Generalizing DPOP: Action-GDL, a new complete algorithm for DCOPs

M. Vinyals
IIIA, Artificial Intelligence Research Institute
Spanish National Research Council
meritxell@iiia.csic.es

J.A. Rodriguez-Aguilar
IIIA, Artificial Intelligence Research Institute
Spanish National Research Council
jar@iiia.csic.es

J. Cerquides *
WAI, Dep. Matemàtica Aplicada i Anàlisi
Universitat de Barcelona
cerquide@maia.ub.es

ABSTRACT

In this paper we propose a novel message-passing algorithm, the so-called Action-GDL, as an extension to the Generalized Distributive Law algorithm (GDL) [1] to efficiently solve DCOPs. We show the generality of Action-GDL by proving that it has DPOP, one of the low-complexity, state-of-the-art algorithm to solve DCOPs, as a particular case. Finally, we provide empirical evidences to illustrate how Action-GDL can outperform DPOP in terms of computation, communication and parallelism needed to solve the problem.

1. INTRODUCTION

Multi-agent Coordination Problems (MCPs), also called distributed multi-agent decision making problems, are a class of problems in MAS focusing on how to coordinate agents' actions in order to yield a global desired behaviour for the MAS. Distributed Constraint Optimization Problems (DCOPs) are an extension of Constraint Optimization Problems (COPs) that can model a large class of MCPs [9].

State-of-the-art complete algorithms to solve DCOPs adopt two main approaches: search and dynamic programming. Search algorithms, like ADOPT [7], require linear-size messages, but an exponential number of messages. Dynamic programming algorithms, represented by the DPOP algorithm and its extensions [10], only require a linear number of messages, but their complexity lies on the message size, which may be very large.

In this paper, we formulate a new algorithm, the so-called Action-GDL, that takes inspiration from the GDL algorithm [1], extending and applying it to DCOPs. GDL is a general message-passing algorithm that exploits the way a global function factors into a combination of local functions generalizing a large family of well-known algorithms (e.g. Viterbi's, Pearl's belief propagation, or Shafer-Shenoy algorithms). Therefore, GDL has a wide range of applicability. In our case, the rationale to apply (and extend) GDL is that a DCOP requires the maximization of a global function resulting from the combination of local functions. In order to ensure optimality and convergence GDL must arrange the global function to optimise into a junction tree structure (JT) [5].

*Partially funded by TIN2006-15662-C02-01. M.Vinyals is supported by the Ministry of Education of Spain (FPU grant AP2006-04636). JAR thanks JC2008-00337.

Cite as: Title, Author(s), *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, Decker, Sichman, Sierra and Castelfranchi (eds.), May, 10–15, 2009, Budapest, Hungary, pp. XXX-XXX.
Copyright © 2009, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

There are several works that have applied JTs (join trees or tree-decompositions) to the constraint optimization problem [3, 4, 6]. Indeed the cluster-tree elimination algorithm [6] is an instance of GDL algorithm applied over a tree decomposition (junction tree). However, to the best of our knowledge, all these approaches construct a JT using triangulation methods [5], which are not suitable when applied to problems that are distributed by nature because they produce JTs disreading the structure of the problem.

Therefore, one of our contributions in this paper is the Action-GDL algorithm that extends GDL by: (1) supporting the distribution of the problem using distributed junction tree structure (DJT) that maps cliques to agents; and (2) modifying the original GDL algorithm to create two phases which allows to reduce the size of one half of the messages.

To generate DJTs, we introduce the Distributed Junction Tree Generator (DJTG) algorithm at the pre-processing phase of Action-GDL. DJTG is a message-passing algorithm, based on the one formulated in [8], that allows agents to *distributedly* compile a DJT keeping any distribution of relations among agents. Therefore, DJTG creates a DJT that adapts to the underlying distributed nature of the problem. We will show how although finding the best junction tree has been shown to be NP-hard [5], this algorithm is general enough to exploit existing distributed heuristics in the literature [9, 2] to produce a DJTG to fed into Action-GDL.

Thereafter, we show that Action-GDL generalises DPOP, one of the low complexity, state-of-the-art algorithm to solve DCOPs. To do so, we: (1) prove that DPOP is a particular case of Action-GDL; and (2) show how Action-GDL can exploit DJTs as a more general structure to generate executions that cannot be achieved by DPOP via pseudotrees. Therefore, Action-GDL can efficiently solve DCOPs. To the best of our knowledge, we are the first in comparing and providing a mapping between the space of DJTs and the space of pseudotree arrangements. Finally we provide empirical evidence to show that the generality of Action-GDL can be exploited to outperform DPOP in terms of amount of computation, communication, and parallelism.

This paper is structured as follows. Firstly, we provide a definition of DCOP (section 2) and the notation we will use through out this paper (section 3). Section 4 introduces Action-GDL, as well as its connection with GDL and the DJTG algorithm that allow agents to compile the DCOP problem into a DJT. Then, in section 5 we show how Action-GDL extends DPOP, one of the state-of-the-art algorithms to solve DCOPs. Next, section 6 provides some evidences of how to exploit the generality of Action-GDL by showing how it can outperform DPOP when solving the same DCOPs. Finally, section 7 draws some conclusions and outlines paths for future research.

2. OVERVIEW OF DCOPS

Distributed Constraint Optimization Problem (DCOP) are an extension to the Constraint Optimization Problem (COP) that can model a large class of MCPs [9]. These problems consist of a set of variables, each one taking on a value out of a finite discrete domain. Each constraint in this context has a set of variables as input specifying a cost, namely a relation. The goal of a COP algorithm is to assign values to these variables so that the total utility is maximized. A DCOP [10, 7] is an extension to a COP where variables are distributed among agents.

Let $\mathcal{X} = \{x_1, \ldots, x_n\}$ be a set of variables over domains $\mathcal{D}_1, \ldots, \mathcal{D}_n$. Let $r : \mathcal{D}_r \rightarrow \Re^+$, where $\mathcal{D}_r$ is the projection of the joint domain space $\mathcal{D} = \mathcal{D}_1 \times \ldots \times \mathcal{D}_n$ over variables in the domain of r, be a *utility relation* that assigns a utility value to each combination of values of its domain variables. Formally, a DCOP is a tuple $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{R}, \alpha \rangle$ where: $\mathcal{A}$ is a set of agents; $\mathcal{X}$ is a set of variables; $\mathcal{D}_n$ is the joint domain space for all variables; $\mathcal{R} = \{r_1, \ldots, r_p\}$ is a set of utility relations; and $\alpha : \mathcal{X} \rightarrow \mathcal{A}$ maps each variable to some agent.

The objective function f is described as an aggregation (typically addition) over the set of relations. Formally:

$$f(d) = \sum_{i=1}^{p} r_i(d_{r_i}) \tag{1}$$

where d is an element of the joint domain space $\mathcal{D}$ and d_{r_i} is an element of $\mathcal{D}_{r_i}$. Solving a DCOP amounts to choosing values for the variables in $\mathcal{X}$ such that the objective function is maximized (minimized).

In a DCOP each agent receives knowledge about all relations that involve its variable(s) in addition to their domains. In general, DCOP algorithms do not impose any restriction regarding the number of variables that can be assigned to each agent or the arity of the relations. However, although all algorithms we refer to in this paper can deal with n-ary relations, for the sake of simplicity we mainly restrict them to unary and binary relations. Therefore, we will refer to unary relations involving variable $x_i \in \mathcal{X}$ as r^i, and to binary relations involving variables $x_i, x_j \in \mathcal{X}$ as r^{ij}.

A DCOP with binary relations is typically represented with its primal-constraint graph, whose vertices stand for variables and whose edges stand for binary relations, as shown by the example depicted in figure 1 (a). DCOPs can also be represented with its dual-constraint graph, whose vertices stand for relations and whose edges link relations that share some variable in their domains, as shown by the example depicted in figure 1 (b).

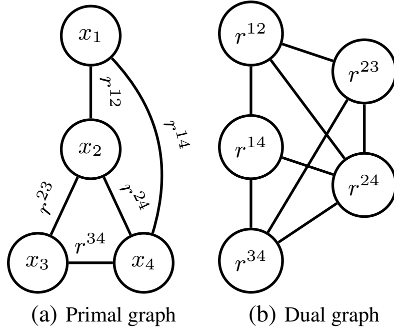


Figure 1: Different representations for the same DCOP

3. NOTATION

Next we provide the definitions of a collection of functions and operators that we shall employ throughout the rest of this paper. Henceforth, given some variable set $X \subseteq \mathcal{X}$, $\mathcal{D}_X$ will stand for the joint domain space of variables in X. Furthermore, for exemplary purposes we assume that each domain $\mathcal{D}_i$ contains constant values $c_i^1, \ldots, c_i^{n_i}$.

DEFINITION 1 (DV). *The domain variable function DV returns the domain variables of a given set of relations.*
Ex: $DV(\{r^{31}\}) = \{x_3, x_1\}$, $DV(\{r^{31}, r^{34}\}) = \{x_1, x_3, x_4\}$.

DEFINITION 2 (COMPLEMENTARY VARIABLES). *Given a set of variables X and a relation r, we define the complementary variables of X by r as the set of variables in r that are not in X. Formally, $\bar{X}^r = DV(r) \setminus X$.*

DEFINITION 3 (UTILITY MESSAGE). *A message from agent a_i to agent a_j is a utility message over $X \subseteq \mathcal{X}$, if the information sent is a utility relation over $\mathcal{D}_X$. Henceforth, we shall denote that utility relation as μ_{ij}.*

DEFINITION 4 (ASSIGNMENT). *Given a set of agents $X \in \mathcal{X}$, an assignment σ over X sets a value to each variable $x_k \in X$ and sets free the remaining variables. Given $Y \subset X$, we note by $\sigma[Y]$ the projection of σ to Y, that is, the assignment that sets the same value as σ for the variables in Y.*
Ex: $X = \{x_1, x_2, x_3\}$, σ *an assignment over* X, $\sigma(x_1) = c_1^2$, $\sigma(x_3) = c_3^5$, x_2 *is free in* σ
$Y = \{x_1\}$, $\sigma[Y](x_1) = c_1^2$, x_2 *and* x_3 *are free in* $\sigma[Y]$

DEFINITION 5 (VALUE MESSAGE). *A message from agent a_i to agent a_j is a value message over $X \subseteq \mathcal{X}$ if the information sent is an assignment over X. Henceforth, we shall denote such assignment by σ_{ij}.*

The joint operator is a combination operator that joins the knowledge represented by two relations into a single one by adding their values.

DEFINITION 6 (JOINT). *Let r, s be two relations and $\mathcal{D}_{r \otimes s} = \times_{x_k \in DV(\{r,s\})} \mathcal{D}_k$ be their joint domain space. The combination of r and s (noted $r \otimes s$) is a utility relation over $\mathcal{D}_{r \otimes s}$ such that $(r \otimes s)(d) = r(d_r) + s(d_s)$ for all $d \in \mathcal{D}_{r \otimes s}$, where $d_r \in \mathcal{D}_r$ and $d_s \in \mathcal{D}_s$ are the projections of d over the domains of relations r and s respectively.*
Ex: $(r^{13} \otimes r^{14})(c_1^2, c_3^5, c_4^1) = r^{13}(c_1^2, c_3^5) + r^{14}(c_1^2, c_4^1)$.

We can readily generalize the joint operator over a finite set of relations: $\bigotimes_{\{r_1, \cdots, r_m\}} = r_1 \otimes (r_2 \otimes \ldots (r_{m-1} \otimes r_m) \ldots)$.
The projection operator sums up the utility that a relation contains over a set of variables. Thus, the projection operator over a relation r and a set of variables Xassesses the r maximum utility for the variables in X.

DEFINITION 7 (PROJECTION). *The projection operator of relation r over a set of variables X is a summarization operator that returns a utility relation over $\mathcal{D}_X$ such that $(\bigoplus_X r)(d_X) = \max_{d_{\bar{X}^r} \in \mathcal{D}_{\bar{X}^r}} r(d_X, d_{\bar{X}^r})$.*
Ex: $(\bigoplus_{\{x_3\}} r^{13})(c_3^2) = \max_{k \in \mathcal{D}_1} r^{13}(k, c_3^2)$.

Notice that we can employ the projection operator by specifying the variables to eliminate from a relation as follows $\bigoplus_{\setminus X} r = \bigoplus_{\bar{X}^r} r = \bigoplus_{DV(r) \setminus X} r$.

DEFINITION 8 (SLICE). *The slice of a relation r by an assignment σ over X is a utility relation over $\mathcal{D}_{\bar{X}^r}$ such that $(\underset{\sigma}{\bigtriangledown} r)(d_{\bar{X}^r}) = r(d_X, d_{\bar{X}^r})$ where $d_X \in \mathcal{D}_X$ contains the values set by σ to the variables in X.*
Ex: $X = \{x_3\}$, $\sigma(x_3) = c_3^2$, $(\underset{\sigma}{\bigtriangledown} r^{13})(c_1^1) = r^{13}(c_1^1, c_3^2)$.

4. THE ACTION-GDL ALGORITHM

In this section we introduce the Action-GDL, a novel complete algorithm to efficiently solve DCOP's, an extension to GDL [1] to efficiently apply it to MAS decision making. We start by introducing GDL in the following section to subsequently propose Action-GDL in section 4.2. Since Action-GDL is executed over a distributed junction tree structure, for completeness, we proposed it to be combined with what we called the DJTG algorithm, an algorithm that allow agents to distributedly compile JTs initially proposed in [8] in the context of sensor networks. DJTG algorithm (section 4.3) allows agents to distributedly compile the DCOP into a DJT to which Action-GDL is executed over and drawbacks that other traditional methods to compile DJTs have been reported to have in distributed environments.

4.1 The GDL Algorithm

GDL [1] is a general message-passing algorithm that exploits the way a global function factors into a combination of local functions to compute the objective function in an efficient manner. GDL is defined over two binary operations [1] that in our case, since we are concerned with the problem of maximizing an utility function, correspond to the addition and the maximization (the max-sum GDL). In order to ensure optimality and convergence, GDL arranges the objective function to assess in a *junction tree structure* (JT)[5].

DEFINITION 9. *A **junction tree** (JT) is a tree of cliques that can be represented as a tuple $\langle \mathcal{X}, \mathcal{C}, \mathcal{S}, \Psi \rangle$ where: $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of variables; $\mathcal{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$ is a set of cliques such that each clique $\mathcal{C}_i \subseteq \mathcal{X}$; $\mathcal{S}$ is a set of separators, where each separator is an edge between two cliques containing the intersection of the cliques[1]; and $\Psi = \{\psi_1, \ldots, \psi_m\}$ is a set of potentials, where potential ψ_i is a function assigned to clique $\mathcal{C}_i$ with domain $\Delta_i \subseteq \mathcal{X}$. Furthermore, the following properties must hold:*
• ***Single-connectedness.*** *Separators create exactly one path between each pair of cliques.*
• ***Covering.*** *Each potential domain is a subset of the clique to which it is assigned, namely $\Delta_i \subseteq \mathcal{C}_i$.*
• ***Running intersection.*** *If a variable x_i is in two cliques $\mathcal{C}_i$ and $\mathcal{C}_j$, then it must also be in all cliques on the (unique) path between $\mathcal{C}_i$ and $\mathcal{C}_j$.*

Likewise variables in DCOP, we assume that the variables in a junction tree are defined over domains $\mathcal{D}_1, \ldots, \mathcal{D}_n$. Moreover, $\mathcal{D}_{\mathcal{C}_i}$ stands for clique $\mathcal{C}_i$ domain space, namely the joint domain space of the variables in clique $\mathcal{C}_i$.

Figure 2 shows a JT where circles stand for cliques, labelled with the variables each one contains, and edges between cliques stand for separators. Thus, for example, C_1 contains variables x_2, x_4; C_3 contains variables x_2, x_3, x_4; and their separator is composed of their intersection x_2, x_4. Each clique C_i is associated with a potential ψ_i, a function whose domain is a subset of C_i.

GDL defines a message-passing phase for cliques to exchange information about their variables. Once the message-passing phase is over, each clique can compute its state, namely its variables states. To illustrate the way the max-sum GDL operates, consider the following example. Say that our goal is to distributedly maximize some objective function $f(x_1, x_2, x_3, x_4) = \psi_1(x_2) + \psi_2(x_1, x_2, x_4) + \psi_3(x_2, x_3, x_4)$, whose factors ($\psi_1$, ψ_2 and ψ_3) are arranged in the directed JT of figure 2. Since the JT is directed, messages are sent in two different message-passing phases: (i) one up the tree in which each clique sends a message to its clique parent when, for the first time, it has received messages from all of its children; (ii) one down the tree so that each clique sends a message to its children when it receives a message from its parent.

[1]Formally, a separator s^{ij} between clique $\mathcal{C}_i$ and $\mathcal{C}_j$ is defined as $s^{ij} = \mathcal{C}_i \cap \mathcal{C}_j$.

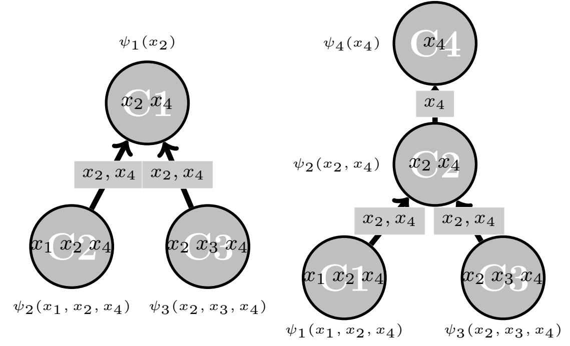


Figure 2: JT **Figure 3: DJT**

#	Message/local knowledge ($\widehat{\mathcal{K}}$)
1.	$\mu_{21}(x_2, x_4) = max_{\{x_1\}} \psi_2(x_1, x_2, x_4)$
2.	$\mu_{31}(x_2, x_4) = max_{\{x_3\}} \psi_3(x_2, x_3, x_4)$
3.	$\widehat{\mathcal{K}}_1(x_2, x_4) = \psi_1(x_2) + \mu_{21}(x_2, x_4) + \mu_{31}(x_2, x_4)$
4.	$\mu_{12}(x_2, x_4) = \psi_1(x_2) + \mu_{31}(x_2, x_4)$
5.	$\mu_{13}(x_2, x_4) = \psi_1(x_2) + \mu_{21}(x_2, x_4)$
6.	$\widehat{\mathcal{K}}_2(x_1, x_2, x_4) = \psi_2(x_1, x_2, x_4) + \mu_{12}(x_2, x_4)$
7.	$\widehat{\mathcal{K}}_3(x_2, x_3, x_4) = \psi_3(x_2, x_3, x_4) + \mu_{13}(x_2, x_4)$

Table 1: Trace of GDL over the JT of Fig. 2

At round 1, clique $\mathcal{C}_2 = \{x_1, x_2, x_4\}$ sends a message μ_{21} to clique $\mathcal{C}_1 = \{x_2, x_4\}$ with the values of its local function, ψ_2, after 'filtering out' dependence on all variables but those common to $\mathcal{C}_2$ and $\mathcal{C}_1$ (namely variables which are not in their separator). At round 3, after clique $\mathcal{C}_1$ receives the values of its children's local functions for its variables x_2, x_4, it combines those values into $\widehat{\mathcal{K}}_1$. $\widehat{\mathcal{K}}_1$ is a function that stands for $\mathcal{C}_1$ knowledge over its variables, namely x_2, x_4. At that point, since $\mathcal{C}_1$ has received messages from all its neighbors, $\widehat{\mathcal{K}}_1$ contains all the information related to x_2, x_4. At rounds 4 and 5, clique $\mathcal{C}_1$ sends messages to its children that contain the combination (joint operation) of its local function, ψ_1, with other children messages. Thus, $\mathcal{C}_2$ receives a message from $\mathcal{C}_1$ that contains the potential ψ_1 combined with μ_{31}. Then it can compute $\widehat{\mathcal{K}}_2$(round 6).

4.2 Extending GDL to solve DCOPs

Recall that our goal is to solve MCPs represented as DCOPs. Therefore, the capability of computing any objective function, as provided by GDL, is not enough. We need to go one step beyond GDL to allow a group of agents make a joint decision (regarding their variables' values) that maximizes any objective function. For this purpose, Action-GDL extends GDL by: (1) inferring decision variables; and (2) supporting the distribution of the problem through the use of a distributed junction tree structure.

Inferring decision variables Consider a DCOP setting. As explained above in GDL, when a clique has received messages from all its neighbors, it has all information related to its variables and it can compute its objective function. In DCOPs, clique variables are decision variables and computing a clique objective function stands

Algorithm 1 Action-GDL($\langle \mathcal{A}, \mathcal{X}, \mathcal{C}, \mathcal{S}, \Psi, \beta \rangle$)

Each agent $a \in \mathcal{A}$ for each one of its cliques $\mathcal{C}_i$ starts with $\langle \widehat{P}(\mathcal{C}_i), \widehat{Ch}(\mathcal{C}_i), \mathcal{C}_i, \psi_i, \widehat{S}(\mathcal{C}_i), \beta_i \rangle$ and runs:

1: **Phase I: UTILITY Propagation**
2: $\widehat{\mathcal{K}}_i = \psi_i$
3: **for all** $\mathcal{C}_j \in \widehat{Ch}(\mathcal{C}_i)$ **do**
4: Wait for utility message μ_{ji} from $\mathcal{C}_j$'s agent (that is $\beta_i(\mathcal{C}_j)$)
5: $\widehat{\mathcal{K}}_i = \widehat{\mathcal{K}}_i \otimes \mu_{ji}$
6: **end for**
7: **if** $\mathcal{C}_i$ is not the tree's root, let $\mathcal{C}_p = \widehat{P}(\mathcal{C}_i)$ **then**
8: Let $s^{ip} \in \widehat{S}(\mathcal{C}_i)$ be the separator between i and its parent
9: Send $\mu_{ip} = \bigoplus_{s^{ip}} \widehat{\mathcal{K}}_i$ to $\mathcal{C}_p$'s agent (that is $\beta_i(\mathcal{C}_p)$)
10: **end if**
11: **Phase II: VALUE propagation**
12: **if** $\mathcal{C}_i$ is not the tree's root, let $\mathcal{C}_p = \widehat{P}(\mathcal{C}_i)$ **then**
13: Wait for a value message σ_{pi} from $\mathcal{C}_p$'s agent (that is $\beta_i(\mathcal{C}_p)$)
14: $\widehat{\mathcal{K}}_i = \bigtriangledown_{\sigma_{pi}} \widehat{\mathcal{K}}_i$; /*Slice $\widehat{\mathcal{K}}_i$ with the value message*/
15: **end if**
16: $d^* = arg \max_{d \in \mathcal{D}_{DV(\widehat{\mathcal{K}}_i)}} \widehat{\mathcal{K}}_i$; /*Fix the values for the free variables*/
17: $d^*_{\mathcal{C}_i} = d^* \cup \sigma_{pi}$; /*Put together the assesed values and the value message received. Assume the root gets an empty value message*/
18: **for all** $\mathcal{C}_j \in \widehat{Ch}(\mathcal{C}_i)$ **do**
19: Let $\sigma_{ij} = d^*_{\mathcal{C}i}[s_{ij}]$; /*Project into the separator*/
20: Send σ_{ij} to $\mathcal{C}_j$'s agent (that is $\beta_i(\mathcal{C}_j)$)
21: **end for**
22: **return** $d^*_{\mathcal{C}_i}$;

for assigning values to these decisions. Therefore, when a clique infers their state, there is no need to propagate more information related to its variables since we can propagate directly the decisions taken. In other words, there is no need to propagate messages containing relations down the tree because all a child requires to make a decision is its father's decisions (variables' assignments). It implies that in a DCOP, when the first message-passing phase of GDL, up to the tree, is over, the second message-passing phase of GDL, down the tree, is no longer necessary. Thus, we require a second message-passing phase for cliques to exchange *decisions* down the tree, which is precisely the extension that Action-GDL introduces. Henceforth, we shall refer to the first message-passing phase as *utility propagation*, and to the second one as *value propagation*. It is relevant to notice that the value propagation phase ensures that whenever multiple optimal joint decisions are feasible, cliques converge to the very same joint decision, namely to the very same solution of a DCOP.

Table 2 displays a trace of Action-GDL over the JT in figure 2. Notice that by making $\Psi = \{\psi_1 = r^2, \psi_2 = r^{12} \otimes r^{14} \otimes r^{24}, \psi_3 = r^{23} \otimes r^{34}\}$ the function encoded in the JT ($f(x_1, x_2, x_3, x_4) = \psi_1(x_2) + \psi_2(x_1, x_2, x_4) + \psi_3(x_2, x_3, x_4)$) of is the same as the constraint graph of figure 1(a), so we are maximizing a DCOP function.

Steps 1-4 are equivalent to steps 1-3 in GDL. However, at step 5 the root clique assesses the optimal value for x_2, x_4 ($x_2^* = c_2^*, x_4^* = c_4^*$) and propagates these values down the tree through value messages to cliques $\mathcal{C}_2$ and $\mathcal{C}_3$ (steps 6 and 7). At steps 8-9 and 10-11, $\mathcal{C}_2$ and $\mathcal{C}_3$ assess the values of x_1 and x_3, respectively, using its parent decision values (c_2^*, c_4^*).

Supporting the distribution of the problem. Another major difference between Action-GDL and GDL has to do with the way they solve a problem. GDL runs over a JT as formalised by definition 9. Hence, all cliques are considered to be located in a single agent, which is in charge of running GDL. Action-GDL solves a

#.	Messages/local knowledge $\widehat{\mathcal{K}}$	#.	Messages/local knowledge $\widehat{\mathcal{K}}$
1.	$\mu_{21}(x_2, x_4) = \max_{\{x_1\}} \psi_2(x_1, x_2, x_4)$	7.	$\sigma_{13}(x_2, x_4) = (c_2^*, c_4^*)$
2.	$\widehat{\mathcal{K}}_1(x_2, x_4) = \psi_1(x_2) + \mu_{21}$	8.	$\widehat{\mathcal{K}}_2(x_1) = \psi_2(x_1; \sigma_{12})$
3.	$\mu_{31}(x_2, x_4) = \max_{\{x_3\}} \psi_3(x_2, x_3, x_4)$	9.	$c_1^* = arg \ \max_{\{x_1\}} \widehat{\mathcal{K}}_2$
4.	$\widehat{\mathcal{K}}_1(x_2, x_4) = \widehat{\mathcal{K}}_1(x_2, x_4) + \mu_{31}$	10.	$\widehat{\mathcal{K}}_3(x_3) = \psi_3(\sigma_{13}; x_3)$
5.	$(c_2^*, c_4^*) = arg \ \max_{\{x_2, x_4\}} \widehat{\mathcal{K}}_1$	11.	$c_3^* = arg \ \max_{\{x_3\}} \widehat{\mathcal{K}}_3$
6.	$\sigma_{12}(x_2, x_4) = (c_2^*, c_4^*)$		

Table 2: Trace of Action-GDL over the JT of Fig. 2

DCOP where variables and relations are distributed over agents that cooperatively solve the problem. Therefore, Action-GDL extends GDL to deal with cliques that are distributed to different agents and control that agents have knowledge about the local information (potential) related to its cliques. This is accomplished by running Action-GDL over a *distributed junction tree* (DJT). Formally:

DEFINITION 10. *A* ***distributed junction tree*** *(DJT) is a tuple $\langle \mathcal{A}, \mathcal{X}, \mathcal{C}, \mathcal{S}, \Psi, \beta \rangle$ where $\langle \mathcal{X}, \mathcal{C}, \mathcal{S}, \Psi \rangle$ is a JT; $\mathcal{A} = \{a_1, \ldots, a_m\}$ is a set of agents; and $\beta : \mathcal{C} \rightarrow \mathcal{A}$ maps each clique to one agent.*

We define $\widehat{N}(\mathcal{C}_i) = \{\mathcal{C}_j | s^{ij} \in \mathcal{S}\}$ as a function that returns the cliques connected by a separator to clique $\mathcal{C}_i$, namely its neighboring cliques. Since a DJT is a tree of cliques it can also be defined as a directed tree. In a directed DJT we define two additional relationships among cliques: $\widehat{P}(\mathcal{C}_i)$, which returns the parent of $\mathcal{C}_i$; and $\widehat{Ch}(\mathcal{C}_i)$, which returns the children of $\mathcal{C}_i$.

An example of DJT is given in figure 3. Likewise JTs, circles stand for cliques, and edges for separators, both labelled with their variables' indices. This DJT has 4 cliques, one for each agent of the DCOP of figure 1(a) (clique C_i is assigned to agent a_i). The set of potentials contains the set of relations of the DCOP distributed as follows: $\psi_1 = r^{12} \otimes r^{14}$,$\psi_3 = r^{23} \otimes r^{34}$,$\psi_2 = r^{24}$,$\psi_4 = \{\}$,$\psi_5 = r^{15}$. Notice that this DJT has the property that agents are assigned a clique whose potential contains relations that this agent knows (in DCOP relations that contains some agent's variable). Thus, agent 1 is assigned clique 1 whose potential contains relations that include variable x_1, namely r^{12}, r^{14}. That is not true in the JT of figure 2 since in that case there is not a single agent who knows all relations assigned to potential ψ_2, namely r^{12}, r^{14}, r^{24}.

Algorithm 1 outlines Action-GDL. Given a DJT $\langle \mathcal{A}, \mathcal{X}, \mathcal{C}, \mathcal{S}, \Psi, \beta \rangle$, each agent $a \in \mathcal{A}$ involved in Action-GDL, only needs to know the subset of the DJT that involves the cliques it has assigned. Hence, every agent is assumed to start knowing a tuple $\langle \widehat{P}(\mathcal{C}_i)$, $\widehat{Ch}(\mathcal{C}_i)$,$\psi_i$,$\widehat{S}(\mathcal{C}_i)$,$\beta_i \rangle$ for each one of its cliques $\mathcal{C}_i$, where $\widehat{S}$ returns a clique's separators ($\widehat{S}(\mathcal{C}_i) = \{s^{ik} | s^{ik} \in \mathcal{S}\}$), and β_i returns the agents assigned to clique $\mathcal{C}_i$'s parent or children.

During the utility propagation phase (lines 1-10), agents exchange utility messages. The initial knowledge for each clique is its potential (line 2). For each clique, its agent waits until receiving a utility message from each of its children cliques (lines 3-4). These messages contain a utility relation over the variables shared by both cliques (their separator) and are sent by agents assigned to the children cliques. Every time that the agent receives a new utility message, it incorporates it (by using the combination operator) to its local knowledge (line 5). After combining utility messages from all the children of a clique, if that clique has a parent (line 7), its agent summarizes that part of its local knowledge (using the projection operator) that is of interest to the clique's parent (by means of a utility relation over its separator) and sends it to the agent associated to the parent's clique (line 9).

During the value propagation phase (lines 11-21), agents compute the optimal values for their variables and exchange value messages, namely decisions. Given a clique, its agent waits until receiving a value message (containing value assignments) for all variables in common (in the separator) with its clique parent (line 12-

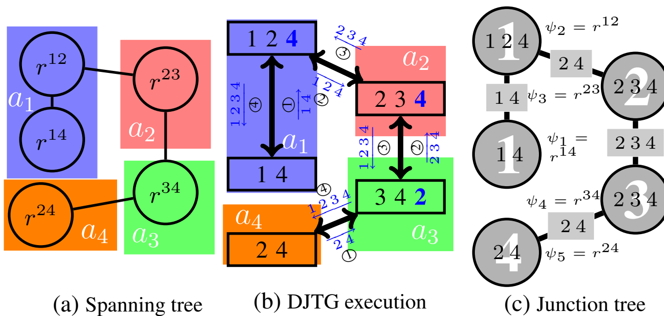
13). At that point, the agent has received all the knowledge, in form of utility (from children) and value (from the parent) messages, required for computing the objective function related to its clique variables. The agent slices its knowledge by incorporating the already inferred decisions (line 14) and computes the optimal values for the rest of its clique variables (line 16). Once an agent knows the variables' values for one of its cliques, it can propagate them down the tree (lines 18-21). Notice however that it only propagates variable assignments that are required by its children cliques, namely assignments for variables in their separator.

Since Action-GDL runs over a DJT, we can readily assess its computation and communication complexity from cliques' and separators' sizes after [1, 8]. Action-GDL requires a number of messages linear to the number of edges in the DJT (exchanging one value message and one utility message per separator). The communication complexity lies in the size of utility messages, which is exponential to separators' sizes, because the size of value messages is linear. Regarding the computation required by each agent to build messages and assess variables' values, it also scales with its cliques' sizes.

4.3 The DJTG algorithm

As explained above, Action-GDL runs over a DJT (as given by definition 10). It has been argued [9] that traditional methods to compile JTs, that is triangulation methods based on the one proposed in [5], are not suitable when applied to problems that are distributed by nature because they produce JTs disregarding the structure of the problem. Thus, cliques in such methods are created independently of the number of agents and their knowledge and therefore since the mapping agent-clique is not clear it can appear a clique that has knowledge about a particular relation/variable that it's hosting agent does not know. Therefore, here we propose to use an alternative algorithm, the so-called Distributed Junction Tree Generator (DJTG) that distributedly compiles a DJT, by exchanging a linear number of messages, that captures the distribution of relations required by the problem. Such DJT can be readily fed into, and hence solved by Action-GDL.

The DJTG algorithm receives as input a set of relations distributed among agents and an spanning tree, $\mathcal{ST}$, defined over them. Figure 4(a) illustrates an input to DJTG. Observe that relation r^{12}



(a) Spanning tree (b) DJTG execution (c) Junction tree

Figure 4: DJTG execution

is assigned to agent a_1 and it is linked to relations r^{23} and r^{14} by edges in the $\mathcal{ST}$.

DJTG has two phases: 1) a pre-processing phase where agents create a distributed clique tree that may not satisfy the running intersection property (RIP); followed by 2) a message-passing phase that calculates the unique set of minimal cliques that satisfy the RIP.

In the pre-processing phase, each agent a creates a clique $\mathcal{C}_j$ for each one of its relations r_j and sets as potential the relation itself, namely $\psi_j = r_j$. Cliques are initially set to their potential domain, namely $C_i = \Delta_i$, in order to readily ensure the covering property. Moreover, for every two relations r_i, r_j connected in the $\mathcal{ST}$, agents create a separator s^{ij} linking their corresponding cliques $\mathcal{C}_i$ and $\mathcal{C}_j$. Figure 4 (b) shows the structure produced by the pre-processing phase. Boxes stand for cliques containing numbers that stand for variables' indices. Cliques are assigned to agents. For instance, clique $\mathcal{C}_1$ with variables x_1, x_4 and clique $\mathcal{C}_2$ with x_1, x_2 are assigned to a_1. The variables correspond to the domains of the cliques' potentials, namely the domains of r^{14} and r^{12} respectively. Cliques $\mathcal{C}_1$ and $\mathcal{C}_2$ are connected as their relations r^{14} and r^{12} in the $\mathcal{ST}$.

The second phase of DJTG is responsible for ensuring the RIP. In that phase, each agent exchanges for each one of its cliques, $\mathcal{C}_i$, *reachability* messages with agents related to $\mathcal{C}_i$'s neighbors that contain the set of reachable variables from $\mathcal{C}_i$. Figure 4(b) shows the messages exchanged. Single-directed arrows between boxes stand for messages exchanged between cliques. Each arrow is labelled with some variables' indices and a circled number standing for the order of the message in the message-passing execution. The set of reachable variables from a clique $\mathcal{C}_i$ to $\mathcal{C}_j$ is calculated as the union of: (i) $\mathcal{C}_i$'s potential domain; and (ii) the variables reachable from $\mathcal{C}_i$'s neighbours other than $\mathcal{C}_j$. Thus, agent a_1 sends a message to agent a_2 for clique $\mathcal{C}_3$ that contains variables (x_1, x_2, x_4), namely the variables that can be reached from clique $\mathcal{C}_2$. These variables are the result of the union of $\mathcal{C}_2$ potential domain, namely (x_1, x_2), with the reachable variables from $\mathcal{C}_2$, namely (x_1, x_4). Once an agent receives, for a given clique, reachability messages from all its neighbours, it redefines its clique adding variables that are in more than one *reachability* message. In figure 4(b) agent a_2 receives two reachability messages for clique $\mathcal{C}_3$: one with (x_1, x_2, x_4) from clique $\mathcal{C}_2$ associated to a_1, another one with (x_2, x_3, x_4) from clique $\mathcal{C}_5$ associated to a_3. Since both messages contain x_4, agent a_2 knows that its clique $\mathcal{C}_3$ must also carry x_4 to satisfy the RIP. After computing cliques, it is straightforward to assess separators (see definition 9). Finally, figure 4(c) depicts the DJT as produced by DJTG from the initial distribution of relations in figure 4(a). Notice that by creating a clique per relation and by assigning each clique to the agent associated to that relation, DJTG manages to preserve the initial distribution of the problem.

This alternative way of building a JT, by directly ensuring the RIP over a set of relations was initially formulated in [8] in the context of sensor networks. However, they restricted each agent to control a single clique whose potential results from the combination of relations located to the agent. DJTG extends the algorithm in [8] to: (1) allow each agent to be associated to more than one clique; and (2) accept as input a spanning tree defined over some set of relations, without making any assumptions on their composition.

Given a set of n relations, there are n^{n-2} different spanning trees that we can define over them, and for each one we can compile the associated DJT with the DJTG algorithm. It is know from [5] that finding the optimal JT is NP-hard, so it is reasonably to wonder what we can do to find good spanning trees to use as input for the DJTG algorithm. However it turns out that existing heuristics proposed in the literature for DCOP problems and DJT construction can be expressed, explicitly or implicitly, as a set of relations connected by a spanning tree that we can use as input of the DJTG. On the one hand, there are heuristics [8] that directly assess an spanning tree defined over the dual-constraint graph and we can readily exploit them. On the other hand, there are heuristics that define a spanning tree, or a subclass of them like pseudotrees, over the primal-constraint graph such as those proposed in [9, 2]. These heuristics associate each relation to the lowest variable of its do-

main in the tree structure. We can combine the relations associated to the very same variable to create a single relation. Since in these approaches variables are connected by an spanning tree, so are the combined relations and we can use this spanning tree as input to the DJTG.

5. GENERALITY OF ACTION-GDL

Action-GDL has a lot of straightforward similarities with DPOP [10], one of the state-of-the-art algorithms to solve DCOPs. Indeed DPOP was inspired by the sum-product algorithm, a GDL iterative version when applied directly to the original constraint graph and that is only guarantee convergence and optimality in acyclic graphs. DPOP ensures optimality and convergence in general graphs by arranging the DCOP into a pseudotree whereas Action-GDL arranges the problem in a DJTs.

Both algorithms, Action-GDL and DPOP have two similar phases when agents exchange same type of messages (UTIL and VALUE phases) and are based on the same operators: the combination and the projection operators. Despite of all these similarities, there are a set of important differences among them [2] namely:

(1) DPOP requires the DCOP to be arranged into a pseudotree structure with a particular distribution of relations (relations are associated to the lowest variable of its domain in the pseudotree) whereas Action-GDL is executed over a DJT. Figure 5(a) illustrates a pseudotree for the DCOP of figure 1(a). By definition of pseudotree[10], variables in different branches can not have direct dependencies among them. Thus, variables x_1 and x_3 are independent (there is no relation r^{13}) and relations r^{12},r^{14} are placed on x_1 (agent 1).

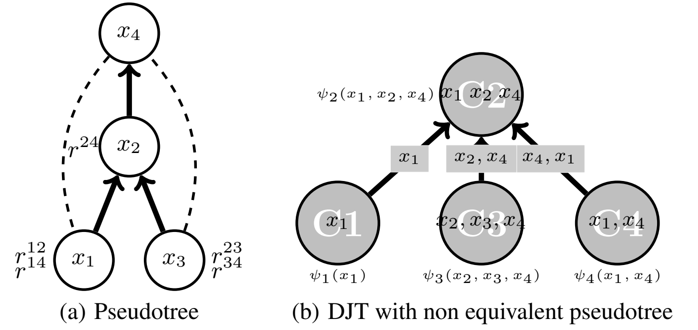
(2) In DPOP when agents create an utility message they use the summarize operator by summarizing out its own variable whereas in Action-GDL agents summarize over variables in the separator. Thus, during DPOP execution agent 1 (assigned to x_1) *filters out* x_1 from its local factor r^{12},r^{14} sending a message to agent 2 (assigned to x_2) that depends on the remaining variables, namely x_2 and x_4.

(3) In DPOP variables are inferred always one by one, in their position in the pseudotree, whereas in Action-GDL multiple variables can be inferred at once in the first agent that contains all the information related to these variables. Thus, during DPOP execution over the pseudotree of figure 5(a), agent 4 (responsible of x_4) infers its variable sending its value down to the tree. Same applies for the rest of variables.

However, if we want to compare DPOP and Action-GDL when applied to the same DCOP, we have to use equivalent arrangements for both algorithms. Thus we need to define a mapping between pseudotrees and DJTs. We will prove that given a pseudotree we can always define its equivalent DJT such that the execution of Action-GDL over such DJT is equal to the execution of DPOP over the pseudotree arrangement.

For example, the equivalent DJT of the pseudotree depicted in figure 5(a) is the DJT of figure 3. Notice that in that DJT relations are placed exactly as in the pseudotree arrangement and when cliques send messages summarizing over the separator, they *filter out* a single variable. Thus, DPOP execution over this pseudotree arrangement and Action-GDL execution over the DJT are equal.

[2]Here we list a set of similarities and differences respect to DPOP. We refer the reader to [10][9] for a detailed description of the DPOP algorithm



(a) Pseudotree (b) DJT with non equivalent pseudotree

Figure 5: Example of a pseudotree and DJT with non-corresponding pseudotree arrangement for the DCOP depicted in figure 1

However, it turns out that the other way around, that given a DJT we can always define an equivalent pseudotree arrangement, is not true. Thus, given an Action-GDL execution over a DJT, the equivalent pseudotree may not exist. It is because, given a DCOP, the space of all possible pseudotrees arrangements map with a subspace of all possible DJTs. DPOP equations are a particular case of Action-GDL equations when it is executed over this subclass of DJTs. We illustrate this with an example. Take the DJT of figure . By making $\Psi = \{\psi_1 = , \psi_2 = r^{12} \otimes r^{24}, \psi_3 = r^{23} \otimes r^{34}, \psi_4 = r^{41}\}$ the function encoded is the same as the constraint graph of figure 1(a). However, this DJT can not have any equivalent pseudotree because multiple variables are eliminated at once (clique 2 infers variables x_1,x_2 and x_3) and there are cliques that do not eliminate any variable when summarizing over the separator, namely C_1 and C_4.

In what follows we provide an sketch of the proof of the equivalence between Action-GDL and DPOP (fully detailed in [11]). Next, in section 6 we will provide empirical evidence of how we can exploit DJTs, as a more general structure, to improve the problem solving cost with respect to DPOP.

5.1 Proving equivalence

In order to prove equivalence between Action-DGL and DPOP, we first define a mapping, which we shall name γ, that builds a DJT from each pseudotree. Then we prove (lemma 1) that both the computation performed and the messages exchanged during the utility propagation phase are the same. After that, we prove (lemma 2) that the messages exchanged during the value propagation phase are also the same. Finally, we prove that the algorithm DJTG can compute this mapping distributedly. Because of lack of space, in the following we just expose the results and sketching proofs. The interested reader can find far more detailed proofs in [11].

LEMMA 1. *Given a DCOP* Φ *and a pseudotree* PT*, the computation performed and the messages exchanged by each agent during the utility phase of* $DPOP(\Phi,\text{PT})$ *and Action-GDL(*$\gamma(\Phi,\text{PT})$ *) are the same.*

PROOF. We prove the lemma by induction on the depth of the agent in the pseudotree. Both in the base and induction cases, we can prove that: (i) the set of variables handled by agents in both algorithms are the same; and (ii) the domain of the utility messages send by agents in DPOP after eliminating its corresponding variable coincides with separators in Action-GDL. By induction the utility messages received by each agent in both algorithms are the same. This fact along with (i) and (ii) forces that the computa-

tion performed and messages exchanged during this phase by each agent must be the same. □

LEMMA 2. *Given a DCOP Φ and a pseudotree* PT *the value assigned by each agent to its variable and the messages exchanged during the value propagation phase of $DPOP(\Phi$,PT$)$ and Action-GDL($\gamma(\Phi, \text{PT})$) are the same*

PROOF. We prove the lemma by induction on the depth of the pseudotree. The base case is trivial since there is only one variable in the pseudotree and both algorithms compute the same value for it. In the induction case we can split our pseudotree into the root and a set of pseudotrees of smaller depth. Then: (i) it is easy to see that the root agent acts equivalently in DPOP and in Action-GDL; and (ii) we can apply the induction hypothesis to the pseudotrees of smaller depth. Our result comes from (i) and (ii). □

Lemmas 1 and 2 prove the main result of this section:

THEOREM 1. *Given a DCOP Φ and a pseudotree* PT*, the execution of DPOP(Φ,* PT *) is equivalent to Action-GDL($\gamma(\Phi$,* PT *)).*

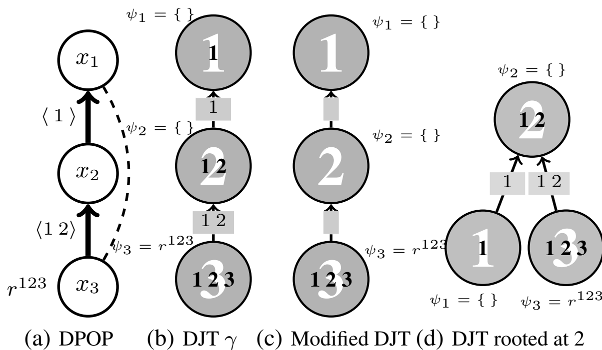
Theorem 1 shows that Action-GDL can be at least as efficient as DPOP in any DCOP problem. In the next section we introduce the DJTG algorithm that distributedly computes mapping γ at a cost that is small with respect to the cost of solving the problem.

THEOREM 2. *Given a DCOP Φ and a pseudotree $\langle P, PP \rangle$, the DJTG algorithm creates the DJT given by $\gamma(\Phi, \langle P, PP \rangle)$*

Therefore, DJTG computes the mapping γ at a cost that is small with respect to the cost of solving the problem. This two results prove that Action-GDL can be at least as efficient as DPOP (by mimicking its behavior).

6. EXPLOITING ACTION-GDL

At this point we have learned that Action-GDL generalises DPOP. It is now reasonable to wonder about the benefits that such generality delivers. In what follows we argue that Action-GDL can yield better algorithmic performance than DPOP. Action-GDL can achieve such improvement because: (i) DJTs allow to explore problem arrangements that cannot be represented via pseudotrees; and (ii) it can assess multiple variables' values at once. To show the benefits of Action-GDL with respect to DPOP, we empirically compare the computation and communication costs of both algorithms when solving the same DCOP. Moreover, we also compare the maximum degree of parallelism each algorithm can achieve.



(a) DPOP (b) DJT γ (c) Modified DJT (d) DJT rooted at 2

Figure 6: Example of experimented rearrangements

Our first experiment is aimed at showing the communication and computation savings achieved by Action-GDL with respect to DPOP. Such savings are obtained by adequately transforming the problem arrangement represented by a pseudotree. Consider the example depicted in figure 6. Figure 6(a) shows a pseudotree for a DCOP composed of a single ternary relation r^{123}. Observe that although variables x_1, x_2 do not have any relation, since DPOP can only eliminate variables one by one, its execution would propagate utility messages over these variables. Figure 6(b) depicts the DJT produced by mapping γ when applied to the pseudotree. Hence, according to theorem 1 the execution of Action-GDL over this DJT and the execution of DPOP over the pseudotree are equivalent. However, we can further transform the DJT in figure 6(b) to obtain savings. Notice that to make Action-GDL and DPOP executions equivalent,the definition of mapping γ (from pseudotree to DJT) *forces* that each clique in the equivalent DJT contains its variable although it is not part of its potential domain. If we do not enforce such constraint, the DJTG algorithm generates the DJT of figure 6(c), which can be regarded as a rearrangement of the one in figure 6(b). When running over the DJT in figure 6(c), Action-GDL does not need to exchange any utility messages, reducing the computation required to solve the problem. Hence there is a rationale for the rearrangement that we propose. Notice that when running Action-GDL, a variable' value is assessed at the clique that concentrates all its information. In figure 6(c), the values of x_2, x_3 are assessed at the clique containing x_1, x_2, x_3. That clique is in charge of propagating its decisions. Hence, there is no need to propagate utility messages involving x_1 and x_2 up the tree.

Next we compare the size of the messages exchanged and the amount of computation required by Action-GDL and DPOP when solving the same DCOP as the number of variables grows. Given a number of variables $n \in \{10, 30, 50, 70, 90\}$, we generate 2000 DCOPs, each one with $1.5 \cdot n$ constraints whose arity is randomly picked from 2 to 4. We create pseudotrees for DPOP using the DFS-MCN heuristic [9] and DJTs for Action-GDL considering the rearrangement of the DJT produced by mapping γ as explained above. Figure 7 (upper) shows the average savings (in percentage) in communication and computation of Action-GDL with respect to DPOP[3]. Observe that a simple rearrangement of the DJT leads to significant savings in communication and computation costs, which increase as the number of variables grows.

In our second experiment we show that we can help Action-GDL to reduce the maximum degree of parallelism with respect to DPOP. We propose to found such improvement on another rearrangement of the DJT produced by mapping γ. This time we propose to change the root of the DJT. Figure 6(d) illustrates such rearrangement for the DJT in figure 6(b). Observe that changing the root of a DJT never changes either the computation or the communication costs because cliques and separators remain the same. Notice also that we cannot explore such an arrangement in DPOP because changing the root of a pseudotree can lead to a non-valid pseudotree. For instance, choosing x_2 as a root in the pseudotree of figure 6(a) makes it an non-valid pseudotree (because of the dependency between variables x_1 and x_3). Next we measure and compare the MPC, formally defined as $MPC = \max_{P_i \in P} \sum_{C_j \in P_i} d^{|C_j|}$, where P stands for the set of all paths, a path P_i contains all cliques from the i-th clique to the root, and d stands for the variable domain size. To run this experiment we employ the same pseudotrees generated for our first experiment above and we set $d = 2$. We rearrange DJTs for Action-GDL as explained above to select as clique root the one that reduces the MPC the most. Figure 7 (lower) shows our empirical results by depicting the average (in percentage) improvement in MPC[3] that Action-

[3]Percentage assessed as $\frac{(DPOP - ActionGDL)}{DPOP} \cdot 100$

GDL achieves. Observe that the gain in parallelism can be very significant (from 25% to 40% of MCP reduction), and it increases as the number of variables grows.

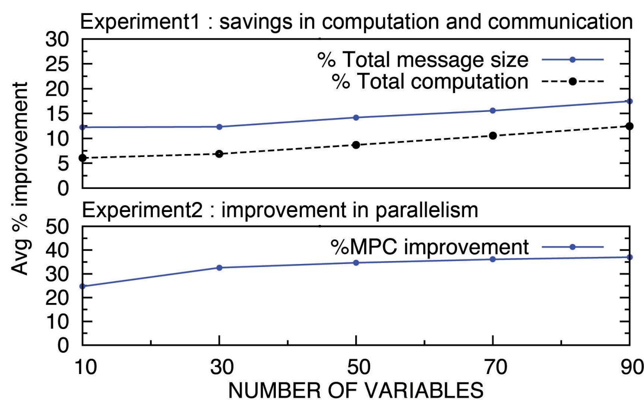


Figure 7: Experimental results

7. CONCLUSIONS AND FUTURE WORK

We made three main contributions in this paper. Firstly, we presented a new algorithm, the so-called Action-GDL, as an extension to GDL [1] to efficiently solve DCOPs. Secondly, we introduced the Distributed Junction Tree Generator (DJTG) algorithm, which allows agents to distributedly compile a distributed junction tree over which Action-GDL can operate. Finally, we show that Action-GDL generalizes DPOP. To do so, we prove that: (1) DPOP is a particular case of Action-GDL; and (2) Action-GDL can exploit distributed junction trees as a more general structure to generate executions that cannot be achieved by DPOP via pseudotrees. Moreover, we provide empirical evidence to show how we can computationally exploit the generality of Action-GDL. Thus, we show that Action-GDL can outperform DPOP in terms of the amount of computation, communication and parallelism of the algorithm solving cost. Finally, we argue that there are also analitical reasons to prefer Action-GDL. Since it is based on GDL, we can benefit from a wealth of theoretical results for GDL over junction trees and other approximate or more general structures such as single-cycle junction graphs [1].

8. REFERENCES

[1] S. M. Aji and R. J. McEliece. The generalized distributive law. *IEEE Transactions on Information Theory*, 46(2):325–343, 2000.

[2] J. Atlas and K. Decker. A complete distributed constraint optimization method for non-traditional pseudotree arrangements. In *AAMAS*, page 111, 2007.

[3] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artif. Intell.*, 34(1):1–38, 1987.

[4] R. Dechter and J. Pearl. Tree clustering for constraint networks (research note). *Artif. Intell.*, 38(3):353–366, 1989.

[5] F. V. Jensen and F. Jensen. Optimal junction trees. In *UAI*, pages 360–366, 1994.

[6] J. L. K. Kask, Rina Dechter and A. Dechter. Unifying cluster-tree decompositions for reasoning in graphical models. *Artif. Intell.*, 2005.

[7] P. J. Modi, W.-M. Shen, M. Tambe, and M. Yokoo. Adopt: asynchronous distributed constraint optimization with quality guarantees. *Artif. Intell.*, 161(1-2):149–180, 2005.

[8] M. A. Paskin, C. Guestrin, and J. McFadden. A robust architecture for distributed inference in sensor networks. In *IPSN*, pages 55–62, 2005.

[9] A. Petcu. *A Class of Algorithms for Distributed Constraint Optimization*. PhD thesis, EPFL, Lausanne, 2007.

[10] A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. In *IJCAI*, pages 266–271, 2005.

[11] M. Vinyals, J.A.Rodriguez-Aguilar, and J. Cerquides. Proving the equivalence of action-gdl and dpop. Technical report, IIIA-CSIC, 2008. Available at `http://www2.iiia.csic.es/~meritxell/publications/TRR200804.pdf`.