Movement, a value we will use heavily in our analysis.
Definition 2. The L-movement, denoted as $L$, of a DCEE algorithm on a graph $G$ is the maximum number of agents that can move in any one round.

There is no restriction inherent to the DCEE formulation on the number of agents that can move at each step. For example a valid, albeit naive, algorithm would move every agent at every step. In SE-Optimistic-1 and SE-Optimistic-2, as well as other $k$-dependent algorithms, only a limited subset of agents can move at each step.

This restriction of movement is caused by the fact that an agent cannot move if more than $k - 1$ of its neighbors also move. For a given graph, we denote the maximum number of agents that can move in each round of a $k$-dependent algorithm as $L$. We note that $L$ is dependent on the graph topology, for example in $k = 1$ algorithms $L$ is the size of the largest maximal independent set of the graph, a value known to be NP-hard to calculate for a general graph. $L$-movement is central to our analysis presented in 5.3 where we show the change in $L$ caused by changing $k$ determines the change in team performance. Consider a ring graph and a complete graph, both with 5 vertices (see Figure 4(b)). In the ring graph, the size of the maximal independent set is 2, whereas in the complete graph it is only 1. In general, the size of the maximal independent set of a ring graph is $\lfloor \frac{|V|}{2} \rfloor$, and is 1 for a complete graph.

5.2 The Configuration Hypercube

For a DCEE problem, we use the following notation:

1. $G = \{V, E\}$ The graph of the DCEE.
2. $A = \{A_1, \ldots, A_{|V|}\}$ The set agents.
3. $R = \{R_1, \ldots, R_{|E|}\}$ The set of constraints.
4. $r_i$ The distribution from which the rewards of $R_i$ are drawn.
5. $T$ The number of rounds for which the algorithms are run.
6. $x_i$ The variable of the agent $A_i$.
7. $D_i$ The domain of $x_i$.
8. $C$ The configuration hypercube (defined below).
9. $\lambda_i$ The value of the $i$-th coordinate of a location in $C$.

We consider the set of all DCEE algorithms where:

- The starting configuration is initialized randomly.
- The algorithm is run for $s$ steps, ending with some configuration.
- That ending configuration is then chosen for every step for the rest of time.

This is to say that $s$ steps are allotted to exploration, after which the algorithm only exploits (keeps the same configuration) for the remaining $T - s$ rounds. We analyze the reward achieved after the $s$ steps of exploration.

To analyze such algorithms we define a $|V|$-dimensional hypercube $C$ as the Configuration Hypercube. Each dimension of $C$ corresponds to an agent, with the location within that dimension defined by the assignment of the agent’s variable. We let $C[\lambda_1, \lambda_2, \ldots, \lambda_{|V|}]$ be the total reward when agent $A_i$ takes value $\lambda_i$.

We note that, for a given agent, no value for its variable is believed a priori to be more beneficial than any other. Therefore, without loss of generality we say that each agent $A_i$ has an ordered list of values $\alpha = \alpha_1, \ldots, \alpha_{|D_i|}$, and that at each round of a DCEE algorithm, an agent has only two choices: (1) select an value previously
chosen, or (2), select the first value (the value with the lowest index) that has not yet been assigned. Further, for any DCEE algorithm, we assume (again without loss of generality) that all agents begin with their first variable setting, $a_1$.

At each round of an algorithm, zero or more agents move. Note that at any location $\lambda = (\lambda_1 \ldots \lambda_{|V|})$ in $C$, the achievable locations after one step are all locations adjacent to $\lambda$ ($l_\infty$ distance to $\lambda$ is 1). That is to say that $\lambda$ is reachable if and only if $\max_i (\lambda_i) \leq s$. Further, for a general DCEE algorithm, the locations reachable from the starting location in $s$ steps form an $(s + 1)$-sized $|V|$-dimensional hypercube $C_s$ where all locations have $l_\infty$ distance to the starting location less than or equal to $s$. The expected maximum value of $C_s$ is an upper bound for the total reward achievable by any DCEE algorithm (under conditions described above) running for $s$ rounds.

However, computing the expected maximum element of $C_s$ is non-trivial. Every entry in $C_s$ is the sum of $|E|$ values (one from each constraint), and thus the entries are highly dependent. Furthermore, while each constraint contributes $s^2$ values to $C_s$ and the expected maximum of each of these can easily be computed, $C_s$ contains only a subset of the possible sums of constraint values. The expected maximum is not simply the sum of the expected maximums of the constraint matrices. Moreover, this subset is a function of the graph structure and finding the expected maximum can be difficult.

We note that the expected maximum of $C_s$ is the expected maximum reward obtainable by an algorithm that does not need to explore. By this we mean an algorithm with perfect knowledge of the values of all locations in $C_s$, but with the limitation that configurations outside of $C_s$ are unreachable. As such, it is not a tight bound for DCEE algorithms. We illustrate this fact with the following example.

**Example 2.** Consider a DCEE with two agents $A_1$ and $A_2$, and one constraint between them. Let $M_d(i)$ denote the expected maximum of $i$ samples drawn independently from distribution $d$. The specific form of $M_d(i)$ is unimportant for our results, but the function can be computed for the distributions we use. The 2-dimensional cube $C_2$ would have $(s + 1)^2$ values, with the expected maximum being $M_d((s + 1)^2)$. An algorithm that knows all values in $C_s$ could assign values for both agents’ variables such that this reward is obtained. An algorithm without access to the individual values, however, has four choices at each step – change $A_1$’s variable, change $A_2$’s variable, change both agents’ variables, or leave the configuration unchanged. When either or both of the agents change their variable the (only) constraint is re-sampled. Thus the best an algorithm could hope to achieve is the expected maximum of $s + 1$ samples, i.e., $M_d(s + 1)$.

Achieving the maximum value of $C_s$ requires the algorithms to know more about the reward matrices than is allowed in the DCEE framework. We also note an additional relaxation. In $k$-dependent algorithms, not all $k$-sized subsets of agents can move in one round. For example, running a $k = 1$ algorithm on a five vertex ring graph has a value of $L = 2$. Our experiments take the maximum value of $C_s$ reachable by changing any two (or fewer) agents in each round. A $k$-dependent algorithm, however, could not have two adjacent agents change their variable’s value in a single round when $k = 1$.

Even with these relaxations, insights can be gained by observing what happens when we consider the portion of $C_s$ reachable when we restrict the number of agents that can move at each step. Consider the example with two agents described above as we discuss the locations in the configuration hypercube that are reachable. If zero, one,
or both agents can move at every step ($L = 2$), then all $(s + 1)^2$ locations in $C_s$ are reachable. However, if at most one agent can move in each round ($L = 1$), at every step only one (or neither) agent can move, then only $\frac{(s+1)^2}{2} + s$ locations can be reached. See Figure 5.

For a DCEE, recall that $C_s$ is defined as all locations in $C$ with $l_\infty$ distance to the starting point less than or equal to $s$. If only one agent can move per step, the reachable locations of $C_s$ instead are all locations $\lambda = (\lambda_1, \ldots, \lambda_{|V|})$ in $C$ with $l_1$ distance less than or equal to $s$ (i.e., $\sum_i(\lambda_i) \leq s$). When two or more agents can move at each step, the set of reachable locations in $C$ becomes less intuitive. We now define such sets.

In general a location $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_{|V|})$ in $C_s$ is reachable if and only if

$$\sum_{i=0}^{|V|} \lambda_i \leq Ls$$

where $L$ is the maximum number of agents that can move at a single step, as defined above. To understand this inequality, imagine having $n$ bins corresponding to the $n$ agents (see Figure 6). In each round of the algorithm, you can place up $L$ balls into these bins, with the restriction that you cannot place more than one ball in any one bin in a single round. Placing a ball in the $i$-th bin on the $j$-th round corresponds to the agent $A_i$ changing its variable on the $j$-th round. After $s$ rounds, there are two restrictions of the ending quantities in the bins: (1) the most full bin can contain at most $s$ balls, and (2) the total number of balls must be less than or equal to $Ls$. Restriction 1 says the location must be within the confines of $C_s$, and the above inequality upholds restriction 2.

5.3 $L$-Movement Experimentation

We constructed $C_s$ for $s = 1 \ldots 30$ for a complete graph and a ring graph (both containing 5 nodes). Every constraint has its rewards drawn from a Gaussian distribution with mean 100 and variance 16. For $L = 1 \ldots 5$, we calculated the maximum value of the achievable locations. Values are averaged over 50 trails.

Note that the plots for Ring graphs and Complete graphs are exceptionally similar (see Figure 7). The difference between $L = i$ and $L = i + 1$ decreases as $i$ increases in both figures – the $L = 4$ curve overlaps almost entirely with the $L = 5$ curve. Specifically, we note that the $L = 1$ curve is drastically different from the $L = 2$ curve. This indicates that $L$ is a considerable contributing factor to the team uncertainty penalty. We ran experiments on other graphs as well, and using different distributions for reward values. All results showed similar behavior in regards to changing $L$. See Figure 8.
Fig. 7. These plots show the best team reward found in each round while exploring five-agent ring and complete graph DCEEs where all rewards are drawn from a Gaussian distribution with mean 100 and standard deviation 16 for \( L = 1 \ldots 5 \). Note that the performance difference between \( L = 1 \) and \( L = 2 \) is larger than between \( L = 2 \) and \( L = 3 \). This relative performance difference decreases as \( L \) increases. Values are averaged over 50 runs, and error bars show standard error of the mean.

Let us consider the specific case of SE-Optimistic-1 and SE-Optimistic-2 running on DCEEs defined in part by these graphs (see Figure 9). In the complete graph, \( L = 1 \) for \( k = 1 \) and \( L = 2 \) for \( k = 2 \). Thus we would imagine that SE-Optimistic-2 would outperform SE-Optimistic-1 by a fair margin. In the ring graph, on the other hand, \( L = 2 \) for \( k = 1 \) and \( L = 3 \) for \( k = 2 \). This means that in the complete graph, we are potentially obtaining a large benefit from changing \( k = 1 \) to \( k = 2 \) because doing so means we go from \( L = 1 \) to \( L = 2 \). In the ring graph, however, the same change in \( k \) yields a change from \( L = 2 \) to \( L = 3 \) which has a much smaller potential performance gain. Therefore we would expect to see a smaller improvement, if any, of SE-Optimistic-2 over SE-Optimistic-1. Empirically, this is precisely what is exhibited both in simulation and on robots.

These results indicate that the \( L \)-movement, determined by the value of \( k \), may contribute to the team uncertainty penalty rather than the operations of the specific algorithms. A general algorithm with \( L = 2 \) will have the ability to explore more of \( C_s \) than an algorithm with \( L = 1 \). Note that \( L \) is strictly determined by the graph topology and \( k \), thus any \( k \)-dependent algorithm will experience diminishing returns as \( k \) increases. It is worth noting that the \( L \)-movement difference between \( k = 1 \) algorithms and \( k = 2 \) algorithms will only be higher in larger ring and complete graphs. Recall that for a general ring graph the maximum independent set is of size \( |\frac{V}{2}| \), and is always 1 for a complete graph. This means that the \( L \)-movement for a \( k = 1 \) algorithm is always \( L = 1 \) for a complete graph, but increases with the number of vertices in a ring graph.

**5.4 Predictions**

We consider how \( L \) varies with \( k \). For these graphs we have the following table:

<table>
<thead>
<tr>
<th></th>
<th>Ring Graph</th>
<th>Complete Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k = 1 )</td>
<td>( L = 2 )</td>
<td>( L = 1 )</td>
</tr>
<tr>
<td>( k = 2 )</td>
<td>( L = 3 )</td>
<td>( L = 2 )</td>
</tr>
<tr>
<td>( k = 3 )</td>
<td>( L = 3 )</td>
<td>( L = 3 )</td>
</tr>
<tr>
<td>( k = 4 )</td>
<td>( L = 4 )</td>
<td>( L = 4 )</td>
</tr>
<tr>
<td>( k = 5 )</td>
<td>( L = 5 )</td>
<td>( L = 5 )</td>
</tr>
</tbody>
</table>
Fig. 8. The first two figures show curves for graphs where all rewards are drawn uniformly from the range 0 . . . 200. The next two figures show curves generated from four agent graphs, with rewards drawn from a Gaussian with $\mu = 100$ and $\sigma = 16$. The final two figures again have five agents, but rewards were drawn from a Gaussian with $\mu = 100$ and $\sigma = 32$.

This indicates that the difference between a $k = 1$ algorithm and a $k = 2$ algorithm is much greater in a complete graph than in a ring graph. Moreover, for a given graph and a given $k$, there is a unique $L$. Thus this analysis provides a heuristic by which one can \textit{a priori} determine if a team will suffer from the team uncertainty penalty by estimating the potential benefit of a higher $k$.

Previous results both from simulation (Taylor et al., 2010) and from on robots we see that $k = 1$ optimistic algorithms out perform $k = 2$ algorithms on ring graphs, but underperform on complete graphs. Our experimental analysis of $L$-movement shows that when increasing the $L$-movement of an algorithm from $L = 1$ to $L = 2$, a much greater gain is expected than when increasing the $L$-movement from $L = 2$ to $L = 3$. In the case of complete versus ring graphs, changing an algorithm from $k = 1$ to $k = 2$ corresponds exactly with changing $L$ from 1 to 2 (on a complete graph), and from 2 to 3 (on a ring graph). Our results also predict that a $k = 3$ algorithm on a ring graph
Fig. 9. The difference between $k = 1$ and $k = 2$ for a complete graph is far greater than for a Ring graph. In the ring graph, the increase from $k = 1$ to $k = 2$ results in an increase from $L = 2$ to $L = 3$. In the complete graph, however, we instead have $L = 1$ and $L = 2$.

would perform very similarly to a $k = 2$ algorithm, because $L = 3$ for both graphs. On a complete graph we would predict that a $k = 3$ algorithm would see a gain in total reward over a $k = 2$ algorithm, albeit it a smaller increase than that from $k = 1$ to $k = 2$. This is observed both in simulation and on robots.

6 Conclusions and Future Work

The DCEE framework is a recent extension of DCOPs enabling agents to cope with uncertainty in the environment and problems in which online reward is critical. While investigating the effect of teamwork in DCEEs, our earlier work (Taylor et al., 2010) showed the counter-intuitive team uncertainty penalty in simulation. This paper as further strengthened the claim that the team uncertainty penalty is an important phenomenon worth studying by confirming its existence on physical agents.

By understanding the team uncertainty penalty and what causes it, we will better be able to design algorithms to avoid or less its impact. In order to attempt to better understand this phenomenon, this paper has made a number of contributions. First, we have introduced the notion of $L$-movement in the context of a DCEE, an extension of $k$-optimality from the DCOP literature. Second, we show how $L$-movement and $k$-optimality are related in a graph-dependant manner, and suggest that the uncertainty penalty may be due to this relation. Third, we have introduced the notion of a configuration hypercube in a DCEE, which we will leverage in the future to theoretically analyze different classes of DCEE algorithms.

This paper has presented evidence that the team uncertainty penalty is possibly an intrinsic phenomenon of the $L$-movement algorithms, which is determined not by the specifics of the algorithms, but the level of $k$. In the future, we will continue analysis of the configuration hypercube to develop bounds on $L$-movement algorithms, design
algorithms to maximize the expected reward found on a given configuration hypercube, and to fully explain the team uncertainty principle.

7 Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments and suggestions. This work was supported in part by a fellowship provided by the National Center for Risk and Economic Analysis of Terrorism Events (CREATE).

Bibliography


Asynchronous Partitioning Framework

Vitaliy Freidovich1 and Amnon Meisels2,

1 Department of Mathematics and Computer Science, The Open University of Israel, Raanana, Israel
f_vitaliy@yahoo.com
2 Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel
am@cs.bgu.ac.il

Abstract. A new general framework for agent cooperation and coordination in solving distributed constraint satisfaction problems (DCSPs) is presented. The Asynchronous Partitioning Framework (APF) first partitions agents into groups of agents, based on some heuristic, prior to any search being conducted. During the partitioning one of the agents in each group is assigned the role of a group leader. Next, two distinct types of searches among the agents are performed in parallel. The first type of search is conducted within each group, in parallel and asynchronously to all searches in other groups. The second type of search, the global search, is conducted between the groups, and treats each group as if it is a single agent represented by its group leader. The structure of the groups remains static throughout the search processes. Two distinct algorithms implementing APF and a correctness proof are presented, and the advantages of APF are evaluated experimentally.

Keywords: Distributed Constraints Satisfaction; Distributed Search; Distributed Problem Solving; Distributed Partitioning; Distributed Framework.

1 Introduction

Distributed Constraint Satisfaction Problems (DCSPs) are composed of a set of agents. Each agent holds a set of variables, has a local representation of the constraint network, and is connected to other agents’ variables by some constraints. The role of an agent is to assign values to its variables, attempting to generate a locally consistent assignment. Consistency checks with external constraints are performed by exchanging messages with constraining agents to check consistency against their proposed local assignments. All agents cooperate in order to find a globally consistent assignment.

Several different algorithms for solving DCSPs have been proposed. The pioneering algorithm is Asynchronous Backtracking (ABT) [9]. In ABT agents are totally ordered and perform assignments to values in a completely asynchronous manner. Agents send new assignments to lower priority agents (e.g., ordered after them), and backtracking inconsistent partial assignments (Nogoods) to higher priority agents.
A more coordinated approach was proposed in the form of the Asynchronous Partial Overlay (APO) [3] algorithm. APO uses the concept of a mediator to centralize the search procedure in parts of the problem. Each mediator tries to solve a sub-problem which is centralized around it. When an agent is in conflict with other agents, it tries to assume the role of a mediator. If successful, the agent links to other agents, effectively extending the size of the sub-problem to be solved. In each step a mediator tries to extend the mediation session, to neighbors of agents that are inside the mediation session. Thus higher priority mediation sessions engulf lower priority mediation sessions during the search [1, 3].

The present paper proposes a new approach for solving DCSPs, based on the actual structure of a given DCSP as represented by its constraints graph. The Asynchronous Partitioning Framework (APF) adheres to the fail-first principle [2] by concentrating computational efforts first on the most constrained regions of the constraints graph. This is achieved by partitioning the DCSP into groups, each representing such a region, and conducting local searches inside each group in parallel to other groups. The results of the local searches are combined into a complete solution by performing a global search between the groups.

APF performs backtracking synchronously inside each group. This is in contrast to ABT that performs backtracking between the agents in a completely asynchronous manner. However, backtracking against agents in different groups is asynchronous in APF. This makes APF a partially coordinated and partially asynchronous algorithm.

APF may seem similar to the APO algorithm. The linking step of APO, which is performed in a mediation session, actually generates a group, using the terminology of APF. However, the major difference is that in APO the partitioning mechanism generates dynamic groups which grow during the search process, while APF uses a static partition, where the groups remain without change during its search phase.

2 Asynchronous Partitioning Framework

The Asynchronous Partitioning Framework (APF) is presented in Figure 1. It is composed of the following 4 distinct components:
1. The group partitioning algorithm, denoted by GroupPartition, is responsible for partitioning a given set of agents into disjoint groups, each group having a single group leader.
2. The local search algorithm, LocalSearch, conducts a search inside each group.
3. The global search algorithm, GlobalSearch, conducts a global search among all the groups.
4. The coordination engine is the heart of APF, and coordinates the LocalSearch and the GlobalSearch algorithms.

2.1 Detailed Description

APF starts by partitioning the agents of a given DCSP into disjoint groups (line 1) and then initiates search for the solution in lines 2 and 18. Synchronization of search
within each group is performed by the use of a token that is passed among the agents of a group and is extended by every agent adding its assignments on the token. While expanding a token, each agent tries to keep it consistent with its constraints inside its group and with the constraints with agents in groups that have a higher priority than the current group (e.g., before it in the global order), ignoring constraints with agents in lower priority groups. Each agent informs agents in lower groups it is connected to, if its value had been changed during the extension of the token, and assumes that they will adjust to its new assignment by changing their assignment if necessary. The notification regarding the change of value is not sent directly to the constrained agent in the lower priority group, but to its group leader. This is done in lines 4-5 of APF, enforcing inter-group constraints. At each step of LocalSearch, (line 3), the token of a group is passed through the group leader, to allow the group leader to react to value changes performed in higher priority groups.

In lines 6-13 of APF, the steps taken by a group leader upon receiving a value change notification from an agent \(a_i\), which belongs to a higher priority group, are described. The group leader first records the connection with the higher priority group, if it is still not aware of it, and then it forwards the notification to agent \(a_j\), for
which the original change of value from agent $a_i$ was intended. Next, the group leader of $a_j$ must act upon the location of the token within the group. If the token has never reached agent $a_j$ during the local search, the group leader doesn't have to do anything, as the local search was not affected yet by the constraint with $a_i$, since $a_j$ didn't have the chance to remove any value from its domain due to $a_i$'s previous assignment. However, if the token did reach agent $a_j$, and was then either advanced to the next agent, or backtracked to the previous agent, by the order of LocalSearch, the search must be restarted at $a_j$.

Restarting the search at $a_j$ allows APF to keep local consistency within the group. To restart the search at $a_j$, the group leader first regenerates a new token (line 10), based on $a_j$'s agent view. Each agent has an agent view which records its local knowledge regarding assignments of other agents, and which is updated by the LocalSearch algorithm each time $a_j$ receives the local token with the assignments that appear in the token. In line 11 the assignments of all agents preceding $a_j$ by the order of LocalSearch, must be reset in accordance with $a_j$'s agent view, so that they would be consistent with the newly regenerated token. Then, in line 12 the domains of all the agents must be reset, as some values might have been deleted from them due to $a_i$'s previous assignment which has now changed. Finally, in line 13 the search is restarted at agent $a_j$.

If the group leader discovers that the last agent by the order of LocalSearch, had managed to find a consistent assignment to its variable, all the intra-group constraints had been satisfied, and since LocalSearch is kept consistent with inter-group constraints as well, a partial assignment which is consistent with both intra and inter group constraints had been found, and the group leader can enter the consistent state at lines 14-15. In the opposite case, when the first agent by the order of LocalSearch had exhausted all possible assignments, the group leader enters the inconsistent state (lines 16-17).

Concurrently to the local searches performed inside each group, a global search, using the GlobalSearch algorithm, between the different groups is being conducted in lines 18-19 of APF. For this purpose a global token is used, which moves between the group leaders. The groups are ordered by the priorities assigned to them by the GroupPartition algorithm. Upon receiving a global token, each group leader advances or backtracks it, accordingly to the state of its group - consistent or inconsistent. Otherwise, if a local search is in progress, the group leader waits for it to enter one of these two states, before proceeding.

### 2.2 Required Conditions

Most of the component of APF - GroupPartition, LocalSearch and GlobalSearch – can be replaced by alternative algorithms. For the APF to be correct, these components must possess the following properties:

- The GroupPartition algorithm must:
  1. Form disjoint groups, such that each agent will belong to exactly one group.
  2. Generate a partition in which each group contains an agent which is connected to all other agents in that group.
  3. Elect exactly one agent in each group which is connected to all other agents in
the group, to be the group leader.
4. Assign a priority to each group, so that all the group priorities are injective.
5. Order the agents inside each group.

- The LocalSearch and the GlobalSearch algorithms has to be:
  1. Correct.
  2. Synchronous. Exactly one consistent partial assignment, extending the token, is performed at any given time.

- In addition, the LocalSearch algorithm must:
  3. Keep an agent view for each agent, recording assignments performed by constraining agents.
  4. Update agent views each time an agent receives a token, with the assignments that appear in the token.

### 2.3 Correctness Proof Summary

The following claims can be proven by examining the coordination engine of APF:

1. APF is sound. This claim follows from the facts that a consistent state of any LocalSearch implies that all agents inside the group are assigned consistently with the constraints between agents in the group and with agents in higher priority groups, and that the global token is advanced only after the current group has entered the consistent state, in accordance with the order of the groups.

2. APF is complete. This claim can be proven by examining (in negation) a consistent assignment which APF had failed to find. The claim is proven by examining the first group which failed to extend the appropriate part of the assignment, in contradiction to the completeness of the LocalSearch algorithm running inside that group.

3. APF terminates in a finite number of steps. This claim can be shown by examining the scenarios in which APF might fail to terminate. Since both LocalSearch and GlobalSearch are correct, this can occur only due to an infinite number of times APF restarts the search inside some group, at some agent \( a \). This situation can only be caused by an infinite number of messages received from higher priority agents. As the number of such agents is finite, some agent \( a_{i+1} \) belonging to a higher priority group, must send an infinite number of messages to \( a_i \). By simple induction it can be concluded that an infinite number of such agents must exist, each belonging to a higher priority group, in contradiction to the injectiveness of group priorities, as stated by property 4 of the GroupPartition algorithm.

4. The correctness of APF follows from the above 3 claims.

### 3 The AGP Algorithm

The Asynchronous Group Partitioning (AGP) algorithm is an implementation of the APF framework. It is composed of 4 phases:

1. Priority computation
2. Group partitioning
3. Group ordering
4. Search for a solution

Phases 1-3 are the implementation of the GroupPartition algorithm, and phase 4 implements the coordination engine of APF. The implementation of AGP synchronizes all agents to enter the next phase of the algorithm only after all the agents had completed the current phase.

### 3.1 Priority Computation Phase

The priorities of agents are calculated using a simple heuristic, which determines the priority of an agent based on the density of the agent's neighborhood in the constraints graph. The heuristic measures the density of the neighborhood of an agent \( a_i \) by employing the formula:

\[
P_{a_i} = k_i + \sum_{j=1}^{k_i} (\text{CommonNeighbors}(a_i, a_j))
\]

where \( P_{a_i} \) is \( a_i \)'s priority, \( k_i \) is the number of \( a_i \)'s neighbors, and \( \text{CommonNeighbors}(a_i, a_j) \) returns the number of agents which are neighbors of both \( a_i \) and of \( a_j \).

To compute the priorities in a distributed manner, an agent \( a_i \) sends each neighbor \( a_j \), a message which instructs it to check how many of the neighbors of \( a_i \) are also its own neighbors in the constraints graph. \( a_i \) then sums the results returned by each neighbor \( a_j \), adds to the result \( k_i \), and notifies all the agents of its computed priority.

Figure 2 (a) presents the results of the priority computation phase on an exemplary instance of a DCSP problem, the computed priority appears below each agent's id.

### 3.2 Group Partitioning Phase

Figure 3 presents the group partitioning phase of agents into disjoint groups, based on the priorities computed in the priority computation phase. A group leader is assigned...
**find_group_leader:**
1. agent ← first(sortedListOfNeighbors)
2. remove(agent, sortedListOfNeighbors)
3. if (agent = self)
   4. set_self_as_group_leader
5. else
   6. send(JOIN, agent)

**set_self_as_group_leader:**
1. groupLeader ← self
2. isGroupLeader ← true
3. add(self, group)
4. for each agent in waitingToJoinList do
   5. add(agent, group)
   6. send(JOINED, agent)
7. for each agent in listOfAllAgents do
   8. send(SET_MY_GROUP_LEADER, groupLeader, agent)

**join(agent):**
1. if (isGroupLeader)
   2. add(agent, group)
   3. send(JOINED, agent)
4. else if (isNotGroupLeader)
   5. send(NOT_JOINED, agent)
6. else
   7. add(agent, waitingToJoinList)

**joined(agent):**
1. groupLeader ← agent
2. isNotGroupLeader ← true
3. for each agent in waitingToJoinList do
   4. send(NOT_JOINED, agent)
5. for each agent in listOfAllAgents do
   6. send(SET_MY_GROUP_LEADER, groupLeader, agent)

Fig. 3. Group partitioning phase

to each group.

**Detailed Description.** The group partitioning phase starts by running find_group_leader, which tries to find the appropriate agent in each agent’s neighborhood for being its group leader. The search is conducted according to the priorities of the different neighbors, which are sorted initially into the sortedListOfNeighbors list, equal priorities being resolved lexicographically. During this process each neighbor is fetched from sortedListOfNeighbors in its turn, and sent a JOIN message which requests it to add the current agent to its group. If it denies the request, by sending a NOT_JOINED message, find_group_leader is called again by AGP, and continues the search to the next agent in the list. When the agent
encountered on the top of sortedListofNeighbors is the searching agent itself, the search after a group leader is terminated, and the agent assigns itself as its own group leader (set_self_as_group_leader).

set_self_as_group_leader marks the current agent as its own group leader, and sends a JOINED message to each agent waiting for its response in waitingToJoinList. This message informs each such agent that the current agent has indeed become its group leader. waitingToJoinList contains all the agents that had asked the current agent whether it is ready to become their group leader. Once the role of the current agent had been determined, it can inform them that it accepts their request. In addition, it informs all other agents that it had determined its group leader to be self by sending a SET_MY_GROUP_LEADER message.

When a JOIN message is received, AGP calls the join method. join checks whether the current agent is a group leader. If so it sends a JOINED message that informs the agent that it is in its group now, and the agent is added to the group list. Otherwise, if the current agent is not a group leader, it sends a NOT_JOINED message that informs the requesting agent to continue its search with some other agent. A third possibility is that the agent hadn't determined yet if it is a group leader or not (still waiting for a response from some other agent). In such a case the requesting agent is added to the waitingToJoinList, and is sent a response in a later stage, when the current agent had determined if it is, or is not, a group leader.

If a JOINED message is received, AGP calls the joined method to handle this situation. joined marks current agent as not being a group leader, and records the agent that is. Next, it informs all agents waiting for a response in waitingToJoinList that it cannot accept their request, by sending them a NOT_JOINED message. It then informs all agents that it had determined its group leader (SET_MY_GROUP_LEADER message).

**Correctness Proof Summary.** A number of claims can be proven by analyzing the algorithmic structure of the group partitioning phase.

1. After a finite number of steps, each agent \( a_i \) satisfies either isGroupLeader=true or isNotGroupLeader=true. This claim follows from the fact that the size of the waitingToJoinList is finite for all agents at any given point of time, since agents send join requests only to higher priority agents.
2. There exists no agent \( a_i \), for which both isGroupLeader=true and isNotGroupLeader=true.
3. For each agent \( a_i \), it holds that (isGroupLeader=true and isNotGroupLeader=false) or (isGroupLeader=false and isNotGroupLeader=true). This claim follows directly from claims 1 and 2.
4. It follows that each agent \( a_i \) will execute one method out of the set_self_as_group_leader and joined methods.
5. Exactly one method out of the methods join or set_self_as_group_leader is executed exactly once, for any agent \( a_i \).
6. Each agent \( a_i \) has exactly one group leader.
7. Each agent \( a_i \) belongs to exactly one group. This claim can be shown by first proving that each agent belongs to at least one group based on claim 6, and that each agent belongs to at most one group by looking on, from the moment of time in which \( a_i \) had been first added to some group.
8. Every group leader belongs to a single group. This follows directly from 7 and from set_self_as_group_leader.
9. Every agent \( a_i \) belongs to the same group as its group leader.
10. To each group belong agents, having the same group leader.
11. The priority of a group is the priority of its group leader. If two or more groups were calculated to have the same priority, their order is resolved lexicographically (see 7 and 9).
12. The group partitioning phase terminates after a finite number of steps (1, 5 and 6 above).

Example. Let us demonstrate the group partitioning phase by examining its execution on the instance of figure 2 (a). All the agents start their execution in parallel, and we will demonstrate the run on some random execution order.

\( a_1 \) enters the find_group_leader method and finds \( a_3 \) at the top of its sortedListOfNeighbors, as it has the highest priority out of all its neighbors. \( a_1 \) then sends it a JOIN message. \( a_3 \) receives the message and calls the join method. There \( a_1 \) is added to waitingToJoinList of \( a_3 \). Now \( a_3 \) starts its execution and calls find_group_leader. It finds \( a_6 \) on top of its sortedListOfNeighbors, so it sends \( a_6 \) a JOIN message. \( a_3 \) receives the message and calls the join method. Since its state has not been determined yet, it adds \( a_3 \) to its waitingToJoinList. Similarly, \( a_6 \) sends a JOIN message to \( a_3 \), and is added to its waitingToJoinList. Now \( a_3 \) starts its execution. It’s on the top of its sortedListOfNeighbors, so it calls set_self_as_group_leader. There it sets itself as a group leader, sends a JOINED message to \( a_6 \) and a SET_MY_GROUP_LEADER to all the agents. \( a_6 \) receives the JOINED message from \( a_3 \), marks it as its group leader, and sends a NOT_JOINED message to \( a_1 \). \( a_1 \) receives the NOT_JOINED, and calls find_group_leader again. Since \( a_1 \) is now on the top of its sortedListOfNeighbors, it calls set_self_as_group_leader. There it sets itself as a group leader, and sends a JOINED message back to \( a_1 \), that finally marks \( a_3 \) as its group leader. The results of the partitioning for all agents are shown in figure 2 (b).

3.3 Group Ordering Phase

The group ordering phase orders the agents within each group, in accordance with the Inner Group Priority (IGP), which is different from the priority calculated for the agent in the priority calculation phase. IGP is calculated by using a simple heuristic, which determines the IGP based on the size of groups which are connected to the current agent by some constraint. This heuristic uses the following formula:

\[
IGP_{a_i} = PG_{a_i} + \sum PCG_{ij}
\]  

(2)

where \( IGP_{a_i} \) is the inner group priority of agent \( a_i \), \( PG_{a_i} \) denotes the priority of the group of agent \( a_i \), and \( PCG_{ij} \) is the priority of the \( j^{th} \) connected group to one of its agents, \( a_i \) is connected. If the agent \( a_i \) is connected to a number of agents in the same group, that group will appear accordingly a number of times in the summation. In this way, an agent which is connected to a greater number of bigger groups will have a
higher IGP.
To calculate its IGP, each agent \( a_i \) adds to the priority of its group leader, the sum of the priorities of the group leaders of all the groups to which \( a_i \) is connected, and then notifies all the agents in its group of its IGP. Each agent then sorts the agents belonging to its group by their IGPs, resolving equal priorities lexicographically, thus defining the search order within the group.

### 3.4 AGP's GroupPartition Correctness

We have thus completed the presentation of AGP's implementation of the GroupPartition algorithm, and are now ready to refer to its correctness, in the following claim:

- The priority computation, group partitioning, and group ordering phases of AGP implement correctly the GroupPartition algorithm, and preserve all the properties required by APF. The proof of this claim is based on verifying that the GroupPartition algorithm is implemented correctly by these phases, as had been partially shown by the correctness proof summary in section 3.2, and that they meet all APF's requirements for the GroupPartition algorithm.

### 3.5 Search for Solution Phase

AGP's search for solution phase is directly based on the implementation of APF's search, as depicted in figure 1. AGP replaces APF's LocalSearch algorithm with the Synchronous Backtracking (SBT) [10] algorithm, and GlobalSearch algorithm with the Conflict based BackJumping (CBJ) [8] algorithm. It is easy to see that both SBT and CBJ meet the requirements of APF.

We have thus completed the presentation of the AGP algorithm. The correctness of AGP follows from the correctness of APF shown in section 2.3, the correctness of AGP's implementation of the GroupPartition algorithm, shown in section 3.4, and from the correctness of its search for solution phase shown in this section.

### 4 The AGP-CBJ Algorithm

As mentioned before, APF is a general framework for agent cooperation and coordination for solving DCSPs. This property is emphasized by the fact that both the search inside each group and the search between the groups can be implemented by any algorithm preserving certain properties. To demonstrate this idea we have implemented the Asynchronous Group Partitioning Conflict Based Back-Jumping (AGP-CBJ) algorithm, producing our second implementation of APF.

AGP-CBJ is a natural evolution of the AGP algorithm. It is identical to the AGP algorithm, besides the fact that inside each group, instead of using for the LocalSearch algorithm an SBT algorithm, a CBJ algorithm is used instead. Thus, in AGP-CBJ both the local search and the global use the CBJ algorithm. The correctness of AGP-CBJ
follows from the correctness of AGP, as well as from the fact that CBJ meets all APF’s requirements for the LocalSearch algorithm.

5 Experimental Evaluation

To evaluate the performance of AGP, a number of experiments had been conducted, which we describe below.

5.1 Experimental Setup

All experiments were conducted on an asynchronous simulator. In this simulator, agents are simulated by threads, which can only communicate by sending messages to each other. In each experiment, the network of the constraints had been randomly generated by selecting the probability $p_1$ of a constraint among any pair of variables, and the probability $p_2$, of a violation of a constraint, by assignments of values to a pair of constrained variables. Uniform random networks of constraints are specified by the number of variables in the network $n$, the number of values in the domain of each variable $k$, the constraints density parameter $p_1$, and the tightness parameter $p_2$, and are commonly used in experimental evaluation of DCSP algorithms [1,4,6,7].

Three sets of problems had been generated with the parameters $(n=10, k=10)$, $(n=15, k=10)$ and $(n=20, k=10)$. For each set the values of $p_1$ and $p_2$ were varied between 0.1 and 0.9. For each combination of the parameters $(n, k, p_1, p_2)$, 10 different instances of problems had been generated. Thus, for each combination of $(n, k)$, 810 different problems were generated, or 2430 problems in total. Our results are presented for the larger instances of 20 variables.

To evaluate the performance of distributed algorithms, two independent measures of performance are commonly used [1,4,6,7] - run time in the form of Non-Concurrent Constraint Checks (NCCCs) [5], and communication load in the form of total number of messages sent.

In all graphs which compare 2 algorithms the number of non-concurrent constraint checks had been averaged for all problems with the given number of $p_1$, and all values of $p_2$. That is, every point in the graphs represents the average run results of two algorithms over 90 problems.

5.2 Impact of the LocalSearch Algorithm

In figure 4 we present the comparison of the computational effort and the network load of AGP and AGP-CBJ. It is apparent that AGP-CBJ outperforms AGP, both by the computational effort performed and by network load, by an order of magnitude, for all values of $p_1$. What differentiates AGP-CBJ from AGP is the search technique used inside each group. These results indicate that the most significant computational effort is performed in APF algorithms, during the search inside each group. Therefore, the search technique used inside the groups will probably have a crucial impact on the performance of the algorithm.
5.3 Partitioning Characteristics

An interesting question which arises upon examining AGPs’ behavior concerns the group partitioning strategy. One might wonder whether the partitioning of agents into groups is in fact a helpful strategy which reduces the computational effort required to solve a DCSP problem. To address this question, a number of experiments had been conducted. The experiments summarized in Figure 5 had been conducted for all values of $p_1$ and $p_2$ between 0.1 and 0.9. That is, each graph summarizes the partitioning of 810 problems into groups.

The first set of experiments tries to find a correlation between the number of groups generated for each problem and the computational effort exhibited by the algorithms for that problem. Figure 5 (a) plots the average number of NCCCs performed by the AGP-CBJ algorithm against the number of groups that problems were partitioned into. Figure 5 (a) shows a strong correlation between the number of the groups, problems were partitioned into, and the computational effort needed to solve these problems. As the number of the groups a problem was partitioned into increases, the computational effort performed by the AGP-CBJ algorithm decreases.

An exception to this relation are problems partitioned into a single group. To better understand this exception we need to reflect on the nature of such problems. From Claim 10 in the correctness proof summary of section 3.2 it follows that if all agents were partitioned into the same group, they are all connected to their group leader. Statistically, there is probably more than one agent which is connected to all other agents in its group. These problems are dense, representing cliques or almost cliques, which are most commonly insoluble, and due to the high density, a no solution can be found quite quickly on them. This explains the low computational effort needed to solve single-group problems.

Similar results had been observed both for the AGP algorithm, and for drilldowns performed for specific values of $p_1$.

The next question to be investigated is, for what kind of problems the partitioning strategy will be most efficient. To answer this question, Figure 5 (b) presents for each value of $p_1$ the average number of groups, problems having that value of $p_1$ were partitioned to. There is a clear correlation between the number of the groups
generated, and the problem density parameter, \( p_1 \). As the density of a problem increases, the number of the groups generated decreases. Since the algorithms were found to be most efficient for problems partitioned into greater number of groups, we will expect the AGP and AGP-CBJ algorithms to be more efficient for problems with lower values of \( p_1 \).

### 5.4 Comparison to APO & ABT

For the comparison of AGP-CBJ and APO algorithms, one needs to use the correct APO version - CompAPO [1]. It is also the best performing version of APO [1]. Figures 6 (a)-(b) present the comparison of the AGP-CBJ and CompAPO algorithms. It is apparent that AGP-CBJ outperforms CompAPO with respect to the computational effort performed. However, CompAPO sends a smaller number of messages than AGP-CBJ, due to its centralized nature.

Figures 6 (c)-(d) present the comparison of the AGP-CBJ and ABT algorithms. We can see that AGP-CBJ generally outperforms ABT with respect to the computational effort. This is true for instances with both low constraint density \((p_1 \leq 0.4)\) and with the highest constraint density \((p_1 \geq 0.8)\). However, AGP-CBJ is outperformed by ABT for instances of problems with density \(0.5 \leq p_1 \leq 0.7\). Regarding the network load, AGP-CBJ outperforms ABT, for all \( p_1 \), except for \(0.5 \leq p_1 \leq 0.6\).

### 5.6 Experimental Evaluation Summary

The importance of the search technique used inside each group was found to be great. AGP-CBJ outperformed AGP by an order of magnitude with respect to all measures.

The benefit of the group partitioning strategy was found experimentally to be positive. As the number of groups a problem was partitioned into increases, the computational effort drops. Problems that were partitioned into a larger number of groups needed a lower computational effort to find a solution. A direct correlation was found between the number of groups a problem was partitioned into and the problem density parameter \( p_1 \). As the density of a problem increases, the number of
generated groups decreases. Thus, the AGP and AGP-CBJ algorithms are expected to be more efficient for problems with lower values of $p_1$. Further experiments indicated that AGP-CBJ performs well for values of $p_1$ that are less than 0.5. Surprisingly, AGP-CBJ is efficient as well for values of $p_1$ which are greater than 0.8.

The static group partitioning strategy of AGP-CBJ was compared to the dynamic group partitioning strategy of APO. The static strategy of AGP-CBJ was found to be superior by an order of magnitude than the dynamic strategy of APO. This applies to computational effort. However, the APO algorithm exchanges much less messages than AGP-CBJ.

The experimental evaluation included also a comparison of the proposed AGP-CBJ algorithm to the fully asynchronous ABT algorithm. AGP-CBJ outperforms ABT for most values of problem density $p_1$, except for the range $0.5 \leq p_1 \leq 0.7$.

6 Discussion

A new framework, the Asynchronous Partitioning Framework (APF), for agent cooperation and coordination in solving DCSPs was presented. APF is driven by the structure of the underlying constraints graph of a given DCSP, focusing the computational efforts on the more difficult regions of the problem first. APF exploits
the benefits of the fail first principle by partitioning the problem into groups that follow the structure of the underlying constraints graph.

APF allows the replacement of most of its components - the GroupPartition, the LocalSearch and the GlobalSearch algorithms - by any other algorithm or strategy which preserve certain properties. The required properties for each component were defined and the correctness of APF proven.

To demonstrate the generality of APF, two distinct algorithms which implement it were presented – the AGP and the AGP-CBJ algorithms. The framework was experimentally evaluated by conducting a series of experiments on the AGP and AGP-CBJ algorithms. The most significant result is that the group partitioning strategy of APF generated a strongly improved performance.

The presented framework, as well as its implementations, the AGP and AGP-CBJ algorithms, can be improved in a number of ways. First, the LocalSearch and GlobalSearch can be replaced by more advanced DCSP algorithms, such as AFC [6]. Second, some of APF’s requirements can be relaxed, to allow a greater degree of versatility to the implementing components. Third, the heuristics of the GroupPartition algorithm can be upgraded. Since the number of groups a problem is partitioned into was found to have a great impact on the performance of APF, a more sophisticated group partitioning strategy can limit the number of agents in a group or split larger groups into smaller ones.

References

Effect of DisCSP Variable-Ordering Heuristics in Scale-free Networks

Tenda Okimoto, Atsushi Iwasaki, and Makoto Yokoo

Kyushu University, Fukuoka 8190395, Japan
{tenda@agent., iwasaki@, yokoo@is.kyushu-u.ac.jp

Abstract. A Distributed Constraint Satisfaction Problem (DisCSP) is a constraint satisfaction problem in which variables and constraints are distributed among multiple agents. Various algorithms for solving DisCSPs have been developed, which are intended for general purposes, i.e., they can be applied to any network structure. However, if a network has some particular structure, e.g., the network structure is scale-free, we can expect that some specialized algorithms or heuristics, which are tuned for the network structure, can outperform general purpose algorithms/heuristics.

In this paper, as an initial step toward developing specialized algorithms for particular network structures, we examine variable-ordering heuristics in scale-free networks. We use the classic asynchronous backtracking algorithm as a baseline algorithm and examine the effect of variable-ordering heuristics. First, we show that the choice of variable-ordering heuristics is more influential in scale-free networks than in random networks. Furthermore, we develop a novel variable-ordering heuristic that is specialized to scale-free networks. Experimental results illustrate that our new variable-ordering heuristic is more effective than a standard degree-based variable-ordering heuristic. Our proposed heuristic reduces the required cycles by 30% at the critical point.

1 Introduction

A surprisingly wide variety of Artificial Intelligence (AI) problems can be formalized as constraint satisfaction problems (CSPs). A CSP is a problem that finds a consistent assignment of values to variables. A Distributed Constraint Satisfaction Problem (DisCSP) is formalized as a CSP in which variables and constraints are distributed among multiple agents [1]. In DisCSP, agents assign values to variables, attempting to generate a locally consistent assignment that is also consistent with all the constraints between agents.

Asynchronous BackTracking algorithm (ABT), which was first presented by Yokoo [2], is the most basic algorithm for solving DisCSPs. It is also the first complete and asynchronous search algorithm for DisCSPs. ABT allows agents to act asynchronously and concurrently without any global control, while guaranteeing the completeness of the algorithm. Various algorithms have been developed for solving DisCSPs, e.g., Distributed BackTracking algorithm [3], an ABT
based algorithm without adding links [4], Dynamic Distributed BackJumping [5], Asynchronous Partial Overlay [6], and Dynamic ordering for ABT [7, 8].

Recently, scale-free graphs in complex networks, introduced by Barabási and Albert [9, 10], has become a very popular interdisciplinary research topic. These graphs have been proposed as a generic and universal model of network topologies that exhibit power-law distributions in the connectivity of network nodes. A scale-free network is inhomogeneous in nature, i.e., there exist a small number of nodes that have many connections, while most nodes have very few connections.

Although there has been a lot of research in network structure for CSPs [11, 12], there is little research in network structure for DisCSP. In this paper, as an initial step toward developing specialized algorithms/heuristics for particular network structures in DisCSP, we examine the effect of variable-ordering heuristics of ABT in scale-free networks. Although a variety of more efficient, sophisticated algorithms have been developed for solving DisCSPs, we focus on ABT as a baseline algorithm, since it is one of the simplest algorithms and is suitable for our purpose. We believe that our analysis and results can be applied to other sophisticated algorithms.

In the rest of this paper, we show that the choice of variable-ordering heuristics is more influential in scale-free networks than in random networks. Specifically, we show that the performances of ABT in the former network depend on which variable-ordering heuristics is used much more than that in the latter network, since the degree distribution of scale-free networks is significantly different from that of random networks. We examine how the performance of ABT in scale-free networks changes in terms of the depth and number of the backedges of pseudo-trees. Given a variable-ordering, ABT determines a pseudo-tree and searches for a solution from it. Since the depth and number of backedges greatly affect the network structure, it is expected that the performance of ABT changes based on those factors. However, surprisingly, our experiments reveal that the performance does not significantly change.

Furthermore, we develop a novel variable-ordering heuristic called Average Length between Hubs (ALH) specialized for scale-free networks. Our experiments show that ALH outperforms a standard degree-based variable-ordering heuristic in scale-free networks. As far as the authors aware, there exists virtually no work on variable-ordering heuristics specialized for scale-free networks in DisCSP, although many studies have dealt with variable-ordering heuristics [8, 13, 14, 15, 16].

The rest of our paper is organized as follows. We describe the definition of a DisCSP (Section 2) and introduce a scale-free network (Section 3). We examine the performance of ABT in scale-free and random networks (Section 4). Next, we present a novel variable-ordering heuristic that is specialized to scale-free networks and show that our new variable-ordering heuristic is effective for scale-free networks (Section 5). Finally, we give a discussion (Section 6) and present a conclusion and some future work (Section 7).
2 Distributed Constraint Satisfaction Problem

A Constraint Satisfaction Problem (CSP) [17] consists of $m$ variables $x_1, \ldots, x_m$, whose values are taken from finite, discrete domains $D_1, \ldots, D_m$, respectively, and a set of constraints on their values. A constraint is defined by a predicate. That is, the constraint $p(k; x_{k1}, \ldots, x_{kj})$ is a predicate that is defined on Cartesian product $D_{k1} \times \ldots \times D_{kj}$. This predicate is true iff the value assignment of these variables satisfies this constraint. Solving a CSP is equivalent to finding an assignment of values to all variables such that all constraints are satisfied.

A Distributed Constraint Satisfaction Problem (DisCSP) is a CSP in which the variables and constraints are distributed among multiple agents [1]. We assume the following communication model:

- Agents communicate by sending messages. An agent can send messages to other agents iff the agent knows the addresses of the agents.
- The delay in delivering a message is finite, although random. For transmission between any pair of agents, messages are received in the order in which they were sent.

Each agent has variables and tries to determine their values. However, there exist inter-agent constraints, and the value assignment must satisfy these inter-agent constraints.

2.1 Asynchronous Backtracking

Asynchronous BackTracking algorithm (ABT), which was first presented by Yokoo [1], is the most basic algorithm for solving DisCSPs. We make the following assumptions while describing this algorithm for simplicity. Relaxing these assumptions to general cases is relatively straightforward:

- Each agent has exactly one variable.
- All constraints are binary.
- Each agent knows all constraint predicates relevant to its variable.

In ABT, the priority order among agents is determined. First, agents instantiate their variables concurrently and send their assigned values to the agents that are connected to them by outgoing links, i.e., there exists a link between two agents who are involved by a binary constraint, and the link is directed from the higher priority agent to the lower priority agent. Then all agents wait for and respond to messages. After each update of its assignment, an agent sends its new assignment to all outgoing links. An agent that receives an assignment from an incoming link, tries to find an assignment for its variable that does not violate a constraint with the assignment it received.

The main message types communicated among agents are $ok$? messages and $nogood$ messages. An $ok$? message carries an assignment of an agent. When agent $A_i$ receives an $ok$? message from agent $A_j$, it places the received assignment in a data structure called Agent_View, which holds the last assignment $A_i$ received
from higher priority neighbors such as $A_j$. Next, $A_i$ checks if its current assignment is still consistent with its Agent\_View. If it is consistent, $A_i$ does nothing. If not, then $A_i$ searches its domain for a new consistent value. If it finds one, it assigns its variable and sends ok? messages to all lower priority agents linked to it. Otherwise, $A_i$ backtracks.

The backtrack operation is executed by sending a nogood message that contains an inconsistent partial assignment. nogood messages are sent to the agent with the lowest priority among the agents whose assignments are included in the inconsistent tuple in the nogood message. Agent $A_i$ that sends a nogood message to agent $A_j$ assumes that $A_j$ will change its assignment. Therefore, $A_i$ removes from its Agent\_View the assignment of $A_j$ and makes an attempt to find an assignment for its variable that is consistent with the updated Agent\_View.

### 3 Scale-Free Network

In recent years, various complex networks have been identified as having a scale-free structure [9, 18, 19], e.g., the Internet, SNS, and the citation relation graphs of scientific articles. Traditionally, these networks are approximated as random graphs, but the degree distributions of these networks (and other networks in nature) are significantly different from the degree distribution of random graphs. The study of random graphs was originally initiated by Erdős and Rényi (ER model) [20]. In this model, most nodes have approximately the same degree and the degree distribution follows a Poisson distribution.

A scale-free network is characterized by a power-law degree distribution as follows:

$$p(k) \propto k^{-\gamma},$$

where $k$ is the degree and $\gamma$ is the exponent that depends on each network structure. Scale-free networks have no scale because there is no typical number of links. The random network models assume that the probability that two nodes are connected is random and uniform. In contrast, most real networks exhibit some preferential connectivity. For example, a newly created webpage will more likely include edges to well-known, popular documents that already have high connectivity. This example indicates that the probability with which a new node connects to an existing node is not uniform, but there is a higher probability of being linked to a node that already has a large number of connections.

### 4 Influence of Variable-Ordering Heuristics in Scale-Free Networks

In this section, we show that the choice of variable-ordering heuristics is more influential in scale-free networks than in random networks. We used a discrete event simulation [21] to compare the performance of ABT-based algorithms, where each agent maintains its own simulated clock. An agent's time is incremented by one simulated time unit whenever it performs one computation cycle.
One cycle consists of reading all incoming messages, performing local computation, and sending messages. We assume that a message issued at time $t$ is available to the recipient at time $t+1$. We analyzed the performance for the number of cycles required to solve the problem.

There are some other simulations to evaluate DisCSP algorithms, e.g., Non Concurrent Constraints Checks (NCCCs). However, we analyze different variable ordering heuristics on a single algorithm and the computational cost for each cycle is almost identical. Therefore, we believe that using only cycles rather than NCCCs is enough.

In this paper, the Java program developed by Sun Microsystems Laboratories is used as a scale-free network formation tool [22]. This program can generate scale-free networks giving the number of nodes, exponent $\gamma$, and the minimal degree of each agent $md$. More specifically, this program can generate a power-law list of nodes and edges.

We examine the performance of ABT in random and scale-free networks. Scale-free networks are generated by the tool with the following parameters: nodes=100, $md=2$, and $\gamma=1.8$. To generate random networks, we chose nodes=100 and edges=247, so that the number of constraints will resemble those of the scale-free networks. We set the domain size of each variable to three, i.e., $domain=3$ which means $|D_1|=...,=|D_m|=3$ for $m$ variables $x_1,...,x_m$. For the evaluations, we generate ten random and ten scale-free networks. Assume $r_1,...,r_{10}$ for the ten random networks and $sf_1,...,sf_{10}$ for the ten scale-free networks. For each network, the constraint tightness is varied from 0.1 to 0.9 by 0.1. For each constraint tightness, 100 random problem instances are generated. Thus, the results represent the averages of these 100 instances in all ten networks. For a variable-ordering of ABT, we determine ten different random variable-orderings.

In Figure 1, we show the performance of ABT with three different random variable-orderings in random network $r_1$ and scale-free network $sf_1$ that exhibit characteristic results. When the constraint tightness is less than 0.3 or greater than 0.3, ABT can terminate early, i.e., ABT can easily find a solution for less than 0.3, and it can easily find that the problem is unsolvable for greater than 0.3. When the constraint tightness equals 0.3, the required cycles of ABT are maximum in $r_1$ and $sf_1$. We call such a peak the critical point.

Random-ABT-Max (Random-ABT-Min) represents the performance of ABT in random network $r_1$, whose required cycles at the critical point are maximum (minimum). SF-ABT-Max and SF-ABT-Min represent the performance of ABT as above in scale-free network $sf_1$. In addition, Random-ABT-Degree and SF-ABT-Degree represent the performance of ABT with a standard degree-based variable-ordering heuristic. In this heuristic, the priority of nodes is determined one by one. First, we choose $n_{1st}$, which has the highest degree. Second, we choose node $n_{2nd}$, which has the highest degree and connected to $n_{1st}$. Similarly, we keep on choosing a node, that has the highest degree with the nodes already chosen, breaking ties using the degree with the unchosen nodes.

\[^1\text{For } \gamma=2.2, 2.6, 3.0, \text{ the essential results did not change.}\]
The performance of ABT significantly depends on variable-ordering in scale-free networks. In random network $r_1$, the required cycles at the critical point vary from 235 to 283 cycles (Figure 1(a)). On the other hand, in scale-free network $sf_1$, the required cycles vary from 47 to 1171 cycles (Figure 1(b)). We confirmed that similar results were obtained in other networks, i.e., in $r_2,...,r_{10}$ and $sf_2,...,sf_{10}$. Particularly, in scale-free networks, ABT with a standard degree-based variable-ordering heuristic requires the smallest cycles at the critical point.

Furthermore, we examine the effect of the depth and the number of backedges in a pseudo-tree on the performance of ABT in scale-free networks. According to a variable-ordering, a pseudo-tree is determined whose depth is the length of the

<table>
<thead>
<tr>
<th>ABT</th>
<th>Depth</th>
<th>Backedges</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABT 1</td>
<td>17</td>
<td>207</td>
<td>10253</td>
</tr>
<tr>
<td>ABT 2</td>
<td>14</td>
<td>176</td>
<td>7815</td>
</tr>
<tr>
<td>ABT 3</td>
<td>15</td>
<td>220</td>
<td>2279</td>
</tr>
<tr>
<td>ABT 4</td>
<td>22</td>
<td>327</td>
<td>1673</td>
</tr>
<tr>
<td>ABT 5</td>
<td>13</td>
<td>173</td>
<td>777</td>
</tr>
<tr>
<td>ABT 6</td>
<td>12</td>
<td>175</td>
<td>380</td>
</tr>
</tbody>
</table>

Fig. 1. Performance of ABT in random and scale-free networks
The longest path from the root agent to one of the leaf agents. A backedge is a link between two agents that are not in a direct parent-child relationship. Our initial expectation was that the performance of ABT would improve with shallower depth and fewer backedges. In Table 1, we show the depth and the number of backedges of the pseudo-trees and the required cycles of ABT at the critical point with six different variable-orderings, where domain=10. Here, we increased the domain size to make the required cycles vary significantly according to variable-orderings. As shown in Table 1, we cannot see any direct relationship between the performance and the parameters we examined (i.e., tree depth and number of backedges). For example, in “ABT 1”, the required cycle at the critical point is 10253, the depth is 17, and the number of backedges is 207. On the other hand, in “ABT 4”, the required cycle at the critical point is 1673, the depth is 22, and the number of backedges is 327.

The experimental results reveal that the choice of variable-ordering heuristics is influential in scale-free networks. We don’t see any direct relationship between the performance of ABT and the parameters of a pseudo-tree (i.e., depth and number of backedges).

5 A Variable-Ordering Heuristic for Scale-Free Networks

The previous section reveals that the performance of ABT is affected by the priority of the order of agents, i.e., variable-ordering. In this section, we propose a novel variable-ordering heuristic called Average Length between Hubs (ALH). Based on the results so far, since ALH focuses on the average length between hubs, it is specialized for scale-free networks. This section introduces our proposed variable-ordering heuristics and evaluates the performance by simulation.

5.1 Heuristic

Let $G = (N, E)$ be a graph, where $N = \{ n_i | i \in \mathbb{N} \}$ is a set of nodes (agents) and $E = \{ e(n_i, n_j) | n_i, n_j \in N, n_i \neq n_j \}$ is a set of edges. A node is called a hub if it has a larger number of connections than constant $c \in \mathbb{N}$. Let $H$ be set of hubs

$$H = \{ n_i | n_i \in N, \text{deg}(n_i) \geq c \}$$

where $\text{deg}(n_i)$ is the degree of node $n_i$. Each agent knows whether he belongs to $H$.

Next, we define border-set nodes by using the distance between nodes $\text{dis} : N \times N \to \mathbb{N}$, i.e., $\text{dis}(n_i, n_j)$ gives the number of the edges of the shortest path between $n_i$ and $n_j$. For node $n_i$, the average distance of the shortest paths to each hub in $H$ is defined as follows:

$$n^\text{av}_i = \frac{\sum_{n_j \in H} \text{dis}(n_i, n_j)}{|H|}.$$

The average distance between hubs is defined as follows:

$$h^\text{av} = \frac{\sum_{n_i \in H} n^\text{av}_i}{|H|}.$$
Then, border-set $BS$ is defined as:

$$BS = \{ n_i | n_i^{av} \leq h^{av} \}.$$ 

The priorities of agents are determined using $BS$. Basically, a node in $BS$ has a higher priority than a node that is not in $BS$. Between two nodes in $BS$, the node that is not in $H$ has a higher priority. If two nodes, $n_i$ and $n_j$, in $BS$ are also in $H$, then $n_i$ has a higher priority than $n_j$ when $\text{deg}(n_i) > \text{deg}(n_j)$ (and vice versa). If two nodes, $n_i$ and $n_j$, in $BS$ are not in $H$, then $n_i$ has a higher priority than $n_j$ when $n_i^{av} < n_j^{av}$ (and vice versa). Ties are broken using the degrees. Further ties are broken using the lexicographical order of identifiers. Then the priority among two nodes that are not in $BS$ is determined by the total distance between $BS$. More specifically, for node $n_i \notin BS$, denote the total distance to the nodes of $BS$ as $td(n_i) = \sum_{n_j \in BS} \text{dis}(n_i, n_j)$. For two nodes, $n_i$ and $n_j$ that are not in $BS$, $n_i$ has a higher priority than $n_j$ when $td(n_i) < td(n_j)$ (and vice versa). Ties are broken using the degrees. Further ties are broken using the lexicographical order of identifiers.

If all hubs are directly connected, the ALH becomes equivalent to a degree-based heuristic, since $BS$ contains only nodes in $H$. Consider a scale-free network where each hub is not directly connected. In ALH, the nodes in $BS$ have the highest priority, i.e., the node in $BS$ has a higher priority than the hubs. Let us consider the pseudo-tree defined by this ordering. In the pseudo-tree, the hubs are placed below the nodes in $BS$, i.e., the hubs are siblings of the nodes in $BS$. Also, under each hub, we can expect that there exists a cluster of nodes, which is independent from other clusters, given that the values of variables in $BS$ are determined. Thus, we can expect that ABT can efficiently solve such a problem instance since these clusters can be solved independently. The cost of implementing the proposed heuristic ALH is enough low compared to the cost of the ABT, since finding a shortest path can be done $O(n)$.

Let us show a simple example. A constraint network of a DisCSP represented as Figure 2(a) exists, where $H_1$ and $H_2$ are hubs. For nodes $H_1, H_2, n_1, \ldots, n_4$, their degrees satisfy the following condition: $\text{deg}(H_1) > \text{deg}(H_2) > \text{deg}(n_1) > \text{deg}(n_2) > \text{deg}(n_3) > \text{deg}(n_4)$. Figure 2(b) represents the pseudo-tree determined by a degree-based heuristic. Since $H_1$ has the highest degree, it becomes the root of this pseudo-tree. Since $h^{av}=1$, $n_1^{av} = n_2^{av} = 1$, $n_3^{av} = 2$, and $n_4^{av} = 3$, $BS$ is determined as follows:

$$BS = \{ H_1, H_2, n_1, n_2 \}.$$ 

Thus, among these nodes, priority ordering is determined as:

$$n_1, n_2, H_1, H_2, n_3, n_4,$$

where $n_1$ is the highest and $n_4$ is the lowest.

Figure 2(c) represents the pseudo-tree determined by ALH. The nodes in $BS$ are placed around the root of the pseudo-tree. The hubs are placed just below the root as siblings, and in particular, hubs $H_1$ and $H_2$ are placed on different branches in this pseudo-tree.
5.2 Evaluations

In our evaluations, we show that ALH is effective and can reduce the required cycles at the critical point in scale-free networks. More specifically, we compare the effect of ALH compared to a standard degree-based heuristic in the following three kinds of scale-free networks:

(SFN 1): \( \text{nodes}=100, \gamma=1.8, \) and \( md=2, \)
(SFN 2): \( \text{nodes}=200, \gamma=1.8, \) and \( md=2, \)
(SFN 3): \( \text{nodes}=100, \gamma=1.8, \) and \( md=3. \)

The evaluations were conducted with \( \text{domain}=10. \) For each parameter we generated ten scale-free networks. For each network, the constraint tightness was varied from 0.1 to 0.9 by 0.1. For each constraint tightness, 100 random constraint instances were generated. The results represent the averages of these 100 instances for all ten scale-free networks (1000 in total). The experimental results in SFN 1 are summarized in Figure 3(a), in which ALH-ABT-Average represents the performance of ABT with the ALH variable-ordering heuristic and Degree-ABT-Average represents the performance of ABT with the standard degree-based variable-ordering heuristic. Here, the critical point appears when the constraint tightness is around 0.4. At the critical point, ALH-ABT-Average requires 678 cycles while Degree-ABT-Average requires 955 cycles. Thus, ALH-ABT-Average performs approximately 30% better than Degree-ABT-Average at the critical point in SFN 1. We confirmed that ALH is also effective in SFN 2 and SFN 3.
The experimental results in SFN 2 are summarized in Figure 3(b). The network in SFN 2 is larger than that in SFN 1, i.e., the number of nodes in SFN 2 is 200, compared to 100 in SFN 1. ALH-ABT-Average requires 6156 cycles and Degree-ABT-Average requires 7487 cycles at the critical point. Thus, ALH-ABT-Average performs approximately 19% better than Degree-ABT-Average.

The experimental results in SFN 3 are summarized in Figure 3(c). The network in SFN 3 is more complicated than SFN 1, i.e., the minimal degree of each agent increased from $md=2$ to $md=3$. ALH-ABT-Average requires 1489 cycles and Degree-ABT-Average requires 2083 cycles at the critical point. ALH-ABT-Average performs approximately 30% better than Degree-ABT-Average at the critical point in SFN 3.

The experimental results reveal that ALH outperforms the standard degree-based heuristic in three scale-free networks, varying the number of nodes and the minimal degree. We also confirmed that fact did not change with other parameter settings.

6 Discussion

The previous section showed that the standard degree-based heuristic is outperformed by the ALH heuristic. One might expect that it is also outperformed by the other simple heuristics, since the ALH heuristic, particularly its way of determining the border set, is somewhat complicated. Thus, we consider a simple variable-ordering heuristic, called a naive heuristic described below.

Let us define a naive border-set ($NBS$) for that heuristic, instead of a BS for ALH. Whether a node belongs to $NBS$ is determined by the distance between the two nearest hubs to the node. Formally, for node $n_i \in N$, denote the two
nearest hubs from \( n_i \) as \( h_{i,1st}, h_{i,2nd} \) \( \in H \) and denote the distances to \( h_{i,1st} \) and \( h_{i,2nd} \) from the node as \( \text{dis}_{1st}(n_i) \), \( \text{dis}_{2nd}(n_i) \), respectively. Then, we define \( NBS \) as follows:

\[
    NBS = \{ n_i | |\text{dis}_{1st}(n_i) - \text{dis}_{2nd}(n_i)| \leq 1 \}.
\]

In short, \( NBS \) contains nodes that lies exactly in the middle of the two nearest hubs. The priority among nodes is determined in exactly the same way as ALH, except that we use \( NBS \) instead of \( BS \). The experimental result is summarized in Figure 4. Naive-ABT represents the performance of ABT with the naive heuristic. ALH-ABT performs approximately 3.7 times better than Naive-ABT at the critical point. Precisely, the required cycles for ALH-ABT is 1016, while that for Naive-ABT is 3744 at that point. As a result, we can say that the simplified version of the heuristics fails to perform as well as ALH.

We examined the reason why the performance of the naive heuristic is much worse than that of ALH, and found that the size of \( NBS \) of the naive heuristic is much larger than the size of \( BS \) of ALH. In fact, \( NBS \) contains at least twice as many agents as \( BS \). Therefore, we conjecture that the size of \( BS \) should be small; otherwise, the priority based on \( BS \) becomes less informative.

In previous evaluations, we set the number of hubs to two. This seems to be a reasonable choice to make the size of the border-set (\( BS \)) small. We further examine the performance of ABT with the ALH heuristic by varying the number of hubs from two to five. The experimental results are summarized in Figure 5, in which ABT-Hub-\( k \) represents the performance of ABT when choosing the number of hubs as \( k \). The performance is basically unchanged even if we change the number of hubs, i.e., the required cycles at the critical point for ABT-Hub-5 is 1048, while that for ABT-Hub-2 is 1004. These results imply that the choice
of the number of hubs is not so influential to the performance of ABT with the ALH heuristic.

Note that, this result does not explain how many hubs we should choose in any scale-free networks. We will research a good value for some of constants (constant for selecting hubs) as a future work.

7 Conclusions

In this paper, we showed that the choice of variable-ordering heuristics is more influential in scale-free networks than in random networks. We observed that in scale-free networks there is more significant difference between maximum and minimum of the required cycles than in random networks.

Furthermore, we examined how the performance of ABT in scale-free networks changes in terms of the depth and number of backedges of pseudo-trees. The experimental result revealed that these parameters do not significantly affect the performance of ABT in scale-free networks.

Finally, we developed a novel variable-ordering heuristic called Average Length between Hubs (ALH) specialized for scale-free networks. We showed that ALH outperforms a standard degree-based variable-ordering heuristic in scale-free networks and can reduce the required cycles by 30% at the critical point.

As future works, we hope to develop dynamic variable-ordering heuristics/algorithms that are specialized to scale-free networks.

References