Auton Agent Multi-Agent Syst
DOI 10.1007/s10458-010-9132-7

Constructing a unifying theory of dynamic programming DCOP algorithms via the generalized distributive law

Meritxell Vinyals · Juan A. Rodriguez-Aguilar · Jesús Cerquides

Abstract In this paper we propose a novel message-passing algorithm, the so-called Action-GDL, as an extension to the generalized distributive law (GDL) to efficiently solve DCOPs. Action-GDL provides a unifying perspective of several dynamic programming DCOP algorithms that are based on GDL, such as DPOP and DCPOP algorithms. We empirically show how Action-GDL using a novel distributed post-processing heuristic can outperform DCPOP, and by extension DPOP, even when the latter uses the best arrangement provided by multiple state-of-the-art heuristics.

Keywords GDL · Generalized distributive law · DCOP · DCPOP · DPOP

1 Introduction

A distributed constraint optimization problem (DCOP) [11,12,14] is a formalism that captures the rewards and costs of local interactions in a multiagent system (MAS) where each agent chooses a set of individual actions. DCOP is a framework that can model a large number of coordination, scheduling, and task allocation problems in MAS that has already been applied to domains such as sensor networks [23], meeting scheduling [10] and synchronization of traffic lights [9].

M. Vinyals (✉) · J. A. Rodriguez-Aguilar
IIIA, Artificial Intelligence Research Institute, Spanish National Research Council,
Campus UAB, 08193 Bellaterra, Spain
e-mail: meritxell@iiia.csic.es

J. A. Rodriguez-Aguilar
e-mail: jar@iiia.csic.es

J. Cerquides
WAI, Departament Matemàtica Aplicada i Anàlisi, Universitat de Barcelona, Gran Via 585,
08191 Barcelona, Spain
e-mail: cerquide@maia.ub.es

Published online: 25 May 2010

State-of-the-art complete algorithms to solve DCOPs adopt two main approaches: search and dynamic programming. Search algorithms, like ADOPT [12] (including extensions such as IDB-ADOPT [21]), require linear-size messages, but an exponential number of messages. Dynamic programming algorithms, like the distributed pseudotree optimization procedure (DPOP) and its extensions [15], only require a linear number of messages, but their complexity lies on the message size, which may be very large. Recently, Atlas and Decker provided a generalization of DPOP, the so-called distributed cross-edged pseudotree optimization procedure (DCPOP) [3], which extends DPOP by handling an extended set of pseudotree arrangements. Moreover, researchers have also proposed hybrid algorithms that are able to preserve optimality whereas offering different trade-offs, as for instance MB-DPOP (message-size vs. memory) or PC-DPOP (number of messages vs partial centralisation) [14]. In general, DCOP algorithms share common goals, namely to minimise communication/computation and to maximise parallelism. For some domains, privacy is also an important issue.

Against this background, in this paper we provide a unifying perspective of several dynamic programming DCOP algorithms under the generalized distributive law (GDL). The GDL algorithm [1] is a general message-passing algorithm that exploits the way a global function factors into a combination of local functions. Particular cases of GDL have been unknowingly used by different communities under different names (e.g. Viterbi's [20], Pearl's belief propagation [13], or Shafer-Shenoy [17] algorithms among others). Thus, in this work we focus on how to specialise GDL to efficiently solve DCOPs. Along this line, we make several contributions:

- We formulate Action-GDL, a specialisation of GDL, which reduces the number and size of messages when solving DCOPs and prove that it generalises DPOP and DCPOP algorithms
- We theoretically characterise when Action-GDL can outperform DPOP. Thus, we observe that Action-GDL: (i) provides significant savings in computation over DPOP when pseudotrees are generated by edge-traversal heuristics; (ii) provides no significant savings in computation for unrestricted pseudotrees; and (iii) can severely reduce communication complexity.
- We formulate a distributed post-processing heuristic to improve Action-GDL problem arrangements, namely junction trees
- We provide empirical evidence that we can benefit from using Action-GDL instead of DCPOP, and by extension of DPOP. Concretely we observe than our distributed post-processing heuristic allows Action-GDL to outperform DCPOP by: (i) decreasing communication (up to around 85%); (ii) reducing computation (up to around 30%); and (iii) increasing parallelism (up to around 60%)

The paper is organised as follows. Section 2 introduces DCOPs as well as the notation used throughout the paper. Section 3 outlines GDL and details Action-GDL. In Sect. 4 we prove that DPOP and DCPOP are particular cases of Action-GDL. Next, in Sect. 5 we theoretically characterise when Action-GDL can outperform DPOP and formulate our distributed post-processing heuristic. Finally, Sect. 6 details our empirical analysis, and Sect. 7 draws some conclusions and sets paths to future research.

2 DCOP definition and notation

In this section we introduce the distributed constraint optimization problem as well as the notation we employ throughout the rest of this paper.

2.1 The distributed constraint optimization problem

A distributed constraint optimization problem (DCOP) [12,15] 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 utility,[1] namely a relation. The goal of a DCOP algorithm is to distributedly assign values to these variables so that the total utility is maximized.

Let $\mathcal{X} = \{x_1, \ldots, x_n\}$ be a set of variables over domains $\mathcal{D}_1, \ldots, \mathcal{D}_n$. A *utility relation* with domain variables $\{x_{i_1}, \ldots, x_{i_m}\}$ is a function $r : \mathcal{D}_r \to \mathbb{R}^+$, where $\mathcal{D}_r = \mathcal{D}_{i_1} \times \cdots \times \mathcal{D}_{i_m}$, that assigns a utility value to each combination of values of its domain variables. Formally, a DCOP is a tuple $\langle \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$ where: $\mathcal{X}$ is a set of variables, each one assigned to a different agent; $\mathcal{D}$ is the joint domain space for all variables; and $\mathcal{R} = \{r_1, \ldots, r_p\}$ is a set of utility relations.

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. In a DCOP each agent receives knowledge about all relations that involve its variable(s). In this work we assume that each agent is assigned a single variable, so we will use the terms "agent" and "variable" interchangeably. A DCOP is typically represented with its constraint graph, whose vertices stand for variables and whose edges link variables that have some direct dependency (appear together in the domain of some relation), as shown by the examples depicted in Figs. 3a and 4a.

2.2 Notation

Next we define the functions and operators that we shall employ henceforth. Henceforth, given some variable set $X \subseteq \mathcal{X}$, $\mathcal{D}_X$ will stand for the joint domain space of variables in X. For the sake of simplicity in given examples we assume binary utility relations involving two variables, although all the results in the paper are valid for any arity. 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}.

Definition 1 (*Scope*) The scope function returns the domain variables of a given set of relations. Ex: $Scope(\{r^{12}\}) = \{x_1, x_2\}$, $Scope(\{r^{12}, r^{24}\}) = \{x_1, x_2, x_4\}$.

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, $\bar{X}^r = Scope(r) \setminus X$.

Definition 3 (*Utility message*) A message from agent x_i to agent x_j is a utility message over $X \subseteq \mathcal{X}$, if the information sent is a utility relation over X. Henceforth, we shall denote that utility relation as μ_{ij}.

Definition 4 (*Assignment*) Given a set of variables $X \in \mathcal{X}$, 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

[1] Costs can be represented with negative values.

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: $X = \{x_1, x_3\}$, σ an assignment over X having $\sigma(x_1) = 1, \sigma(x_3) = 0$. x_2 and x_4 are free in σ. If $Y = \{x_1\}$ we have that $\sigma[Y](x_1) = 1$. x_2, x_3 and x_4 are free in $\sigma[Y]$.

Definition 5 (*Value message*) A message from agent x_i to agent x_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}.

Definition 6 (*Join*) Let r, s be two relations, $Scope(\{r, s\}) = \{x_{k_1}, \ldots, x_{k_m}\}$ be their joint scope and $\mathcal{D}_{r\otimes s} = \mathcal{D}_{k_1} \times \ldots \times \mathcal{D}_{k_m}$ be their joint domain space. The combination of r and s (noted $r \otimes s$) is a utility relation over $Scope(\{r, s\})$ 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: $(r^{12} \otimes r^{24})(0, 1, 0) = r^{12}(0, 1) + r^{24}(1, 0)$. We can readily generalize the join operator over a finite set of relations: $\bigotimes\limits_{\{r_1,\ldots,r_m\}} = r_1 \otimes (r_2 \otimes \ldots (r_{m-1} \otimes r_m) \ldots)$. Thus, the join operator is a combination operator that joins the knowledge represented by two relations into a single one by adding their values.

Definition 7 (*Summarization*) The summarization operator of relation r over a set of variables X returns a utility relation over X such that $(\bigoplus\limits_X r)(d_X) = \max\limits_{d\in\mathcal{D}_{\bar{X}^r}} r(d_X, d)$. Ex:$(\bigoplus\limits_{\{x_2\}} r^{12})(0) = \max\limits_{d_1\in\mathcal{D}_1} r^{12}(d_1, 0)$ and $(\bigoplus\limits_{\{x_2\}} r^{12})(1) = \max\limits_{d_1\in\mathcal{D}_1} r^{12}(d_1, 1)$. Notice that we can employ the summarization operator by specifying the variables to eliminate from a relation as follows $\bigoplus\limits_{\backslash X} r = \bigoplus\limits_{\bar{X}^r} r = \bigoplus\limits_{Scope(r)\backslash X} r$. The summarization operator sums up the utility that a relation contains over a set of variables.

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 $(\mathop{\nabla}\limits_{\sigma} r)(d_{\bar{X}^r}) = r(d_X, d_{\bar{X}^r})$ where $d_X \in \mathcal{D}_X$ contains the values set by σ to the variables in X. Ex: $X = \{x_2\}$, $\sigma(x_2) = 1$ then $(\mathop{\nabla}\limits_{\sigma} r^{12})(0) = r^{12}(0, 1)$ and $(\mathop{\nabla}\limits_{\sigma} r^{12})(1) = r^{12}(1, 1)$.

3 The Action-GDL algorithm

In this section we introduce Action-GDL, a specialization of the GDL algorithm in [1] to efficiently solve DCOPs. Firstly, in Sect. 3.1 we outline the GDL algorithm. Secondly, in Sect. 3.2 we show how to extend GDL to efficiently solve DCOPs, namely to allow individual agents to calculate the variable assignment that maximizes the DCOP objective function of Eq. 1.

3.1 The generalized distributive law

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. In our case, since we are concerned with the problem of maximizing a utility function, such operations are addition and 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) [8].

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, where each clique $\mathcal{C}_i$ is a subset of variables $\mathcal{C}_i \subseteq \mathcal{X}$; $\mathcal{S}$ is a set of separators, where each separator s^{ij} is an edge between clique $\mathcal{C}_i$ and $\mathcal{C}_j$ containing their intersection, namely $s^{ij} = \mathcal{C}_i \cap \mathcal{C}_j$; and $\Psi = \{\psi_1, \ldots, \psi_m\}$ is a set of potentials, one per clique, where potential ψ_i is a function assigned to clique $\mathcal{C}_i$. 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 $Scope(\psi_i) \subseteq \mathcal{C}_i$.
- *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 JT 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, namely variables x_2 and 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. The purpose of the algorithm is that cliques distributedly compute some objective function that is factored among them. To illustrate the way GDL operates, consider the following example. Say that our goal is to distributedly maximise some objective function $f(x_1, x_2, x_3, x_4) = \psi_1(x_2, x_4) + \psi_2(x_1, x_2) + \psi_3(x_2, x_3, x_4)$, whose subfunctions ($\psi_1$, ψ_2 and ψ_3) are arranged in the directed JT of Fig. 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 it has received messages from all of its children; and (ii) one down the tree so that each clique sends a message to its children when it receives a message from its parent.

Figure 1 shows a trace of the operation of GDL over the JT in Fig. 2. At round 1, $\mathcal{C}_2 = \{x_1, x_2\}$ sends a message μ_{21} to $\mathcal{C}_1 = \{x_2, x_4\}$ 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 $\mathcal{C}_1$ receives the values of its children's local functions for its variables x_2, x_4, it combines those values into $\bar{\mathcal{K}}_1$. $\bar{\mathcal{K}}_1$ is a function that stands for $\mathcal{C}_1$ knowledge over its variables. At that point, since $\mathcal{C}_1$ has received messages from all its neighbors, $\bar{\mathcal{K}}_1$ has complete knowledge about the global objective function for the variables in the clique. At rounds 4 and 5, $\mathcal{C}_1$ sends messages to its children that contain the combination (join operation) of its local function, ψ_1, with other children messages.

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

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

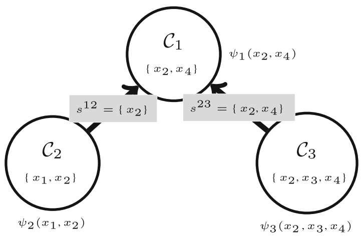


Fig. 2 Example of JT

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 $\bar{\mathcal{K}}_2$ (round 6). Analogously $\mathcal{C}_3$ can compute $\bar{\mathcal{K}}_3$ at round 7 using the message from $\mathcal{C}_1$.

Once the message-passing phase of GDL is over, each clique has complete knowledge of the global objective function for the variables in the clique. Now each clique can locally compute the assignment of variables that maximises the objective function. However, such local computations do not guarantee that cliques agree on their assignments. This is the case when several assignments maximise the objective function. In what follows we generalise the example above to provide a more formal description of GDL[2] that will help formalising the operations that Action-GDL requires.

Firstly, the knowledge of a clique $\mathcal{C}_i$ results from the combination of its local function with each of the messages it has received from its neighbours. More formally, the knowledge of a clique $\mathcal{C}_i$ is defined as a table containing the values of a function $\bar{\mathcal{K}}_i : \mathcal{D}_{\mathcal{C}_i} \rightarrow \mathbb{R}$. Initially, $\bar{\mathcal{K}}_i$ is set to be the local potential ψ_i, but when $\bar{\mathcal{K}}_i$ is updated, GDL uses the following rule:

$$\bar{\mathcal{K}}_i = \psi_i \otimes \left[\bigotimes_{\mathcal{C}_k adj\ \mathcal{C}_i} \mu_{ki} \right] \tag{2}$$

where $\mathcal{C}_k\ adj\ \mathcal{C}_i$ indicates that $\mathcal{C}_k$ and $\mathcal{C}_i$ are adjacent cliques in the JT.

Secondly, notice that if two cliques $\mathcal{C}_i$ and $\mathcal{C}_j$ are connected by a separator s^{ij}, the message from $\mathcal{C}_i$ to $\mathcal{C}_j$ is a table containing the values of function $\mu_{ij} : \mathcal{D}_{s^{ij}} \rightarrow \mathbb{R}$.[3] When a clique $\mathcal{C}_i$ sends a message to $\mathcal{C}_j$, it combines its local knowledge with all messages it has received from its neighbours other than $\mathcal{C}_j$, and transmits the result to $\mathcal{C}_j$. Following [1], we can regard a JT as a communication network where an edge from $\mathcal{C}_i$ and $\mathcal{C}_j$ is a transmission line that "filters out" dependence (by summarization in our case) on all variables but those common to $\mathcal{C}_i$ and $\mathcal{C}_j$. When a message μ_{ij} is updated, the following rule applies:

$$\mu_{ij} = \bigoplus_{s^{ij}} \left[\psi_i \otimes \bigotimes_{\mathcal{C}_k adj\ \mathcal{C}_i, k \neq j} \mu_{ki} \right] \tag{3}$$

To summarise, Eqs. 2 and 3 are the basis of GDL. While Eq. 3 helps a clique assess messages to send to its neighbours, Eq. 2 allows to compute the objective function at a particular clique.

Consider two special cases regarding the application of GDL [1]: the *single-vertex* problem, when the goal is to compute the objective function at a single clique, and the *all-vertices* problem, when the goal is to compute the objective function at all cliques. For instance, con-

[2] For the sake of simplicity we restrict our description to the max-sum commutative semiring. Notice though that GDL applies to a larger variety of semirings. We refer the interested reader to [1] for an excellent discussion on this issue.

[3] Initially all μ_{ij} functions are set to 0.

sider the trace in Fig. 1. At step 3 the single-vertex problem for clique $\mathcal{C}_1$ is solved, whereas at step 7 the all-vertices problem is solved because all cliques can assess the objective function.

GDL is a synthesis of the work in many fields. More concretely, it generalises Viterbi's [20], Pearl's belief propagation [13], or Shafer-Shenoy [17] algorithms among others. Here, we observe that it is also the case of the cluster tree elimination (CTE) algorithm described in [6], a message-passing algorithm that can help solve several automated reasoning tasks over a tree decomposition. Interestingly, the original description of CTE is equivalent to the *fully serial* version of GDL [1]: a clique sends a message to a neighbour when, for the first time, it has received messages from all of its other neighbours, and computes its knowledge when, for the first time, it has received messages from all its neighbours. Notice that GDL can also employ either a *fully parallel* schedule (at every iteration, knowledge at each clique is updated, and every message is computed and transmitted, simultaneously) or *hybrid* schedule (intermediate between fully serial and fully parallel) to solve the all-vertices problem. Importantly, no matter the schedule, after a linear number of iterations the knowledge of the cliques are equal to the desired objective functions, and GDL terminates.

3.2 Extending GDL to solve DCOPs

Recall that our goal is to efficiently solve DCOPs as formalised in Sect. 2. Therefore, the capability of distributedly computing an objective function,[4] as provided by GDL, is not enough. We need to go one step beyond GDL to allow agents to individually assign values to local variables such that these values are a joint assignment that maximizes the DCOP objective function of Eq. 1, namely the *optimal assignment*. At this aim, Action-GDL extends GDL to efficiently infer the optimal assignment.

Consider a DCOP setting arranged as a JT. According to the description of GDL above, when a clique has received messages from all its neighbors, it counts on all the information related to its variables. Thus, it can compute its objective function. Thereafter, the clique would be able to find the optimal assignment for its variables provided that it is aware of its parent clique decisions (variables' assignments) and can set the rest of clique's variables according to them. Once the clique finds the optimal assignment for its variables, it can directly propagate its assignment down the tree. Notice that there is no need to propagate utility messages down the tree (unlike GDL when solving the all-vertices problem) because all a child requires to find an optimal assignment for its variables is its parent's assignment. Therefore, when solving a DCOP, after the first message-passing phase of GDL, up the tree, is over, the second message-passing phase of GDL, down the tree, is no longer necessary. Instead, 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*. This leads to savings in communication, since utility messages are space-exponential in the separator size whereas the size of a value messages is linear. Moreover, 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.

Algorithm 1 outlines Action-GDL. For simplicity we have assumed that each agent x_i is assigned a single clique $\mathcal{C}_i$. Given a $JT = \langle \mathcal{X}, \mathcal{C}, \mathcal{S}, \Psi \rangle$, each clique $\mathcal{C}_i$ starts with a tuple $\langle C_p, Ch_i, \psi_i, S_i \rangle$, where $C_p \in \mathcal{C}$ stands for its parent, $Ch_i \subseteq 2^{\mathcal{C}}$ stands for its children, and $S_i \in \mathcal{S}$ stands for the separators relating the clique to its adjacent cliques.

[4] In fact, each clique ends up in GDL with a summarization of the global objective function over its variables.

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

```
Each clique $\mathcal{C}_i$ starts with $\langle \mathcal{C}_p, Ch_i, \psi_i, \mathcal{S}_i \rangle$ and runs:
1: **Phase I: UTILITY Propagation**
2: $\widehat{\mathcal{K}}_i = \psi_i$;
3: **for all** $\mathcal{C}_j \in Ch_i$ **do**
4:     Wait for utility message $\mu_{ji}$ from $\mathcal{C}_j$
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 **then**
8:     Let $s^{ip} \in \mathcal{S}_i$ be the separator between $\mathcal{C}_i$ and its parent $\mathcal{C}_p$
9:     Send $\mu_{ip} = \bigoplus_{s^{ip}} \widehat{\mathcal{K}}_i$ to $\mathcal{C}_p$
10: **end if**
11: **Phase II: VALUE propagation**
12: **if** $\mathcal{C}_i$ is not the tree's root **then**
13:     Wait for a value message $\sigma_{pi}$ from $\mathcal{C}_p$
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: $d_i^* = arg \max_{d \in \mathcal{D}_{Scope(\widehat{\mathcal{K}}_i)}} \widehat{\mathcal{K}}_i(d)$; /*Assess best values for unassigned local variables*/
17: $d^* = d_i^* \cup \sigma_{pi}$; /*Put together the assessed value and the message received*/
18: **for all** $\mathcal{C}_j \in Ch_i$ **do**
19:     Let $\sigma_{ij} = d^*_{s^{ij}}$; /*Project over separator*/
20:     Send $\sigma_{ij}$ to $\mathcal{C}_j$
21: **end for**
22: **return** $d^*$;
```

During the utility propagation phase (lines 1–10), cliques exchange utility messages. The initial knowledge for each clique is its potential (line 2). Each clique 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 children cliques. Every time that a clique receives a new utility message, it incorporates it (by using the join operator) to its local knowledge (line 5). After combining utility messages from all the children of a clique, if the clique has a parent (line 7), it summarizes that part of its local knowledge (using the summarization operator) that is of interest to its parent (by means of a utility relation over its separator). After that, the result of the summarisation is sent to its parent (line 9).

During the value propagation phase (lines 11–22), cliques compute the optimal values for their variables and exchange value messages, namely decisions. Given a clique, it waits until receiving a value message (containing assignments) for all variables in common (in the separator) with its parent (line 12–14). At that point, the clique has received all the knowledge, in form of utility (from children) and value (from the parent) messages, required to compute the objective function related to its variables. The clique slices its knowledge by incorporating the already inferred decisions (line 15) and computes the optimal values for the rest of its variables (line 17). Once a clique finds the optimal assignment for its variables, it can propagate it down the tree (lines 19–22). Notice however that a clique only propagates variable assignments required by its children cliques, namely assignments for variables in their separator.

Table 1 displays a trace of Action-GDL over the JT in Fig. 2. In order to run Action-GDL, the set of potentials is defined as $\Psi = \{\psi_1, \psi_2, \psi_3\}$, where $\psi_1 = r^{24}$, $\psi_2 = r^{12}$, and $\psi_3 = r^{23} \otimes r^{34}$. Hence, the objective function encoded in the JT is defined as $f(x_1, x_2, x_3, x_4) = \psi_1(x_2, x_4) + \psi_2(x_1, x_2) + \psi_3(x_2, x_3, x_4)$, which is the same as the one in the constraint graph

Table 1 Trace of Action-GDL over the JT of Fig. 2

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

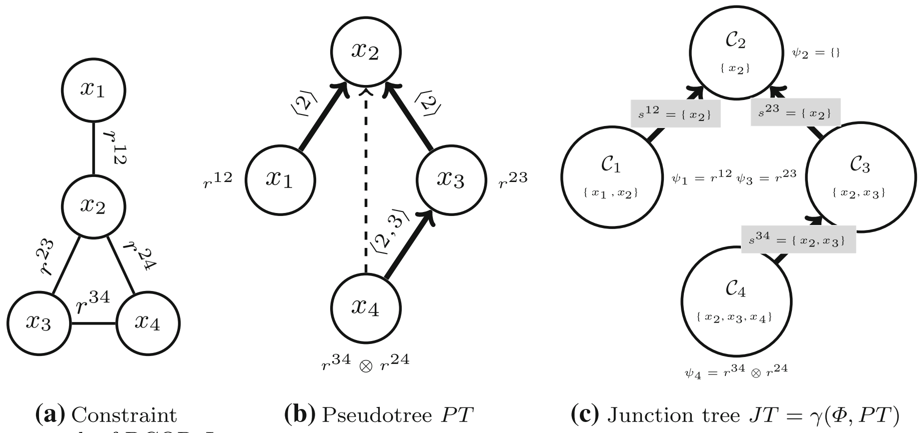


Fig. 3 Example of constraint graph, a pseudotree PT and its equivalent junction tree JT

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

To summarize, as can be seen by following algorithm 1, the knowledge of a clique $\mathcal{C}_i$ at the end of an Action-GDL run is:

$$\widehat{\mathcal{K}}_i = \underset{\sigma_{pi}}{\triangledown}\left[\psi_i \otimes \bigotimes_{\mathcal{C}_j \in Ch_i} \mu_{ji}\right] \tag{4}$$

We can readily assess Action-GDL complexity from cliques' and separators' sizes. Action-GDL requires a number of messages linear to the number of edges in the JT (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. The local memory required by each clique $\mathcal{C}_i$ depends on its separators' sizes and on the size of its local knowledge $\mathcal{K}_i$, which is exponential to the clique's size. Regarding the computation required by each clique to build messages and find assignments, it also scales with a cliques' size.

4 Generality of Action-GDL

In the previous section we have presented Action-GDL. In this section we show the generality of Action-GDL by showing that it unifies two state-of-the-art dynamic programming optimal DCOP algorithms that are based on the general distributed law (GDL): DPOP [15] and DCPOP [3].

4.1 Action-GDL generalizes DPOP

In this section we prove that DPOP is a particular case of Action-GDL when it is executed under certain JTs. First, we give an overview of the DPOP algorithm in Sect. 4.1.1. Next we prove that Action-GDL generalizes DPOP by: (1) providing a mapping from pseudotrees (input of DPOP) to JTs (Sect. 4.1.2); and (2) proving that given any pseudotree, the execution of DPOP is equivalent to the execution of Action-GDL over the JT produced by our mapping for the pseudotree (Sect. 4.1.3).

4.1.1 Overviewing DPOP

DPOP [15] is a complete state-of-the-art dynamic programming algorithm to solve DCOPs. DPOP arranges a DCOP in a pseudotree (PT), namely a rooted tree with the same variables as the DCOP and the property that adjacent nodes from the DCOP constraint graph fall in the same branch of the tree. Figure 3b shows a PT for the constraint graph in Fig. 3a. A PT of a constraint graph has two kinds of edges: *tree-edges* (boldfaced lines); and *pseudoedges* (dashed lines). These edges stand for two relationships between variables: (1) parent/children for variables connected through an edge (e.g. in Fig. 3b x_2 is the parent of x_3); (2) pseudoparent/pseudochildren for variables connected through a pseudoedge (e.g. x_4 is a pseudochild of x_2). Therefore, we can represent a PT as a pair $\langle P, PP \rangle$, where P and PP are functions that map each variable to its parent and pseudoparents, respectively. We obtain functions Ch and PCh, which return a variable's children and pseudochildrens respectively, from the functions above as $Ch(x_i) = \{x_j \in \mathcal{A} | P(x_j) = x_i\}$ and $PCh(x_i) = \{x_j \in \mathcal{A} | x_i \in PP(x_j)\}$.

Thus, when running DPOP, agents start with a pre-processing phase, to generate a PT by running a distributed depth first search (DFS) algorithm guided by some heuristic. Then, given a PT, DPOP has two message-passing phases: (1) to exchange *utilities* about variables; and (2) to propagate *values* of variables inferred. Algorithm 2 shows these two phases in terms of the operators introduced in Sect. 2.2.

In DPOP the initial knowledge of an agent x_i, namely $\mathcal{K}_i^0$, is set to the combination of some unary relation involving x_i and of some binary relations linking x_i with one of its parent or pseudoparent variables. Thus, in Fig. 3b the knowledge of agent x_4 ($\mathcal{K}_4^0$) is initially composed of relations r^{24} and r^{34} (no unary relation in that case).

During the first message-passing phase, the utility propagation phase (lines 1–9), each agent x_i receives utility messages from all its children variables. The utility message that agent x_i exchanges with the agent related to its parent variable, x_p, is the summarization of its current knowledge filtering out x_i (line 8). Formally:

$$\mu_{ip} = \bigoplus_{\backslash x_i} \left[\mathcal{K}_i^0 \otimes \bigotimes_{x_j \in Ch(x_i)} \mu_{ji} \right] \tag{5}$$

Algorithm 2 DPOP($\langle \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$,$\langle P, PP \rangle$)

```
Each agent $x_i \in \mathcal{X}$, receives $\langle P_i, PP_i, \mathcal{K}_i^0 \rangle$ where $\mathcal{K}_i^0 = r^i \otimes \bigotimes_{x_k \in \{P(x_i)\} \cup PP(x_i)} r^{ik}$ and runs:
1: Phase I: UTILITY Propagation
2: $\mathcal{K}_i = \mathcal{K}_i^0$;
3: for all $x_j \in Ch(x_i)$ do
4:     Wait for the utility message $\mu_{ji}$ from $x_j$
5:     $\mathcal{K}_i = \mathcal{K}_i \otimes \mu_{ji}$;
6: end for
7: if $x_i$ is not the tree's root, let $x_p = P(x_i)$ then
8:     Send $\mu_{ip} = \bigoplus_{\backslash x_i} \mathcal{K}_i$ to $x_p$
9: end if
10: Phase II: VALUE propagation
11: if $x_i$ is not the tree's root, let $x_p = P(x_i)$ then
12:     Wait for a value message $\sigma_{pi}$ from $x_p$
13:     $\mathcal{K}_i = \bigtriangledown_{\sigma_{pi}} \mathcal{K}_i$; /*Slice $\mathcal{K}_i$ with the value message*/
14: end if
15: $d_i^* = arg \max_{d_i \in \mathcal{D}_i} \mathcal{K}_i(d_i)$; /* Assess best value for $x_i$ */
16: $d^* = d_i^* \cup \sigma_{pi}$; /* Put together the assessed value and the message received. */
17: for all $x_j \in Ch(x_i)$ do
18:     Send $\sigma_{ij} = d^*_{Scope(\mu_{ji})}$ to $x_j$ /* Send to $x_j$ the variables he is interested in */
19: end for
20: return $d_i^*$;
```

During the second message-passing phase, the value propagation phase (lines 10–19), each agent x_i receives a value message (σ_{pi}) from the agent assigned to its parent variable, x_p. That value message contains assignments for all variables in the domain of the utility message (μ_{ip}) that agent x_i has sent to x_p in the previous phase. Once agent x_i has received the value message from its parent x_p, agent x_i restricts its knowledge by incorporating the assigned variables (line 13).

Then, agent x_i assesses the value of x_i as the one that maximizes its local knowledge (line 15) and completes it with the value message received from its parent (line 16).

Thereafter, agent x_i propagates to every children of x_i a value message that contains the values assigned to already decided variables that it is interested in (lines 17–19).[5]

To summarize, from algorithm 2 we obtain that the knowledge of agent x_i at the end of a DPOP execution is:

$$\mathcal{K}_i = \bigtriangledown_{\sigma_{pi}} \left[\mathcal{K}_i^0 \otimes \bigotimes_{x_j \in Ch(x_i)} \mu_{ji} \right] \tag{6}$$

4.1.2 Mapping PTs to JTs

Before proving the equivalence of Action-GDL and DPOP, in this section we define a mapping that builds a JT from a PT. First of all, we offer the intuitions behind our mapping. In general, we propose to map each PT to a JT with as many cliques as nodes in the PT. In fact, for each node in a PT we require its counterpart as a clique in the JT to be produced

[5] When looking at lines 17–19, recall that $d^*_{Scope(\mu_{ji})}$ stands for the values to be assigned to the variables in the domain of the utility message μ_{ji}.

by the mapping. Hereafter, we consider the variables to include in each clique. For each node in the PT, its clique in the JT must contain: (1) the node's variable; (2) the variables expected by the node's parents/pseudoparents up in the PT; and (3) the variables that the node's children need to forward up the PT.

We will refer to the node's variable and the second set of variables as the *directly related variables* (DRV), and to the third set of variables as the *inherited related variables* (IRV). Hence, given a node x_i in a PT, we can readily define the variables of its clique by wrapping up directly and inherited related variables as follows:

$$\mathcal{C}_i = DRV(x_i) \cup IRV(x_i) \tag{7}$$

On the one hand, the directly related variables of a node include its variable, its parents' and its pseudoparents'. Formally:

Definition 10 Given a variable x_i in a PT, its **directly related variables** are:

$$DRV(x_i) = \{x_i\} \cup \{P(x_i)\} \cup PP(x_i) \tag{8}$$

On the other hand, the inherited related variables of a node include the variables that its children must send up the tree after eliminating their own variables. Formally:

Definition 11 Given a variable x_i in a PT, its **inherited related variables** are:

$$IRV(x_i) = \bigcup_{x_j \in Ch(x_i)} Sep(x_j) = \bigcup_{x_j \in Ch(x_i)} \mathcal{C}_j \backslash \{x_j\} \tag{9}$$

Observe that the only variable that a node can remove from a clique's child is its child variable. Notice also that the definition of inherited related variables leads to a recursive definition of cliques and that the set of inherited related variables is empty for leaf nodes.

Once obtained cliques' variables, we can assess the potentials and separators completing the definition of a JT. Thus, finally the mapping γ below allows us to build a JT from a PT.

Definition 12 (γ) Let γ be a function that given a DCOP $\Phi = \langle \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$ and a $PT = \langle P, PP \rangle$ maps them to a junction tree $\gamma(\Phi, PT) = \langle \mathcal{X}, \mathcal{C}, \mathcal{S}, \Psi \rangle$, where:

1. The set of variables $\mathcal{X}$ is the same as in PT.
2. The set of cliques $\mathcal{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_{|X|}\}$ contains one clique per variable in PT. The clique $\mathcal{C}_i$ contains all the variables directly or inherited related to variable x_i as defined by expression 7.
3. The set of potentials Ψ contains one potential associated to each clique. Each clique potential ψ_i is the combination of: (i) a unary relation r_i that involves the clique decision variable x_i; and (ii) the binary relations that link x_i with its parent or one of its pseudoparents. Formally:

$$\psi_i = r^i \otimes [\bigotimes_{x_j \in \{P(x_i)\} \cup PP(x_i)} r^{ij}] \tag{10}$$

4. The set of separators $\mathcal{S}$ contains one separator s^{ij} per pair of cliques $\mathcal{C}_i$ and $\mathcal{C}_j$ such that x_j is parent of x_i in the PT. By definition of JT, each separator s^{ij} contains the intersection of its cliques ($s^{ij} = \mathcal{C}_i \cap \mathcal{C}_j$).

Figure 3b shows a PT over the DCOP of Fig. 3a while Fig. 3c shows the junction tree $\gamma(\Phi, PT)$.

4.1.3 Proving equivalence

The previously introduced mapping (γ) builds a JT from each PT. In the remaining of the section we prove that running DPOP over that PT is equivalent to running Action-GDL over the JT resulting from applying γ to the PT. First we state (Lemma 1) that both the computation performed and the messages exchanged during the utility propagation phase are the same. After that, we state (Lemma 2) that the messages exchanged during the value propagation phase are also the same. Finally we combine these two lemmas to prove our main result (Theorem 1). The proofs of the lemmas are provided in [18].

Lemma 1 *Given a DCOP Φ and a PT, the computation performed and the messages exchanged during the utility phase of DPOP(Φ, PT) and $Action - GDL(\gamma(\Phi, PT))$ are the same.*

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

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

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

Proof Since both algorithms are only composed of an utility phase and a value propagation phase, the result follows directly from Lemmas 1 and 2.

Since computing mapping γ can be done efficiently and distributedly [19], Theorem 1 proves that Action-GDL can be at least as efficient as DPOP in any DCOP (by mimicking its behavior).

4.2 Action-GDL generalizes DCPOP

In this section we prove that Action-GDL can be at least as efficient as DCPOP in any DCOP by producing equivalent executions. First, we overview the DCPOP algorithm in Sect. 4.2.1. Next we prove that Action-GDL can produce DCPOP-equivalent executions by: (1) providing a mapping from cross-edged PT (input of DCPOP) to JTs (Sect. 4.2.2); and (2) proving that given any cross-edged PT, the execution of DCPOP over the cross-edged PT is equivalent to the execution of Action-GDL over the junction tree produced by our mapping for the cross-edged PT (Sect. 4.2.3).

4.2.1 Overviewing DCPOP

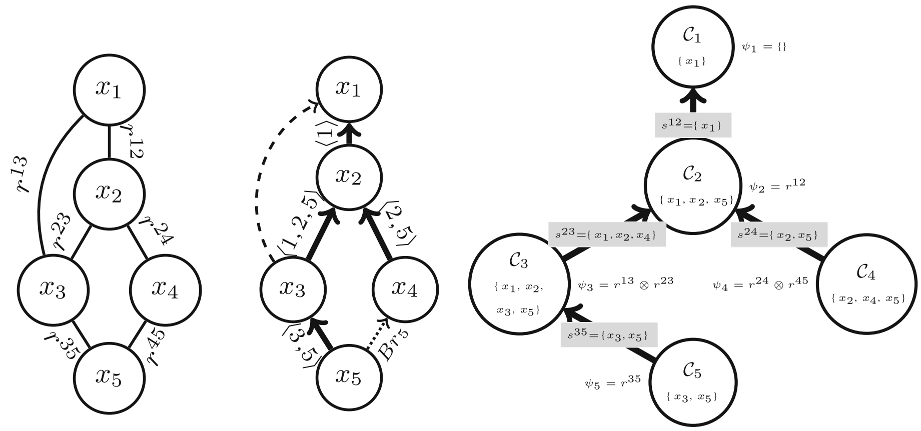
DCPOP [3] is a generalization of DPOP based on an extension of PTs, namely cross-edged PTs. A cross-edged PT (CT) is a PT with the addition of cross-edges (dotted line). Figure 4b shows a CT for the constraint graph in Fig. 4a. A cross-edge is an edge from node x_i to node x_j that is above x_i but not in the path from x_i to the root. Thus, besides the parent and pseudoparent relationships, a CT adds a new type of relationship: branch-parent/branch-children for variables connected through a cross-edge (for instance, from x_3 to x_4 in Fig. 4b). Therefore, we can represent a CT as a tuple $\langle P, PP, BP\rangle$, where P, PP, and BP are functions that map each variable to its parent, pseudoparents and branchparents respectively. We obtain function BCh, which returns a variable's branch-children, as $BCh(x_i) = \{x_j \in \mathcal{A} | BP(x_j) = x_i\}$.

Thus, when running DCPOP, agents start with a pre-processing phase to generate a CT by running a distributed Best-first search (BFS) algorithm guided by some heuristic. Then, likewise DPOP, DCPOP has two main phases: to have agents exchange *utilities*, and to have agents propagate *values*. Algorithm 3 shows the phases of DCPOP once a CT is generated in terms of the operators introduced in Sect. 2.2. Notice that the algorithm splits the original utility propagation phase in [3] into two phases: to propagate branch information and to propagate utility information. This encoding aims at easing the comparison with both DPOP and Action-GDL.

In DCPOP the initial knowledge of an agent x_i ($\mathcal{K}_i^0$), besides being composed of some unary relation involving x_i and binary relations linking x_i with one of its parent/pseudoparent variables, also contains binary relations linking x_i with its branch-children variables. Thus,

Algorithm 3 DCPOP($\langle \mathcal{X}, \mathcal{D}, \mathcal{R}\rangle$,$\langle P, PP, BP\rangle$)

```
Agent $x_i \in \mathcal{X}$ receives $\langle P(x_i), PP(x_i), BP(x_i), \mathcal{K}_i^0\rangle$ where
$\mathcal{K}_i^0 = r^i \otimes \bigotimes_{x_k \in \{P(x_i)\} \cup PP(x_i)} r^{ik} \otimes \bigotimes_{x_k \in BCh(x_i)} r^{ik}$ and runs:
1: Phase I: Branch information propagation
2: $Br_i = \langle i, |BP(x_i)| + 1, 1\rangle$; /*Create branch information for own variable*/
3: Send $Br_i$ to all $BP(x_i)$
4: for all $x_k \in BCh(x_i)$ do
5:     Wait for branch information $Br_k$ from $x_k$
6:     $\langle Br_i, MV\rangle = mergeBranches(Br_i, Br_k)$
7: end for
8: Phase II: UTILITY Propagation
9: $\mathcal{K}_i = \mathcal{K}_i^0$;
10: for all $x_j \in Ch(x_i)$ do
11:     Wait for utility message $\langle \mu_{ji}, Br_j\rangle$ from $x_j$
12:     $\mathcal{K}_i = \mathcal{K}_i \otimes \mu_{ji}$;
13:     $\langle Br_i, MV\rangle = mergeBranches(Br_i, Br_j)$
14: end for
15: if $x_i$ is not the tree's root, let $x_p = P(x_i)$ then
16:     Send $\langle \mu_{ip} = \bigoplus_{\backslash MV} \mathcal{K}_i, Br_i\rangle$ to $x_p$
17: end if
18: Phase III: VALUE propagation
19: if $x_i$ is not the tree's root, let $x_p = P(x_i)$ then
20:     Wait for a value message $\sigma_{pi}$ from $x_p$
21:     $\mathcal{K}_i = \underset{\sigma_{pi}}{\triangledown} \mathcal{K}_i$; /*Slice $\mathcal{K}_i$ with the value message*/
22: end if
23: $d_i^* = arg \max_{d \in \mathcal{D}_{MV}} \mathcal{K}_i(d)$; /* Assess best value for merged variables */
24: $d^* = d_i^* \cup \sigma_{pi}$; /* Put together the assessed values and the message received. */
25: for all $x_j \in Ch(x_i)$ do
26:     Send $\sigma_{ij} = d^*_{Scope(\mu_{ji})}$ to $x_j$ /* Send to $x_j$ the variables he is interested in */
27: end for
28: return $d_i^*$;
```



(a) Constraint graph of DCOP Φ. **(b)** DCPOP over CT. **(c)** Junction tree $JT = \gamma_{CT}(\Phi, CT)$.

Fig. 4 Example of constraint graph, cross-edged pseudotree and equivalent junction tree

in Fig. 4b the knowledge of agent x_4 is initially composed of relations r^{24} (shared with its parent) and r^{45} (shared with branch-child).

The main operational difference between DPOP and DCPOP has to do with the mechanics that DCPOP incorporates to deal with cross edges during utility propagation. That is because, in DCPOP, a branch-child variable x_i is not eliminated at its node, instead it is eliminated in some node up the tree, at the so-called merge point of x_i. Thus, in DCPOP, each branch-child x_i starts by sending branch information to its branch-parents to calculate the merge point of x_i (lines 2–3). The branch information for a variable x_i contains its identifier, the number of branches of x_i and the number of merged branches (initially set to 1). After that, each x_i receives and merges branch information from its branch-children (lines 4–7). Next, during the utility propagation phase (lines 8–17), each x_i receives utility messages from all its children variables and combines them with its local knowledge in the very same way as in DPOP. However, in DCPOP, these messages also contain branch information of branch-children variables. Therefore, x_i merges all branches with the same originator by adding up the number of merged branches and assesses the set of variables for which it is merge point (MV), namely variables for which the number of merged branches equals the total number of branches (line 13). At the end of this phase, x_i exchanges a message with its parent x_p (line 16) that contains a utility message that summarizes its current knowledge after filtering out MV and the merged branch information.

Regarding the second message-passing phase, the value propagation phase (lines 18–27), notice that each agent's behaviour is similar as in DPOP with the difference that instead of assessing its own variable, each node x_i assesses all variables in MV.

4.2.2 Mapping cross-edged trees into junction trees

Before proving the equivalence of Action-GDL and DCPOP, in this section we define a mapping that builds a JT from a CT. First of all, we offer the intuitions behind our mapping from a CT to a JT. In general, we propose to map each CT to a JT with as many cliques as nodes in the CT. For each node in a CT, its clique in the corresponding JT must contain:

(1) the node's variable; (2) the variables expected by the node's parents/pseudoparents up the CT; (3) the variables that the node's children need to forward up the CT; (4) the variables that the node's branch-children need to forward up the CT.

Therefore, notice that the mapping is very similar to mapping γ described in Sect. 4.1.2. The difference lies in the addition of point (4) above involving variables of branch-children and on the set of variables in point (3), which is extended.

Analogously to the approach followed in Sect. 4.1.2, given a node x_i in a CT, we can readily define the variables of its clique by wrapping up directly and inherited related variables (see Eq. 7). However, we must extend both sets.

Firstly, the set of directly related variables in Eq. 8 is extended to include the variables that the node's branch-children need to forward the tree. Formally:

Definition 13 Given a variable x_i in the CT, its directly related variables are:

$$DRV(x_i) = \{x_i\} \cup \{P(x_i)\} \cup PP(x_i) \cup BCh(x_i) \tag{11}$$

On the other hand, the inherited related variables of a node include the variables that each child must send up the tree after eliminating: (i) those that have already been merged (either by the child or below); and (ii) the child's own variable if it has not branch-parents. Formally:

Definition 14 Given a variable x_i in a CT, its inherited related variables are:

$$IRV(x_i) = \bigcup_{x_j \in Ch(x_i)} Sep(x_j) = \bigcup_{x_j \in Ch(x_i)} \mathcal{C}_j \setminus Removable(x_j) \tag{12}$$

where $Removable(x_j) = \{x_k | x_k \in \mathcal{C}_j, x_k \neq x_j, x_k \text{ and all } x_l \in BP(x_k) \text{ are descendants of } x_j\} \cup \{x_j | BP(x_j) = \emptyset\}$. Note that the difference between the definition of IRV for a CT (Eq. 12) and a PT (Eq. 9) lies in the set of removable variables (the variables that the node's children don't need to forward up the tree). Thus, the set of removable variables of a clique $\mathcal{C}_i$ in a PT is uniquely composed of variable x_i, while in a CT is composed of variables whose merge point is $\mathcal{C}_i$ (which includes x_i in case it has not branch-parents).

Next, we formulate function γ_{CT}, which defines the mapping from CTs to JTs by providing definitions for potentials and separators in addition to cliques.

Definition 15 (γ_{CT}) Let γ_{CT} be a function that maps a DCOP $\Phi = \langle \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$ and a $CT = \langle P, PP, BP \rangle$ into a junction tree $\gamma_{CT}(\Phi, CT) = \langle \mathcal{X}, \mathcal{C}, \mathcal{S}, \Psi \rangle$, where:

- The set of variables $\mathcal{X}$ is the same as in CT.
- The set of cliques $\mathcal{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_{|X|}\}$ contains one clique per variable in CT. Clique $\mathcal{C}_i$ contains all the variables directly or inherited related to variable x_i.
- The set of potentials Ψ contains one potential per clique. Each clique potential ψ_i is the combination of: (i) a unary relation r_i that involves the clique decision variable x_i; (ii) the binary relations that link x_i with its parent and pseudoparents; and (iii) the binary relations that link x_i with its branch-children. Formally:

$$\psi_i = r^i \otimes \left[\bigotimes_{x_j \in \{P(x_i)\} \cup PP(x_i)} r^{ij} \right] \otimes \left[\bigotimes_{x_j \in BCh(x_i)} r^{ij} \right] \tag{13}$$

- The set of separators $\mathcal{S}$ contains one separator $s^{ij} = \mathcal{C}_i \cap \mathcal{C}_j$ per pair of cliques $\mathcal{C}_i, \mathcal{C}_j$ such that x_j is parent of x_i in CT.

Figure 4b shows a CT over the DCOP Φ of Fig. 4a while Fig. 4c shows the junction tree $\gamma_{CT}(\Phi, CT)$. Observe that mapping γ_{CT} creates one clique per variable in the CT and that cliques' potentials are assessed following Eq. 13. Thus, the potential of $\mathcal{C}_4$ is composed of the combination of the relation with its parent x_2, namely r^{24}, and the relation with its branch-child x_5, namely r^{45}. The example also illustrates how merge points are naturally captured by their corresponding cliques. Notice that x_2 in the CT is the merge point for variable x_5. Say now that we have already generated clique $\mathcal{C}_2$ corresponding to variable x_2 and we intend to generate $\mathcal{C}_1$ for variable x_1. Following Eq. 11, the set of DRV of x_1 is uniquely composed of x_1. Next, we apply Eq. 12 to assess the set of IRV of x_1, which in Fig. 4c is composed of $\mathcal{C}_2 = \{x_1, x_2, x_5\}$ excluding the set of removable variables at x_2. The set of removable variables at x_2 includes x_2 itself because it has not branch-parents, and x_5 because both x_5 and its branch-parent x_4 are descendants of x_2. Hence $\mathcal{C}_1 = \{x_1\}$.

In general, if a variable x_i in a CT is the merge point for another variable x_j, our mapping guarantees that $\mathcal{C}_i$ eliminates variable x_j. Therefore, because variables' merge points are explicitly represented, the JT produced by our mapping saves both the computing and sending of branch information.

4.2.3 *Proving equivalence*

Analogously to the equivalence analysis involving Action-GDL and DPOP, in this section we analyse the relationship between DCPOP and Action-GDL. We argue that running DCPOP over a CT is equivalent to running Action-GDL over its (as produced by mapping γ_{CT}) JT whenever the computing and sending of branch information is disregarded. As argued above, such information is not required because in a JT the merging points of variables are explicitly represented. Under this assumption, we obtain analogous equivalence results to those obtained for DPOP.

Lemma 3 *Given a DCOP Φ and a CT, the computation performed and the messages exchanged during the utility phase of DCPOP(Φ, CT) and Action-GDL($\gamma_{CT}(\Phi, CT)$) are the same disregarding the computing and sending of branch information.*

Lemma 4 *Given a DCOP Φ and a CT the value assigned by each agent to its variable and the messages exchanged during the value propagation phase of DCPOP(Φ, CT) and Action-GDL($\gamma_{CT}(\Phi, CT)$) are the same.*

We can build the proof for Lemmas 3 and 4 following the same approach as in [18]. Here we only comment on the intuitions behind these proofs.

Regarding Lemma 3, if there are no cross-edges, DCPOP behaves like DPOP. If there are cross-edges, branch-children variables are eliminated on their merge points, namely on the lowest variables in the CT that are between them and the root and between their branch-parents and the root. As argued above, Action-GDL does not require branch information because merge points are explicitly represented in cliques and separators of the JT generated by mapping γ_{CT}. Thus, in the junction tree $\gamma_{CT}(\Phi, CT)$, the set of variables whose merge point is x_i are the variables in $\mathcal{C}_i$ that are not in the separator with its parent s^{ip}. Hence, if we focus on comparing the computing and sending of utility information as well as on the computing of local knowledge, we observe that the utility phase of DCPOP(Φ, CT) and Action-GDL($\gamma_{CT}(\Phi, CT)$) are the same.

Regarding Lemma 4, since variables assessed at some node x_i are the set of variables for which x_i is a merge point which Lemma 3 states that are correctly captured by our mapping γ_{CT}, it is rather straightforward that Lemma 4 holds.

The combination of Lemmas 3 and 4 leads to the following equivalence theorem:

Theorem 2 *Given a DCOP Φ and a CT, the execution of DCPOP(Φ, CT) is equivalent to Action-GDL($\gamma_{CT}(\Phi, PT)$) disregarding the computing and sending of branch information.*

Likewise mapping γ, since the computing of mapping γ_{CT} can be done efficiently and distributedly [19], we can consider the overhead of computing the mapping negligible with respect to the time of solving the DCOP. Therefore, Theorem 2 proves that Action-GDL can be at least as efficient as DCPOP in any DCOP.

5 Characterizing Action-GDL usefulness

From Theorems 1 and 2 we conclude that we can obtain no benefit from using DPOP and DCPOP over Action-GDL. Now the question is: can Action-GDL improve DPOP/DCPOP in terms of (i) the computational/communication needs required from the agents or (ii) the degree of parallelism when solving a DCOP?

Next, in Sect. 5.1 we provide some theoretical results that help answer the first question with respect to DPOP. Moreover, from these theoretical results we obtain some insights regarding how to exploit the space of JTs effectively. Thus, in Sect. 5.2 we propose a postprocessing of JTs to improve the computation, communication and degree of parallelism of a JT. In Sect. 6 we empirically show that such postprocessing helps Action-GDL significantly outperform DCPOP over the best CT/PT generated out of multiple heuristics.

5.1 Theoretical improvements with respect to DPOP

In this section we provide theoretical results showing in which cases Action-GDL can outperform DPOP in terms of communication and computation.

5.1.1 Action-GDL provides significant savings in computation over DPOP when pseudotrees are generated by edge-traversal heuristics

In [3] Atlas and Decker show by means of an example that there exists DCOP instances for which a CT significantly outperforms all possible PTs based on edge-traversal heuristics. Because, by Theorem 2, Action-GDL execution is equivalent to DCPOP execution when it runs over a γ_{CT} mapping JT, Action-GDL can also benefit from this result with respect to DPOP.

5.1.2 Action-GDL provides no significant savings in computation for unrestricted pseudotrees

In this subsection we prove that for any DCOP, given a JT, we can always construct a PT so that the amount of computation for DPOP is of the same order of magnitude than that of Action-GDL.

Lemma 5 *Given a DCOP Φ and a JT, algorithm 4 computes a PT such that the computational requirements of DPOP are of the same order of magnitude than those of Action-GDL (the size of the largest table to be maximized is the same)*

Algorithm 4 JT2PT(JT)

```
1: $\mathcal{C}_k$ = Find the largest clique in $JT$
2: $JT'$ = $JT$ rooted at the agent responsible for $\mathcal{C}_k$
3: $T = GenerateSpanningTree(JT', \emptyset)$;
4: $PT$ = Construct the PT corresponding to $T$;
5: return $PT$
6:
7: function GenerateSpanningTree($JT$,$V$)
8: $\mathcal{C}_k = getRoot(JT)$ /*Let $\mathcal{C}_k$ be the root of $JT$ */
9: $Scope(\mathcal{C}_k) \setminus V = \{x^1, \ldots, x^m\}$ /*Establish an order among the variables in $\mathcal{C}_k$ not in $V$*/
10: $T = \{(x^i, x^{i+1}) | 1 \leq i < m\}$ /*Include into $T$ a chain linking variables in $\mathcal{C}_k$ not in $V$ */
11: for all $JT_i \in Subtree(JT, \mathcal{C}_k)$/*For each subtree of $JT$, one for each child of $\mathcal{C}_k$*/ do
12:     $T_i = GenerateSpanningTree(JT_i, V \cup Scope(\mathcal{C}_k))$
13:     if $Scope(T) \cap Scope(JT_i) \neq \emptyset$ then
14:         $j = \max\{k | 1 \leq k < m$ and $x^k \in JT_i\}$ /*Find $JT_i$ variable with lowest position in $T$*/
15:     else
16:         $j = m$
17:     end if
18:     $T = T \cup T_i \cup \{x^j, Root(T_i)\}$/* Include $T_i$ into $T$ by linking its root as a child of $x^j$ */
19: end for
20: return $T$;
```

Proof First we have to ensure that the tree constructed by algorithm 4 is a PT, that is, we have to check that adjacent nodes in the constraint graph fall in the same branch of the tree. Let x_i and x_j be two adjacent nodes in the constraint graph. By virtue of the covering property, there should be a node of the JT that contains both x_i and x_j. If this is the highest node where both x_i and x_j appear then they will be placed in a chain and hence they will be in the same branch of the tree (lines 9–10). Otherwise, assume without loss of generality that x_i appears in a node in a different branch. By the running intersection property, x_i must appear also in the root of the subtree containing these two nodes. By construction, the branch for x_j will never be inserted into the PT before the appearance of x_i. Hence, both x_i and x_j appear in the same branch (the one that has x_i as root) (line 13–15). Then, our main claim can be proven by induction on the number of variables of JT. If there is a single variable both algorithms are equivalent and hence our result holds. If JT has more than one variable then the computational requirements to run DPOP in the subset of PT composed by the variables of the largest clique (appearing as a chain hanging from the root of the PT) are of $\mathcal{O}(d^m)$ (where m is the size of the clique and d is the highest cardinality of any variable in the clique). By induction hypothesis this is also the case in each of the subpseudotrees hanging from variables in the largest clique. It is easy to see that the size of the largest table to be maximized for Action-GDL is also $\mathcal{O}(d^m)$. □

This result does not mean that the processing of Action-GDL and DPOP will be the same but ensures that the improvement that we can expect from Action-GDL cannot be very large. However, there is no mention on the amount of messages exchanged. In fact, our next result proves that Action-GDL can effectively improve on that.

5.1.3 Action-GDL can severely reduce communication complexity

In this subsection we show that there are DCOPs for which Action-GDL severely reduces the amount of communication with respect to DPOP. Concretely, we prove that when the DCOP

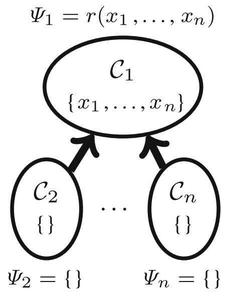


Fig. 5 Best JT

is composed of a single utility relation involving all variables,[6] the amount of communication required grows linearly with the number of variables for Action-GDL using the best JT and grows exponentially for DPOP with any PT.

Lemma 6 *Given a DCOP $\Phi = \langle \mathcal{X} = \{x_1, \ldots, x_n\}, \mathcal{D}, \mathcal{R} = \{r\}\rangle$ such that $Scope(r) = \mathcal{X}$, the amount of communication required to run Action-GDL using the JT depicted in Fig. 5 grows linearly in the number of variables.*

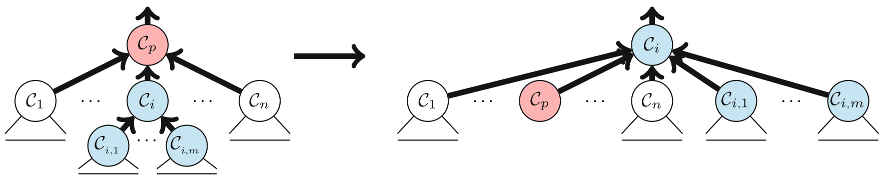
Proof In the utility propagation phase $x_2, \ldots, x_n$ send empty messages to x_1. Then x_1 computes the overall solution and distributes the decisions to $x_2, \ldots, x_n$ in the value propagation phase, exchanging $n - 1$ messages, the largest of them being of size $\log d$, where $d = \max_i |D_i|$. Hence the amount of communication is $\mathcal{O}(n \log d)$. □

Lemma 7 *Given a DCOP $\Phi = \langle \mathcal{X} = \{x_1, \ldots, x_n\}, \mathcal{D}, \mathcal{R} = \{r\}\rangle$ such that $Scope(r) = \mathcal{X}$, the amount of communication required for DPOP independently of the PT grows exponentially in the number of variables.*

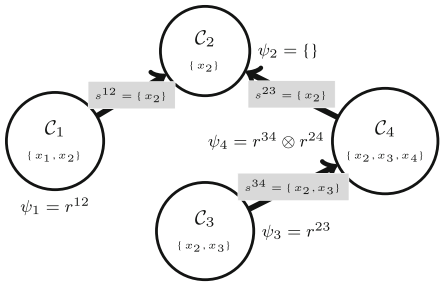
Proof First note that the only possible structure for a PT is a chain, because otherwise adjacent vertices in the graph will appear in different branches (since all vertices are adjacent). There are as many PTs as variable orderings. Assume without loss of generality that the ordering places x_1 in the root, then x_2 as its child and so on until x_n as a single leave. DPOP places the relation r in x_n. The execution starts maximizing r with respect to x_n. The computed relation is sent to x_{n-1} which maximizes it with respect to x_{n-1} and the process continues that way until it reaches x_1. Then the best value for x_1 is computed and sent to x_2 where the best value for x_2 is computed and sent to x_3 together with the optimal value for x_1 and the process continues that way until it reaches x_n. The algorithm exchanges $n - 1$ utility messages, the largest of them of size d^{n-1} and $n - 1$ value messages, the largest of them of size $\sum_{i=1}^{n-1} \log |D_i|$. Hence, the overall amount of communication is $\mathcal{O}(d^n)$. □

Lemmas 6 and 7 prove that Action-GDL can severely improve DPOP communication complexity. Furthermore, it suggests that for more complex graphs, the improvement could be related to the treewidth of the constraint graph.

[6] Note that the relation containing all variables does not result from any partial centralization of the algorithm, instead it is formulated like this in the original DCOP.

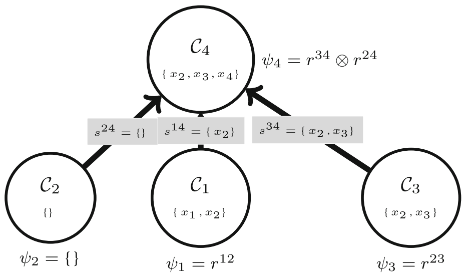


(a) Transforming a JT by clique swap.



$\mathcal{C}_2$ $\{x_2\}$
$\psi_2 = \{\}$
$s^{12} = \{x_2\}$
$s^{23} = \{x_2\}$
$\mathcal{C}_1$ $\{x_1, x_2\}$
$\psi_1 = r^{12}$
$\psi_4 = r^{34} \otimes r^{24}$
$\mathcal{C}_4$ $\{x_2, x_3, x_4\}$
$s^{34} = \{x_2, x_3\}$
$\mathcal{C}_3$ $\{x_2, x_3\}$
$\psi_3 = r^{23}$

(b) $\mathcal{C}_4$ swapped for $\mathcal{C}_3$.



$\mathcal{C}_4$ $\{x_2, x_3, x_4\}$
$\psi_4 = r^{34} \otimes r^{24}$
$s^{24} = \{\}$
$s^{14} = \{x_2\}$
$s^{34} = \{x_2, x_3\}$
$\mathcal{C}_2$ $\{\}$
$\psi_2 = \{\}$
$\mathcal{C}_1$ $\{x_1, x_2\}$
$\psi_1 = r^{12}$
$\mathcal{C}_3$ $\{x_2, x_3\}$
$\psi_3 = r^{23}$

(c) $\mathcal{C}_4$ swapped for $\mathcal{C}_2$.

Fig. 6 **a** Postorder transformation and **b,c** transformations of JT of Fig. 3c

5.2 Postprocessing junction trees

Taking inspiration on Lemmas 6 and 7, we propose to postprocess the JT constructed by the γ_{CT} mapping to reduce the amount of computation, the sizes of messages and the degree of parallelism. Firstly, in order to reduce the amount of computation and the size of messages we propose to exchange two connected cliques in the JT, namely $\mathcal{C}_i$ and its parent $\mathcal{C}_p$, following the transformation depicted in Fig. 6, whenever the set of variables in $\mathcal{C}_p$ is a subset of $\mathcal{C}_i$. Formally:

$$\mathcal{C}_p \subseteq \mathcal{C}_i \tag{14}$$

After swapping parent for child, the child takes its parent's children but keeping also its own children as depicted on the right hand side of Fig. 6. The intuition behind the transformation is straightforward. Since the structure in Fig. 5 is the best one for processing a clique with Action-GDL, whenever there is a clique whose variables are included into one of its children, we can think of swapping parent for child. Figure 6a, b depict the two transformations carried out by our postprocessing over the JT in Fig. 3c. The first transformation only swaps $\mathcal{C}_4$ and $\mathcal{C}_3$ without involving any deeper change. The second transformation entails a more profound rearrangement because in order to swap $\mathcal{C}_4$ for $\mathcal{C}_2$, $\mathcal{C}_4$ must keep $\mathcal{C}_3$ as a child.

Notice that if we start from a valid JT, the resulting JT after this transformation still satisfies the running intersection property (RIP) without increasing any clique. Furthermore, it is likely that cliques can be reduced after the swap by deleting some variables not longer necessary to ensure the RIP. Concretely, after the transformation clique C_p can be restricted to deal only with variables in the scope of its potential, that is $Scope(\psi_p)$, thus reducing the amount of computation and size of messages for C_p. That is because after a swap, C_p is always a leaf node so it will not have to enlarge its clique to carry variables to satisfy the RIP. Thus, in the example of Fig. 6c, as a consequence of the change of position, $\mathcal{C}_2$ can delete x_2 from its set of variables. To summarise, our postprocessing performs a postorder tree traversal of the JT computed by γ_{CT}, applying the transformation depicted in Fig. 6 whenever the condition in Eq. 14 holds. Hence, its distributed implementation is direct [16].

Secondly, after the postorder traversal, we select the root of the JT that maximises the degree of parallelism (the maximum amount of sequential computation required by agents when running Action-GDL). This last step is important because although changing the root of a JT does not change the amount of computation nor of messages exchanged, it can modify its degree of parallelism.

Observe that the resulting JT in Fig. 6c reduces communication, computation, and improves parallelism with respect to the original JT in Fig. 3a.

5.2.1 Postprocessing complexity

In what follows we assess the complexity of the postprocess methods described above. Firstly, in the postorder tree traversal the information exchanged is $O(n^2)$ (each node that swaps exchanges messages with all its neighbours) and the overall computation is $O(n^2)$ where n is the number of variables in the DCOP. Secondly, to distributedly select the root of the PT, agents can execute a distributed leader election algorithm [4] which information exchanged and overall computation is $O(n)$. Therefore we can conclude that: (1) the postprocessing can be computed distributedly, and; (2) the overhead introduced is not significant with respect to the costs of solving the DCOP.

6 Empirical evaluation

In this section we aim at providing evidence that using Action-GDL instead of DCPOP (or DPOP) is useful from a practical point of view. In [3] Atlas and Decker provide empirical evidence of the significant improvements that DCPOP can obtain when compared to DPOP. Since ActionGDL generalizes DCPOP, it can also benefit from the same improvements with respect to DPOP. Thus, in our experiments, we directly compare Action-GDL with DCPOP.

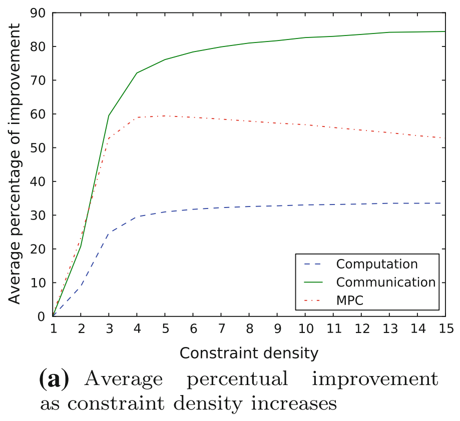
6.1 Measures of interest

We are interested in comparing DCPOP and Action-GDL regarding the amount of communication, computation, and parallelism required in an experimental scenario. Since we have proved that Action-GDL is a generalization of DCPOP, the metrics defined below for Action-GDL can be readily used for DCPOP.

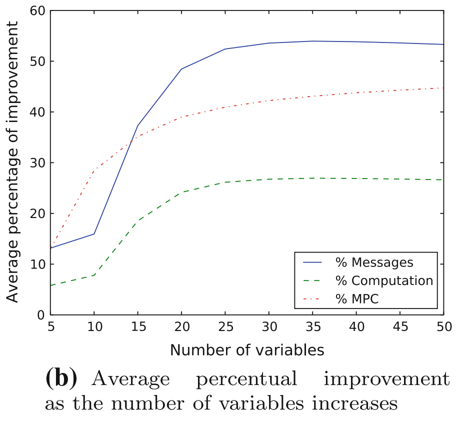
Computation. The amount of computation at node i is assessed as the sum of the product of the domains' cardinality of variables in its clique, $MC_i = \prod_{x_k \in \mathcal{C}_i} |\mathcal{D}_k|$. The total amount of computation is $\sum_{i=1}^{n} MC_i$.

Communication. The size of a utility message μ_{ij} is $\prod_{x_k \in s_{ij}} |\mathcal{D}_k|$. As noted in [3], most communications in DCPOP are utility messages. This is also true for Action-GDL. Hence, we have disregarded value messages in our comparison because they only add a small constant factor. As with computation, we assess the overall amount of communication by adding the size of every message.

Parallelism. Since both DCPOP and Action-GDL are distributed algorithms, we are also interested in the degree of parallelism that we can obtain in its processing. Following [3], we measure the degree of parallelism using the maximum path cost (MPC) that measures the maximum amount of sequential computation to perform. The maximum path cost for a



(a) Average percentual improvement as constraint density increases



(b) Average percentual improvement as the number of variables increases

Fig. 7 Action-GDL improvement over DCPOP in computation, communication and MPC

given JT is defined as $MPC = \max_i \sum_{\mathcal{C}_j \in P_i} MC_j$ where P_i is the path from the root of the JT to clique $\mathcal{C}_i$.

6.2 Experimental design and results

In the experiments we use four heuristics to generate DCPOP arrangements: (1) DFS-MCN (Depth-First Search Maximum Connected Node) heuristic [14], which generates PTs; and (2) BFS-MCN (Best-First Search Maximum Connected Node), BFS-LCN (BFS Less Connected Node) and BFS-A-B (BFS Ancestors\ Branch-parents\Branch-children rule) heuristics [3] that generate CTs.

For DCPOP we chose the best arrangement produced by these heuristics. These PTs/CTs are subsequently input to the γ_{CT} mapping to generate JTs which are further postprocessed as explained in Sect. 5.2 to obtain the input for Action-GDL. For Action-GDL we chose the best JT produced by this post-processing.

We empirically compare DCPOP with Action-GDL by plotting the average of the percentual improvement of ActionGDL with respect to DCPOP for each metric. We assess the percentual improvement as $P = \frac{(A-D)}{(A+D)} \cdot 200$, where A is the value of the chosen metric for ActionGDL and D stands for the value for DCPOP.

6.3 Generic DCOP instances

Our initial tests perform a comparison over randomly generated DCOPs with binary variables. We analyse the differences between DCPOP and Action-GDL as we increase the number of constraints as well as the number of variables. Thus, we characterize each scenario by a number of variables n and a constraint density d. For each scenario, we generate 10.000 random problems. We have explored scenarios with n ranging from 10 to 100 in 10 steps increments and d ranging from 1 to 15 in 1 step increment.

Figure 7 summarizes our experimental results. Figure 7a shows the average of percent improvement among tests as the constraint density increases. We observe that the denser the DCOP, the larger the improvement of Action-GDL regarding communication and computation with respect to DCPOP. Concretely, Action-GDL reduces communication up to around 85%. The amount of computation is not reduced so significantly, though we still obtain

Table 2 Results for the different scenarios of the meeting scheduling dataset

Scenario	Meetings	# Var.	# Dom	Den.	Comp. (%)	Comm. (%)	MPC (%)
A/1	8	23	9	1.9	2.3	22.6	5.6
B/2	10	26	9	1.8	9.2	134.2	3.0
C/3	12	71	9	1.7	2.8	28.0	31
D/4	12	72	9	1.7	2.3	23.0	21.4

average percent improvements of around 30%. We also measured the computation and communication improvement as in [3] in terms of the average of the difference in the number of dimensions. Using these metrics the experiments show improvements up to 10 dimensions. With respect to the improvement on computation and communication, the improvement on the degree of parallelism behaves differently: observe that the MPC reaches the highest value, around 60%, when density is set to 4, and after that it smoothly decreases up to around 50%. That result is explained because in denser DCOPs it is more likely that there is a single clique with a larger number of dimensions than others. Then, this clique conditions the MPC no matter the arrangement. The difference reported for these metrics between ActionGDL and DCPOP is statistically significant within a single value of density (paired Student's t-tests calculate $p < 0.05$) except for density 1.

Moreover, we also show in Fig. 7b the average of percent improvement as the number of variables increases. We observe that Action-GDL reaches the higher computation/communication improvement with respect to DCPOP, around 80% and 30% respectively, in medium size DCOPs (with 30–40 variables). As the number of variables increases the average of improvements tend to around 70% and 28% on respectively. In terms of the average of the difference in the number of dimensions, these results implies an improvement of up to 16 dimensions. With regard to the degree of parallelism, we observe that MPC increases with the number of variables up to 55%. We run paired Student's t-tests and the difference between the metrics between ActionGDL and DCPOP is statistically significant within a single value of variables ($p < 0.05$).

To sum up, the cost of solving random DCOPs is significantly reduced respect to DCPOP when running Action-GDL over the postprocessed JTs mapped from best CTs. Next we show that such improvement is specially significant for dense problems.

6.4 Meeting scheduling dataset

Besides the generic DCOP tests, we also run additional tests on a meeting scheduling dataset, a common problem used by the DCOP community. Concretely, we use the meeting scheduling dataset from [10], publicly available in [22]. This dataset is composed of four scenarios (labeled as A/1,B/2,C/3 and D/4), which correspond to four different topologies, with 30 different instances per scenario.

Table 2 shows the results for the meeting scheduling dataset as well as the characteristics of each scenario (the number of variables/constraints, the cardinality of the variables' domain, etc). All scenarios are composed of sparse problems with a constraint density lower than 2. The results obtained in the meeting scheduling dataset are in line with those obtained for generic DCOPs for similar scale and density and are also statistically significant ($p < 0.05$ for all paired Student's t-tests). Firstly, the MPC is around 3–10% in small scenarios (8–10 variables) and around 20–30% for larger scenarios (70 variables). Thus, as shown in Fig. 7b for random instances, the MPC increases with the number of variables. With regard to the improvement on computation, it is less than 10% in all scenarios, with similar values to those shown in Fig. 7a when density is set to 2. Finally, the improvements on communication are, in most scenarios, close to those reported in Fig. 7a for random instances of density 2, with average percent improvements of around $20 - 30\%$. However, in scenario $C/3$ we obtain a much higher average percentual improvement. Therefore, although one can characterize the average improvement given the density and the scale of the problem, we observe that the topology of the constraint graph is also an important factor. In particular, our results showed that the communication improvement on some structured topologies is significantly larger than on random ones.

7 Conclusions and future work

In this paper we have presented Action-GDL, a specialisation of GDL to efficiently solve DCOPs. We have shown that Action-GDL is a promising framework that unifies several dynamic programming DCOP algorithms that are based on the General Distributive Law (GDL). In particular, we show that it is the case of DPOP and DCPOP. On the one hand, we show that Action-GDL generalises DPOP. Furthermore, we provide some theoretical results that prove that moving from pseudotrees (used by DPOP) to junction trees (used by Action-GDL) leads to significant benefits in terms of computation and communication. With regard to DCPOP, we show that Action-GDL can mimick any DCPOP execution by providing a mapping from cross-edged pseudotrees (used by DCPOP) to junction trees. To exploit the space of junction trees we propose a distributed heuristic to post-process them. We empirically showed that Action-GDL significantly outperforms DCPOP when running over the junction trees that result from post-processing the best cross-edged pseudotrees DCPOP can operate on. Finally, we argue that, from an analytical point of view, this unifying perspective provided by Action-GDL builds a bridge with a wealth of theoretical results for GDL over junction trees [1] from which Action-GDL may benefit.

Regarding future work, we consider several directions. Firstly, since the efficiency of Action-GDL depends on the underlying junction tree, a natural line of future research is to study the potential of the existing junction trees heuristics [5,2,7] in a distributed environment. Secondly, although we provide a mapping from cross-edged pseudotrees to junction trees, the issue of whether all junction trees can be represented as cross-edged pseudotrees remains open. Thus, we leave open the intriguing question of whether the space of junction trees is larger than the space of cross-edged trees or instead there exists an isomorphism between both spaces. Thirdly, we plan to analyze the privacy aspects of Action-GDL, which can limit its applicability to some domains, such as distributed scheduling, where privacy is the main issue. Finally, given the multiple extensions formulated to DPOP [14] (e.g H-DPOP or MB-DPOP) to tailor it to different domains, future work should also consider to formulate such extensions in the GDL framework.

Acknowledgements Work funded by IEA (TIN2006-15662-C02-01), AT (CONSOLIDER CSD2007-0022, INGENIO 2010) and EVE (TIN2009-14702-C02-01 and 02). Vinyals is supported by the Spanish Ministry of Education (FPU grant AP2006-04636).

References

1. Aji, S. M., & McEliece, R. J. (2000). The generalized distributive law. *IEEE Transactions on Information Theory, 46*(2), 325–343.
2. Amir, E. (2001). Efficient approximation for triangulation of minimum treewidth. In: *UAI*, pp. 7–15.

3. Atlas, J., & Decker, K. (2007). A complete distributed constraint optimization method for non-traditional pseudotree arrangements. In: *AAMAS*, pp. 741–784.
4. Barbosa, V. (1996). *An introduction to distributed algorithms*. Cambridge: The MIT Press.
5. Cano, A., & Moral, S. (1994). *Heuristic algorithms for the triangulation of graphs*. In: *IPMU*, pp. 98–107.
6. Dechter, R. (2003). *Constraint processing*. San Francisco: Morgan Kaufmann.
7. Flores, M. J., Gámez, J. A., & Olesen, K. G. (2003). Incremental compilation of bayesian networks. In: *UAI*, pp. 233–240.
8. Jensen, F. V., & Jensen, F. (1994). Optimal junction trees. In: *UAI*, pp. 360–366.
9. Junges, R., & Bazzan, A. L. C. (2008). *Evaluating the performance of DCOP algorithms in a real world, dynamic problem*. In: *AAMAS* (Vol. 2, pp. 599–606).
10. Maheswaran, R. T., Tambe, M., Bowring, E., Pearce, J. P., & Varakantham, P. (2004). Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling. In: *AAMAS*, pp. 310–317.
11. Mailler, R., & Lesser, V. R. (2004). *Solving distributed constraint optimization problems using cooperative mediation*. In: *AAMAS*, pp. 438–445.
12. Modi, P. J., Shen, W. M., Tambe, M., & Yokoo, M. (2005). Adopt: Asynchronous distributed constraint optimization with quality guarantees. *Artif Intell, 161*(1–2), 149–180.
13. Pearl, J. (1988). *Probabilistic reasoning in intelligent systems*. San Francisco, CA: Morgan Kaufmann Publishers Inc.
14. Petcu, A. (2007). *A class of algorithms for distributed constraint optimization*. PhD thesis, EPFL, Lausanne.
15. Petcu, A., & Faltings, B. (2005). *A scalable method for multiagent constraint optimization*. In: *IJCAI*, pp. 266–271.
16. Santoro, N. (2006). *Design and analysis of distributed algorithms*. (Wiley series on parallel and distributed computing) Chicheste: Wiley-Interscience.
17. Shafer, G., & Shenoy, P. P. (1990). Probability propagation. *Ann Math Artif Intell, 2*, 327–351.
18. Vinyals, M., Rodriguez-Aguilar, J., & Cerquides, J. (2008). *Proving the equivalence of Action-GDL and DPOP*. http://www.iiia.csic.es/files/pdfs/TRR200804.pdf.
19. Vinyals, M., Rodriguez-Aguilar, J. A., & Cerquides, J. (2009). Generalizing DPOP: Action-GDL, a new complete algorithm for DCOPs. In: *OPTMAS second international workshop on optimization in agent systems*.
20. Viterbi, A. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory, 13*(2), 260–269.
21. Yeoh, W., Felner, A., & Koenig, S. (2009). Idb-adopt: A depth-first search dcop algorithm. In: *CSCLP, Lecture notes in computer science 5655, 2009*, pp 132–146.
22. Yin, Z. (2008). *USC dcop repository*. http://teamcore.usc.edu/dcop.
23. Zhang, W., Wang, G., Xing, Z., & Wittenburg, L. (2005). Distributed stochastic search and distributed breakout: Properties, comparison and applications to constraint optimization problems in sensor networks. *Artif Intell, 161*(1–2), 55–87.