Communication-constrained DCOPs: Message approximation in GDL with function filtering

Marc Pujol-Gonzalez, Jesus Cerquides, Pedro Meseguer, J. A. Rodriguez-Aguilar
Artificial Intelligence Research Institute (IIIA)
Spanish Scientific Research Council (CSIC)
Campus de la UAB, Bellaterra, Spain
{mpujol, cerquide, pedro, jar}@iiia.csic.es

ABSTRACT

In this paper we focus on solving DCOPs in communication constrained scenarios. The GDL algorithm optimally solves DCOP problems, but requires the exchange of exponentially large messages which makes it impractical in such settings. Function filtering is a technique that alleviates this high communication requirement while maintaining optimality. Function filtering involves calculating approximations of the exact cost functions exchanged by GDL. In this work, we explore different ways to compute such approximations, providing a novel method that empirically achieves significant communication savings.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*intelligent agents, multiagent systems*

General Terms

Algorithms, Design

Keywords

Distributed constraint reasoning, DCOP, GDL, Filtering

1. INTRODUCTION

Distributed constraint optimization (DCOP) is a model for representing multi-agent systems in which agents cooperate to optimize a global objective. There are several complete DCOP algorithms that guarantee global optimality such as ADOPT [11], DPOP [13], and its generalization GDL [1, 15]. Since DCOPs are NP-Hard [11], solving them requires either an exponential number of linear size messages (ADOPT) or a linear number of exponentially large messages (DPOP, GDL). Nonetheless, some application domains are specially communication constrained. For instance, data transmission is severely limited in wireless sensor networks [17], and bandwidth is a scarce resource in peer-to-peer networks [6]. An approach in these domains is to drop optimality in favor of lower complexity, approximate algorithms with weaker guarantees [7, 9, 16]. As an alter-

native, we aim at reducing the communication costs while keeping optimality.

Function filtering [2, 3] is a technique that reduces the size of exchanged messages, and can be readily applied to GDL. Essentially, GDL with function filtering exchanges approximations of the messages that would be sent by GDL. As a consequence, the resulting algorithm's efficiency is highly dependent on the strategy used to compute these approximations. Intuitively, *better* strategies produce better approximations, leading to larger communication savings.

We first review state-of-the-art approximation methods [5, 14], fitting them in a common framework of bottom-up approximations. However, these methods are designed for the centralized case. Hence, their purpose is to reduce the overall computation, disregarding any communication costs.

Our main contribution in this paper is a novel class of message approximation techniques, the top-down approximations, that are specifically aimed at lowering such communication costs. Thereafter, we present two realizations of this new approach, namely brute-force decomposition and zero-tracking decomposition. Finally, we empirically evaluate the overall computation time and communication costs of these methods on several experiments. The results show that top-down approximations outperform bottom-up ones, always achieving larger communication savings. Further, zero-tracking decomposition remains competitive in computational effort with respect to state-of-the-art approximation methods [5, 14] while transmitting much less information.

This paper is structured as follows. Firstly, Section 2 introduces the DCOP model, and Section 3 presents the GDL algorithm with function filtering. Next, Section 4 describes the message approximation problem we aim to solve. Then, Section 5 introduces the bottom-up approximation framework, which represents current state-of-the-art approaches to message approximation. Next, Section 6 introduces our novel top-down approximation framework. Section 7 provides an empirical evaluation of both bottom-up and top-down approximation methods. Finally, Section 8 draws the most important conclusions of this work.

2. DCOP

Distributed Constraint Optimization Problems (DCOPs) involve a finite set of variables, each taking a value in a finite discrete domain. Variables are related by cost functions that specify the cost of assigning certain values to a subset of variables. Costs are positive real numbers plus 0 and ∞. A DCOP is defined as a tuple (X, D, C, A, α), where $X = \{x_1, \ldots, x_n\}$ is a set of n variables; $D = \{D(x_1), \ldots, D(x_n)\}$

is a set of finite discrete domains; $D(x_i)$ is the finite set of x_i's possible values; C is a set of cost functions; each cost function $f_S \in C$ is defined on the ordered set $S \subseteq X$ (S is f_S's scope, $|S|$ is f_S's arity), specifies the cost of every combination of values of variables in S, namely $f_S : \prod_{x_j \in S} D(x_j) \mapsto \mathbb{R}^+$; A is a set of p agents; and $\alpha : X \to A$ maps each variable to some agent.

The objective of DCOP algorithms is to find the assignment of individual values to variables, such that the total (aggregated) cost over all cost functions is minimized. We make the common assumption that there are as many agents as variables, each agent controlling one variable, so from now on the terms variable and agent will be used interchangeably.

Next, we introduce some definitions of concepts and operations that will be used throughout the rest of this paper. A *tuple* t_S, with scope S, is an ordered set of values assigned to each variable in the subset $S \subseteq X$. The *cost* of a complete tuple t_X that assigns a value to each variable $x_i \in X$ is the addition of all individual cost functions evaluated on that particular tuple. Whenever a complete tuple t_X yields a cost lower than a user-specified threshold, we say that it is a *solution*. Further, a solution t_X is an *optimal solution* if its cost is the minimum of all possible solutions.

DEFINITION 1   (MIN-MARGINAL). *The projection $t_S[T]$ of a tuple t_S to $T \subset S$ is a new tuple t_T, removing the values assigned to variables not found in T. The min-marginal $f_S[T]$ of a cost function f_S onto $T \subset S$ is a new cost function f_T where each tuple t_T is assigned the minimum cost among all tuples t_S whose projection to T is t_T.*

$$\forall t_T \qquad f_T(t_T) = \min_{t_S[T]=t_T} f_S(t_S[T]).$$

DEFINITION 2   (COMBINATION). *The combination of two cost functions f_S and f_T, written $f_S \bowtie f_T$, is a new cost function f_U defined over their joint domain $U = S \cup T$, s.t.:*

$$\forall t_U \quad (f_S \bowtie f_T)(t_U) = f_S(t_U[S]) + f_T(t_U[T])$$

Combination is an associative and commutative operation.

DEFINITION 3   (LOWER BOUND). *A function f_T is a lower bound of f_S, noted $f_T \leq f_S$, iff $T \subseteq S$ and for all t_S tuples, the value by f_T of its projection $t_S[T]$ is lower than or equal to the value by f_S of t_S:*

$$\forall t_S \qquad f_T(t_S[T]) \leq f_S(t_S).$$

Given two lower bounds $f_T, f_U \leq f_S$, we say that f_T is at least as good as f_U iff $f_U \leq f_T$.

3. GDL

Several algorithms can optimally solve DCOPs [11, 13]. In particular, we consider the GDL algorithm [1], following the Action-GDL description [15]. GDL works over a special structure named junction tree (JT) [8], also known as joint tree or cluster tree. JTs represent a decomposition of the DCOP objective function into an equivalent, partially ordered sequence of combinations and marginalizations that are computationally easier to perform. It is partially ordered because those operations at the same tree-level can be carried out in parallel. Further, GDL is easily applicable to distributed solving because, given a DCOP where each agent holds a different variable, a JT can be computed in distributed form [12]. Additionally, in the distributed case, JTs also represent the communication links that are going to be exploited, and what information they are going to exchange. As a result, each node of the JT (also known as clique) represents an agent operating over a subset of problem's variables and cost functions, whereas edges represent communication links that nodes will use by exchanging marginalizations of their problem parts onto their shared variables.

In short, the full serial version of GDL for the all-vertices problem (also known as DCTE [2]) works as follows [1, 15]: it sends messages up and down the JT, two messages per edge of the JT. Each message contains a cost function that summarizes the effect of the JT part from which this message comes on the considered agent. After exchanging these messages, each agent contains enough information to optimally solve its subproblem. However, it may happen that two optimal solutions t_1 and t_2 (with the same cost) exist, so some agents would choose t_1 while others choose t_2. This may lead to assigning two different values to the same variable. To prevent this, the JT root decides the optimal assignment and informs its children in the JT, which recursively inform their own children and so on. In this way, a coherent optimal solution is selected. For a detailed description of GDL's operation, consult [1, 15].

Limiting Message Size. A major drawback of GDL is that the size of exchanged messages is exponential with respect to the number of variables in the JT edges (s). A strategy to alleviate this fact is to relax GDL to an approximate form, such that the size of messages can not exceed $exp(r)$, where $r < s$. This approach was first introduced in [2], where it is named DMCTE(r). Since messages' sizes are now limited, DMCTE(r) does not necessarily compute the optimal solution anymore. Nevertheless, it provides an approximate solution, as well as an interval [lower bound, upper bound] limiting the optimum cost. In general, the larger the r-arity parameter, the better the solution quality, but the higher the communication cost. Hence, the algorithm can also be iterated by slowly increasing r until an acceptable (or optimal when the lower bound equals the upper bound) solution is found.

Function Filtering. Function filtering [2, 3] is a technique to further reduce messages' sizes. Remember that messages are formed by cost functions (f of arity s in GDL, f' of arity r in DMCTE(r)) represented as tuples t, and their associated cost $f(t)$. The idea is to avoid sending those tuples t that, when combined with other functions, will have a cost equal to or higher than the current upper bound.

To see how this can be done, imagine the following situation: let t be a tuple that should be sent from agent i to agent j, and let UB be an upper bound of the global cost. After t arrival, agent j may realize that all possible combinations of t with its own cost functions reach or exceed the UB, so t can not be part of an optimal solution. In this situation, actually sending t is useless. Hence, if i was able to detect such irrelevant tuples, it could avoid sending them altogether, further reducing its message size. In GDL, function f containing t should be added with g, a function formed by combining all cost functions of agent j plus all cost functions received by agent j except the one from agent i. In the DMCTE(r) approximation, function f' containing t is a lower bound of the exact function f. If agent i knows any lower bound $g' \leq g$ before sending f', it can detect some tuples that exceed the UB cost, and therefore avoid sending them. Removing such useless t tuples is called *filtering f'*

GDL (DCTE)	DMCTE($r = 1$)	DIMCTEf
		Iteration 1: DMCTE($r = 1$)
		Iteration 2:
$CF(C_2 \to C_1)$: $f_2[S_1] = g_1 = \begin{array}{c|c} y & \\ \hline a & 1 \\ b & 2 \end{array}$	$CF(C_2 \to C_1)$: $f_2[S_1] = g_1 = \begin{array}{c|c} y & \\ \hline a & 1 \\ b & 2 \end{array}$	$CF(C_2 \to C_1)$: $\overline{f_2[S_1]}^{g_3} = g_1 = \begin{array}{c|c} y & \\ \hline a & 1 \\ b & 2 \end{array}$
$CF(C_3 \to C_1)$: f_3	$CF(C_3 \to C_1)$: $\begin{cases} \text{a unary lower bound of } f_3, \\ \text{i.e. } f_3[y] = g_2 = \begin{array}{c|c} y & \\ \hline a & 3 \\ b & 1 \end{array} \end{cases}$	$CF(C_3 \to C_1)$: $\overline{f_3}^{g_4} = \begin{array}{cc|c} y & z & \\ \hline a & a & 3 \\ b & b & 1 \end{array} = \begin{array}{cc|c} y & z & \\ \hline a & a & 3+1 \\ a & b & 5+1 \geq UB \\ b & a & 5+4 \geq UB \\ b & b & 1+4 \end{array}$
$CF(C_1 \to C_2)$: $(f_1 \bowtie f_3)[S_1] = \begin{array}{c|c} y & \\ \hline a & 5 \\ b & 3 \end{array}$	$CF(C_1 \to C_2)$: $(f_1 \bowtie g_2)[S_1] = g_3 = \begin{array}{c|c} y & \\ \hline a & 3 \\ b & 3 \end{array}$	$CF(C_1 \to C_2)$: $\overline{(f_1 \bowtie f_3)[S_1]}^{g_1} = \begin{array}{c|c} y & \\ \hline b & 3 \end{array} = \begin{array}{c|c} y & \\ \hline a & 5+1 \geq UB \\ b & 3+2 \end{array}$
$CF(C_1 \to C_3)$: $f_1 \bowtie g_1 = \begin{array}{cc|c} y & z & \\ \hline a & a & 4 \\ a & b & 1 \\ b & a & 7 \\ b & b & 4 \end{array}$	$CF(C_1 \to C_3)$: $\begin{cases} \text{a unary lower bound of } f_1 \bowtie g_1, \\ \text{i.e. } (f_1 \bowtie g_1)[y] = g_4 = \begin{array}{c|c} y & \\ \hline a & 1 \\ b & 4 \end{array} \end{cases}$	$CF(C_1 \to C_3)$: $\overline{f_1 \bowtie g_1}^{g_2} = \begin{array}{cc|c} y & z & \\ \hline a & b & 1 \\ b & b & 4 \end{array} = \begin{array}{cc|c} y & z & \\ \hline a & a & 4+3 \geq UB \\ a & b & 1+3 \\ b & a & 7+1 \geq UB \\ b & b & 4+1 \end{array}$
$SS(C_1 \to C_2)$: $\{y = b\}$ $SS(C_1 \to C_3)$: $\{y = b, z = b\}$	$SS(C_1 \to C_2)$: $\{y = a\}$ $SS(C_1 \to C_3)$: $\{y = a, z = b\}$	$SS(C_1 \to C_2)$: $\{y = b\}$ $SS(C_1 \to C_3)$: $\{y = b, z = b\}$
	$BB(C_2 \to C_1)$: $lb = 4, pub = 1$ $BB(C_3 \to C_1)$: $lb = 4, pub = 5$	$BB(C_2 \to C_1)$: $lb = 5, pub = 2$ $BB(C_3 \to C_1)$: $lb = 5, pub = 1$
	$BB(C_1 \to C_2)$: $lb = 4, pub = 5$ $BB(C_1 \to C_3)$: $lb = 4, pub = 1$	$BB(C_1 \to C_2)$: $lb = 5, pub = 3$ $BB(C_1 \to C_3)$: $lb = 5, pub = 4$

Figure 1: Sequence of messages exchanged by GDL, DMCTE($r = 1$) and DIMCTEf solving the instance of Figure 2. Messages in the same cell are exchanged in parallel.

with g', noted $\overline{f'}^{g'}$.

Finally, notice that if we iterate DMCTE(r) with increasing r values, the function g' sent from agent j to agent i in the previous iteration is a lower bound of g. Hence, agent i can readily use g' to filter f' before sending it at the current iteration, leading to a lower size message. The resulting algorithm is known as DIMCTEf, consisting of three phases for each iteration: (1) Cost propagation: agents exchange approximate cost functions (CF messages), using cost functions of the previous iteration to filter cost functions at the current iteration. (2) Solution propagation: values for the variables in the separators of the JT are decided in a top-down manner (SS messages). (3) Bound propagation: agents exchange global lower bounds and local upper bounds (BB messages) among nodes of the JT, following the same communication strategy as cost propagation. For a detailed description of each phase and the structure of the different message types, the reader should consult [2].

Example. The toy example of Figure 2 allows us to illustrate the behavior of the above mentioned algorithms. Figure 1 shows messages exchanged by GDL (left), DMCTE($r = 1$) (middle) and DIMCTEf (right). GDL performs exact solving. CF messages contain cost functions, while SS messages contain assignments of variables propagated top-down in the JT. After receiving them, each agent is able to compute the optimum cost (5) and the same global solution ($xyz \leftarrow bbb$). DMCTE($r = 1$) performs approximate solving, where only cost functions of arity 1 ($r = 1$) can be exchanged (when sending/receiving cost functions to/from C_3, since initial cost functions are binary they are approximated by unary lower bounds). BB messages contain a lower bound lb and a partial upper bound pub of the cost of the solution propagated by SS messages. After receiving them, each agent is able to compute a lower bound (4) of the optimum cost, a global upper bound (6) and the
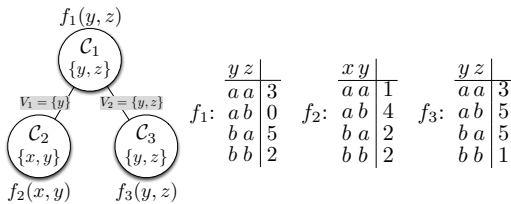
same solution of that cost ($xyz \leftarrow aab$). DIMCTEf performs exact solving with function filtering. The first iteration is DMCTE($r=1$), only unary cost functions can be sent. The second iteration, when also binary cost functions can be sent, uses cost functions of the first iteration (g_1, g_2, g_3, g_4) to filter current cost functions (the filtering process is shown between brackets). As result, some tuples are removed because they reach or exceed the UB computed at the previous iteration (6). Since the allowed arity equals the largest separator size, at the end of this iteration each agent is able to compute the minimum cost (5) and the same optimal solution ($xyz \leftarrow bbb$).

4. MESSAGE APPROXIMATIONS

During the cost propagation phase, GDL's task is to propagate cost functions in the form of min-marginals. DMCTE(r) relaxes this phase by sending lower-bound approximations instead of exact min-marginals, so that less information needs to be sent. Therefore, the more accurate these approximations are, the better results DMCTE(r) is going to achieve at the same iteration. Further, because greater accuracy means that function filtering will be able to prune more tuples, increasing the accuracy should also lower the total amount of communication needed to solve the problem.

Since we are interested in communication-constrained scenarios, from now on the bound r means that agents can not send functions of more than r variables. However, agents can compute functions of any arity. Consider an agent operating in DMCTE(r). Eventually, it will receive messages from all its children in the JT. Then, the agent combines this information with its own and marginalizes it onto the variables in the separator, to send the result to its parent. Nevertheless, since the agent is now constrained by the arity limit r, it can not send the exact min-marginal and it has to compute an approximation. Hence, the objective of the approximation task is to find a good lower bound for the min-marginal while communicating only functions of at most r variables. Some algorithms for this task have already been proposed in the literature [5, 14]. In the next section we review them and we fit them into a common framework which we call bottom-up approximation. In order to do that precisely, we need to introduce some additional definitions.

In the following, let $F = \{f_{T_1}, \ldots, f_{T_n}\}$ be a set of functions, V a set of variables and r an arity limit.

Figure 2: Toy example and a simple junction tree.

$$F = \{f_{xt}, f_{yt}, f_{zt}\} \qquad f_{xt}: \begin{array}{cc|c} x & t & \\ \hline a & a & 2 \\ a & b & 1 \\ b & a & 3 \\ b & b & 2 \end{array} \qquad f_{yt}: \begin{array}{cc|c} y & t & \\ \hline a & a & 4 \\ a & b & 1 \\ b & a & 2 \\ b & b & 2 \end{array} \qquad f_{zt}: \begin{array}{cc|c} z & t & \\ \hline a & a & 0 \\ a & b & 2 \\ b & a & 0 \\ b & b & 1 \end{array}$$

$$V = \{x, y, z\}$$
$$r = 2$$

Figure 3: Example of functions to approximate.

We define $\bowtie F$, *the combination of F*, to be the function resulting from the joint combination of every function in F,

$$\bowtie F = f_{T_1} \bowtie \ldots \bowtie f_{T_n}.$$

The *min-marginal of F onto V* is the min-marginal of the combination of all the functions in F, that is $(\bowtie F)[V]$. In contrast, the *one-to-one min-marginal of F onto V* is the set containing the min-marginal of each of its functions f_{T_i} onto $T_i \cap V$, namely $F \downarrow V = \{f_{T_1}[T_1 \cap V], \ldots, f_{T_n}[T_n \cap V]\}$.

Given a function f_V, we say that F is a *V-lower bound of f_V* iff the combination of the one-to-one min-marginal of F onto V is a lower bound of f_V, that is if $\bowtie(F \downarrow V) \leq f_V$.

Furthermore, F is an *(r,V)-lower bound of f_V* iff F is a V-lower bound of f_V and every function in $F \downarrow V$ has arity smaller than or equal to r.

Given F and G (r,V)-lower bounds, *F is at least as good as G ($G \leq F$)* iff $\bowtie(F \downarrow V)$ is at least as good as $\bowtie(G \downarrow V)$.

OBSERVATION 1. *F is a V-lower bound of the min-marginal of F onto V. Formally,*

$$\bowtie(F \downarrow V) \leq (\bowtie F)[V].$$

For any $f_S, f_T \in F$, *the combination of f_S and f_T in F*, is the set of functions that results from removing f_S and f_T from F and adding $f_S \bowtie f_T$, namely

$$F_{f_S \bowtie f_T} = (F \setminus \{f_S, f_T\}) \cup \{f_S \bowtie f_T\}.$$

Two functions f_S and f_T are *(r, V)-combinable* iff the min-marginal onto V of the combination of f_S and f_T can be expressed by a function of arity smaller than or equal to r. Any f_S and f_T such that $|(S \cup T) \cap V| \leq r$ are *(r,V)-combinable*.

The approximation task receives as input a set of functions F^0, a set of variables V, and an arity limit r. Its goal is to find an (r,V)-lower bound for the min-marginal of F onto V. Since for a given approximation task the set of variables V is fixed, in the following we talk of r-combinable, r-lower bound, and min-marginal without explicitly mentioning V.

5. BOTTOM-UP APPROXIMATIONS

Informally, the fundamental idea behind current algorithms for min-marginal approximation is the following. If we combine any pair of functions from a set that is an r-lower bound of the exact min-marginal, the result is another lower bound which is at least as good as the original (and, in fact, most of the times better). Furthermore, if the two functions selected are r-combinable, then the result is also an r-lower bound.

The pseudocode for bottom-up approximation appears in Algorithm 1. Since by Observation 1 we know that F is a lower bound of the min-marginal of F onto V, a bottom-up algorithm starts from the original set of functions F^0. At each iteration, the algorithm: (1) selects a pair of r-combinable functions (f_S, f_T) from the current set of functions; and (2) updates the set of functions to the combination of f_S and f_T in F, that is $F_{f_S \bowtie f_T}$. Since $F_{f_S \bowtie f_T}$ is also an r-arity lower bound and it is at least as good as F, the iterations are likely to improve the lower bound. When

---

Algorithm 1 Bottom-up approximation(F, V, r)

1: $(found, (f_S, f_T)) \leftarrow bestCombinablePair(F, V, r)$
2: **while** $found$ **do**
3: $\quad F \leftarrow F_{f_S \bowtie f_T}$
4: $\quad (found, (f_S, f_T)) \leftarrow bestCombinablePair(F, V, r)$
5: **end while**
6: **return** $F \downarrow V$

---

no more pairs of r-combinable functions are found, the algorithm returns approximation represented by the last F.

5.1 Scope-based partitioning

Scope-based partitioning (SCP) is the most common bottom-up method [5]. Basically, it tries to combine as many functions as possible by choosing the two highest arity functions at each iteration, so long as they are r-combinable.

More in detail, the r-combinable pairs are selected as follows. First, the set of functions F is sorted decreasingly by arity and each function in the list is marked as non-finished. At each iteration, SCP takes the first non-finished element f_{S_1} of F and the element f_{S_i} of F closer to the head such that f_{S_1} and f_{S_i} are r-combinable. It removes them from F and inserts its combination at the head of the list. When there is no function f_{S_i} r-combinable with f_{S_1}, it marks f_{S_1} as finished. The algorithm proceeds until all functions are marked as finished.

Figure 4a depicts how SCP would compute an approximation for the example in Figure 3. Since all functions in F have the same arity (two), they are readily sorted. Hence, SCP would merge the two leftmost ones, and send the third one independently, resulting in the approximation:

$$F' = \{(f_{xt} \bowtie f_{yt})[xy], f_{zt}[z]\} = \left\{ \begin{array}{cc|c} x & y & \\ \hline a & a & 2 \\ a & b & 3 \\ b & a & 3 \\ b & b & 4 \end{array} \quad, \quad \begin{array}{c|c} z & \\ \hline a & 0 \\ b & 0 \end{array} \right\}$$

The main advantage of SCP is the low computational complexity that results from its simplicity. The algorithm performs a nested scanning through the list of functions. During this scanning process, the algorithm computes up to $|F| - 1$ function combinations. To determine the maximum arity of these functions, consider that S is the joint domain of all functions in F. Then, the number of variables that do not appear in V is $|S \setminus V|$. Since the arity limit is r, the maximum arity of each single merged function is $|S \setminus V| + r$. As a result, the complexity of the algorithm is $O(|F|exp(|S \setminus V| + r))$.

5.2 Content-based partitioning

A major advantage of scope-based partitioning is its small computational overhead. Nonetheless, its main drawback is that it does not consider the information within each function. For instance, consider again the previous example. We showed that scope-based partitioning would produce partition F' above. However, notice that there are further approximations that satisfy the r-bound.

Content-based partitioning techniques guide the bottom-up approximation by consulting the functions' contents in addition to their scopes. In general, content-based partitioning tries to assess which pair of r-combinable functions yield the highest improvement.

In [14] Rollon and Dechter present a framework for content-based partitioning that implements the general approach
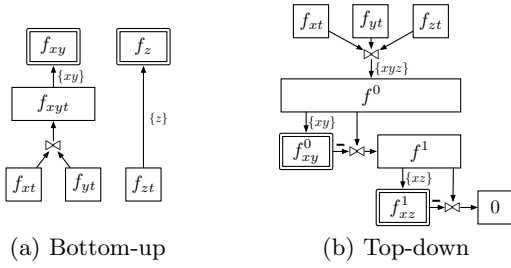
(a) Bottom-up   (b) Top-down

Figure 4: Examples of approximation strategies. Functions in a double-lined box are the ones finally sent.

Algorithm 2 Top-Down Approximation(F, V, r).

1: $f \leftarrow (\bowtie F)[V]$
2: $F' \leftarrow \emptyset$
3: $(found, f_{S'}) \leftarrow \text{selectBestMarginal}(f, r)$
4: **while** $found$ **do**
5:    $F' \leftarrow F' \cup \{f_{S'}\}$
6:    $f \leftarrow f \bowtie (-f_{S'})$
7:    $(found, f_{S'}) \leftarrow \text{selectBestMarginal}(f, r)$
8: **end while**
9: **return** F'

outlined in Algorithm 1. Given an r-lower bound F, content-based partitioning provides the mechanism for selecting the best r-combinable pair of functions $f_S, f_T \in F$ such that the approximation represented by $F_{f_S \bowtie f_T}$ is better than the approximation represented by F. At each iteration, the technique: (1) generates every pair of r-combinable functions $f_S, f_T \in F$; (2) measures the gain obtained by combining f_S and f_T; and (3) selects the pair that maximizes the gain.

Notice that the advantage of combining two functions before sending them is that they will be marginalized together. Hence, the gain can be calculated based on the difference between marginalizing together or marginalizing separately, which can be computed as:

$$f_V = (f_S \bowtie f_T)[V] - (f_S[V] \bowtie f_T[V]).$$

Therefore, the gain function is a metric that takes f_V as its input. Rollon and Dechter present two such functions. Firstly, the local relative error (LRE) metric, which is equivalent to the averaged 1-norm of f_V, assigns as gain the result of adding the costs of all tuples in f_V and dividing by the total number of tuples. Secondly, the local maximum relative error (LMRE) metric, which is equivalent to the ∞-norm of f_V, assigns as gain the maximum cost of any tuple in f_V.

The downside of content-based decomposition is that, in the worst case, the algorithm performs up to $|F| - 1$ selections, computing $|F| - 1$ differences for each selection. Hence, the complexity of the algorithm is $O(|F|^2 exp(|S \setminus V| + r))$.

6. TOP-DOWN APPROXIMATIONS

Bottom-up approximation methods focus on lowering the computational cost. Instead, our purpose is to primarily reduce communication costs. With this aim, we propose a new approach to generate approximations based on: (1) initially computing the function to approximate, and; (2) subsequently decomposing it into lower arity output functions. Figure 4b represents the process of building a top-down approximation of the example in Figure 3. As a first step, f_{xt}, f_{yt} and f_{zt} are combined and then marginalized onto $\{x, y, z\}$ to produce f^0, the function to approximate. After that, the decomposition process starts. Firstly, consider that we select $S' = \{x, y\}$ out of all possible subsets of $\{x, y, z\}$ with arity two. Secondly, f^0 is marginalized onto S' to produce f^0_{xy}, and this marginal is subtracted from f^0 to obtain f^1 (namely $f^1 = f^0 \bowtie (-f^0_{xy})$).[1] Therefore, f^0 is decomposed as $f^0_{xy} \bowtie f^1$, where f^0_{xy} can be regarded as a function ready to communicate and f^1 as the *remainder*

---

[1] Given f, we define $-f$ as f but changing the sign of f costs. It is easy to see that $f \bowtie -f$ is the null function.

after communicating f^0_{xy}. The process continues searching for a decomposition for this remainder.

The main advantage of top-down over bottom-up methods is that they can represent a much wider space of approximations, and hence they should yield more accurate results. Unlike bottom-up methods, which start from an inital set of input functions and proceed by deciding which functions to join, top-down methods start from the function to approximate and proceed by successively selecting the best lower arity min-marginal. While such min-marginal is already part of the decomposition, the process continues by decomposing the result of subtracting the min-marginal from the function to approximate. In general a top-down approximation method is an iterative procedure that at each step i focuses on finding the *most informative min-marginal* for f^i whose arity is smaller than or equal to r. It incorporates the selected min-marginal, $f^{i-1}[S'_i]$, to the list of output functions and updates the function to approximate as follows:

$$f^i = f^{i-1} \bowtie (-f^{i-1}[S'_i]). \tag{1}$$

When the iterative process terminates, the following set of functions stands for the resulting decomposition of f:

$$F = \{f^0[S'_1], f^1[S'_2], \ldots, f^n[S'_{n+1}]\}.$$

More in detail, a general top-down approximation method works as outlined in Algorithm 2. First, it computes the function to approximate (f) by combining the input functions in F and marginalizing onto V. After that, it uses some heuristic to select the best min-marginal $f_{S'}$. Finally, $f_{S'}$ is added to the set of output functions and subtracted from f. This process is repeated until no min-marginal provides additional information. In the remaining of the section we introduce two top-down approximation methods that implement the general method outlined in Algorithm 2.

6.1 Brute force decomposition

In order to determine the most informative min-marginal, a first approach is to consider every possible min-marginal onto r variables from V.[2] Then, we can readily use the gain functions from content-based partitioning to rank the min-marginals and select the most informative one.

This procedure has, however, a high computational cost. At the first iteration, it must compute $\binom{|V|}{r}$ marginals and evaluate them, each requiring $exp(|V|)$ operations. At each following iteration, the number of marginals to compute decreases by one (the selected marginal is never computed again, but all others have to be reevaluated because f^i is

---

[2] Note that discarding functions whose arity is lower than r does not reduce the space of representable functions.

$i = 0$	aa	ab	ba	bb	$|C|$
xy					4
xz					4
yz					4

$i = 1$	aa	ab	ba	bb	$|C|$
xy	X				3
xz		X			3
yz		X			3

$i = 2$	aa	ab	ba	bb	$|C|$
xy	X	X			2
xz	X	X			2
yz	X	X	X	X	0

$i = 3$	aa	ab	ba	bb	$|C|$
xy	X	X	X	X	0
xz	X	X	X	X	0
yz	X	X	X	X	0

(a) Zeroes tracking table and counter vector

$x\,y\,z$	f^0	f^1	f^2	f^3
$a\,a\,a$	4	2	0	0
$a\,a\,b$	2	0	0	0
$a\,b\,a$	4	2	0	0
$a\,b\,b$	3	1	0	0
$b\,a\,a$	5	3	1	0
$b\,a\,b$	3	1	1	0
$b\,b\,a$	5	3	1	0
$b\,b\,b$	4	2	1	0

(b) Per iteration remainders

$$f^0[\emptyset] = 2$$

$$f^1[yz] = \begin{array}{c|c} y\,z & \\ \hline a\,a & 2 \\ a\,b & 0 \\ b\,a & 2 \\ b\,b & 1 \end{array}$$

$$f^2[xz] = \begin{array}{c|c} x\,z & \\ \hline a\,a & 0 \\ a\,b & 0 \\ b\,a & 1 \\ b\,b & 1 \end{array}$$

(c) Selected min-marginals

Figure 5: Zero-tracking decomposition example.

different from f^{i-1}). Hence, its worst time complexity is $O\left(\left(\binom{|V|}{r}\right)^2 \cdot exp(|V|)\right)$.

6.2 Zero-tracking decomposition

The main disadvantage of brute force decomposition is its high computational cost. Here we introduce zero-tracking decomposition, a top-down approximation method that aims at dodging this burden to reduce the computational cost.

Zero-tracking decomposition uses the zero norm of a min-marginal as the heuristic to assess its quality. The zero norm of a function is simply the number of elements in the domain whose image is not zero. Intuitively, if a function is only composed of zeros, it communicates no information whatsoever. The larger the number of non-zero entries in a function, the more informational it will be considered.

The reduction in computational cost comes from realizing that, at each iteration, there is a way to compute the zero norms of each min-marginal from the results of the previous iterations. This avoids the need for recomputing every min-marginal at each iteration as we pursue.

In what follows we detail the operation of the zero-tracking methods when applied to the example in Figure 3. Let $p=\binom{|V|}{r}$ be the number of possible r-arity min-marginals and q the number of tuples for each min-marginal.[3] First, the algorithm allocates a boolean table $Zeroes$ of p rows and q columns. Each entry $[U, t_U]$ in $Zeroes$ encodes whether the value for t_U of the min-marginal of f onto U is zero or not. That is, $Zeroes[U, t_U]$ is true whenever $f[U](t_U)$ is zero. Hence, all entries are initialized to $false$. Additionally, it allocates a vector C of p integers to count the number of non-zero tuples for each min-marginal. Since C counts non-zero elements, it is initialized to the number of tuples in each min-marginal (q). At the top of Figure 5a we show (for iteration $i=0$) table $Zeroes$ and vector C after initialization. Next, the exact min-marginal $(\bowtie F)[V]$ is calculated by combining all the initial functions and marginalizing the result onto $\{x, y, z\}$. The result is shown in Figure 5b as f^0.

Notice that, at this point, f^0 does not contain any zero. In

[3]For simplicity of exposition we assume that all variables are defined over the same domain.

Algorithm 3 ZeroDecomposition(F, V, r).

1: initialize($Zeroes$, C)
2: $f \leftarrow (\bowtie F)[V]$
3: $F' \leftarrow \emptyset$
4: $(f_{S'}, gain) \leftarrow (f[\emptyset], 1)$
5: **while** $gain > 0$ **do**
6: $\quad F' \leftarrow F' \cup \{f_{S'}\}$
7: $\quad f \leftarrow f \bowtie (-f_{S'})$
8: $\quad (f_{S'}, gain) \leftarrow$ selectBestMarginal(f,$Zeroes$,C)
9: **end while**
10: **return** F'
11:
12: **function** selectBestMarginal(f,$Zeroes$,C)
13: $\quad$ **for all** new t_S s.t. $f(t_S) = 0$ **do** // new zeroes in f
14: $\qquad$ **for all** $U \in T$ **do** // subsets of r variables
15: $\qquad\quad$ **if not** $Zeroes(U, t_S[U])$ **then**
16: $\qquad\qquad C(U) \leftarrow C(U) - 1$
17: $\qquad\qquad Zeroes(U, t_S[U]) \leftarrow true$
18: $\qquad\quad$ **end if**
19: $\qquad$ **end for**
20: $\quad$ **end for**
21: $\quad S' \leftarrow \arg\max_{U \in C} C(U)$
22: $\quad$ **return** $f[S']$, $C(S')$
23: **end function**

---

order to introduce some zeroes, we subtract from f^0 its minimum (which amounts to the min-marginal of f^0 onto the empty set). In the example, this subtraction yields function $f^0[\emptyset]$, shown at the top of Figure 5c. Subsequently, it calculates the next remainder f^1 using Equation 1.

After calculating the new remainder f^1, the new function to approximate, the algorithm proceeds to update the $Zeroes$ table along with the C counter. Back to our example, notice that f^1 contains a single tuple with zero cost $t_S = (xyz \leftarrow aab)$. Then, the algorithm calculates the projection of t_S to each row U and sets cell $[U, t_S[U]]$ to $true$ in the $Zeroes$ table. In the example, the cell for row xy and column aa is set to true in the $Zeroes$ table. Moreover, the counter for row xy decreases to record that there is one less non-zero cost tuple. Figure 5a ($i=1$) shows the state of both the $Zeroes$ table and the counter vector after iteration $i=1$. In general, for each new zero cost tuple t_S, the algorithm checks the $Zeroes$ table cell at row U and column $t_S[U]$. If the cell is $false$, it is set to $true$ to indicate that the cost of the min-marginals for the tuple will be zero from iteration i onwards. Moreover, the value of the counter of non-zero cost tuples for the min-marginal, $C(U)$, decreases by one.

Once the $Zeroes$ table and counters are updated, there are two cases: (1) If all counters' values are zero, it means that the cost for all tuples of all subsequent min-marginals will be zero. Therefore, since it is not possible to extract more information from subsequent min-marginals, the algorithm terminates and returns the list of selected min-marginals so far, $\{f^0[S'_1], f^1[S'_2], \ldots, f^m[S'_{m+1}]\}$, as the resulting decomposition. (2) Otherwise, the min-marginal with more non-zero tuples is selected as the best min-marginal, and the algorithm continues.

In the example in Figure 5, all candidate min-marginals (see the rows in table $Zeroes$ at iteration $i=1$) contain 3 non-zero tuples. Thus, at the next iteration ($i=2$), the algorithm can randomly choose the marginalization of f^1 onto any pair of variables. Say that the algorithm chooses $\{yz\}$. There-

fore, the selected best min-marginal is $f^1[yz]$, and hence the new remainder f^2 can be computed. After updating the *Zeroes* table, there are still two counters larger than zero, as shown in Figure 5a ($i=2$). In our case, the algorithm selects $f^2[xz]$ (discarding $f^2[xy]$), calculates the new remainder f^3, and updates the *Zeroes* table to yield the table in Figure 5a ($i=3$). At this point, since all counters are zero, the algorithm terminates to return the following set of selected best min-marginals as the resulting decomposition:

$$F' = \{f^0[\emptyset], f^1[yz], f^2[xz]\}.$$

On the one hand, notice that the whole procedure –shown in Algorithm 3– never calculates a min-marginal unless it is going to be returned as part of the resulting decomposition. Further, since the maximum number of functions in a decomposition is $\binom{|V|}{r}$, the worst case complexity of calculating the decomposition is $O(\binom{|V|}{r}exp(|V|))$. On the other hand, the algorithm has to maintain the zeroes table, which also has a cost. Note that function selectBestMarginal only processes the tuples that are zero in the current iteration and were not zero in the previous iteration[4]. This means that to maintain the table, each tuple will be processed at most once. Since for each tuple we mark each possible min-marginal, the time complexity of maintaining the table is $O(\binom{|V|}{r}exp(|V|))$, not increasing the overall time complexity.

7. EMPIRICAL EVALUATION

In this section we evaluate the performance of the different function approximation approaches on DIMCTEf. For each experiment, we present both the communication savings and increase in overall computational cost with respect to GDL. We choose to track these measures because they are the key ones in constrained environments. For instance, consider a wireless sensor networks setting. Since running out of battery disables a node, battery consumption is probably the most important figure to consider. Therefore, both communication and computation costs are important because they directly determine battery consumption. We estimate the overall computational cost by adding the processing times incurred by each node, while ignoring communication times. Similarly, the overall communication cost can be easily determined by adding the number of bytes of all sent messages.

Because GDL's communication and computation is mainly determined by the maximum clique size of the computed Junction Tree, experiments are segmented by this parameter. Consequently, both GDL and all DIMCTEf approaches use the very same Distributed Junction Tree Generator [15] algorithm to compute JTs. Additionally, notice that the parallelism degree is roughly the same for all algorithms, because it mainly depends on the computed JT. As a consequence, since DIMCTEf always communicates less information than GDL, the relative increase in real solving time between GDL and DIMCTEf would be lower than the relative increase in overall computation shown in this paper.

Since DIMCTEf removes tuples, it generates sparse functions. Sending sparse functions can lead to communication savings, but only if the implementation uses a special codification to transmit these functions. However, exploring the codification of sparse functions was not one of the objectives of this paper. Hence, we simply set a special value as

---

[4]New zeros can be detected at no cost while computing the combination in line 7.

cost for the filtered tuples, and compressed the messages. Specifically, we chose an Arithmetic Encoder [4] with a Partial Prediction Matching model of 8 bytes. This compression method is known to achieve good compression ratios, so long as its input contains repeated values. The downside is that compressing has a high computational cost, which is considered as part of our overall cost. Regarding GDL, compression hurts because the overall computation increases by an order of magnitude, while communication savings are practically negligible. Therefore, we report GDL results without compressing (following the idea of presenting results for each algorithm in its best possible condition). Likewise, although we tried both the LRE and LMRE metrics for both content-based partitioning and brute-force decomposition, we only report the best results obtained.

We conducted tests with the sensor networks instances from [10], but they were very easy for GDL (5 maximum clique variables). Thus, all approaches lead to the same results, requiring 3 times less communication while maintaining the same computation cost as GDL.

Next, we designed an experiment to measure the methods' trend as the variables' arity increases. Thus, it is composed of 35 problems of 20 variables for each domain size, with a random structure of densities ranging from $p=0.1$ to $p=0.3$, where p is the probability of appearance for all edges. Function costs are taken from a normal distribution $\mathcal{N}(0,1)$, and then made positive by adding its minimum value to each relation. Results in Figure 6a show that top-down approximation methods perform significantly better than bottom-up approximations in the communication front, with nearly constant savings between two and three times better. As expected, the brute force approach is way more expensive computationally than other methods (up to 100 times slower than GDL in the worst case). Nevertheless, zero-tracking's overall computation cost is just slightly higher (13 times that of GDL at most) than that of the content-based approach (10.5 times), whereas their savings are much larger (110.5 times less sent bytes for zero-tracking against 38.3 times for content-based). Moreover, its savings in communication increase almost 10 times faster than the computational cost.

Then, we conducted a second experiment that measures the trends when the problems' maximum clique variables increases. Consequently, it contains problems of 20 variables of arity 5 for each maximum clique variables, also with random structures between $p=0.1$ and $p=0.3$, and normal costs. Figure 6b shows that the communication savings increase exponentially for all methods, yet zero-tracking grows at a much faster rate than the others while keeping the computational cost under control.

Finally, the third experiment measures the impact of structure in the problems' constraint graph. Thus, it contains lattice-structured problems of 25 and 36 variables, leading to JTs of 8 and 10 maximum clique variables. Once again, top-down approximation methods achieve the largest communication savings. In particular, zero-tracking decomposition requires up to 612 times less bytes than GDL in 25% of the clique size 8 problems, while being only 44 times slower.

In summary, top-down approximations result in large communication savings. Additionally, zero-based decomposition remains competitive in computational effort with respect to state-of-the-art approximation methods. Hence, we see zero-based decomposition as the method of choice to optimally solve DCOPs in communication-constrained scenarios.
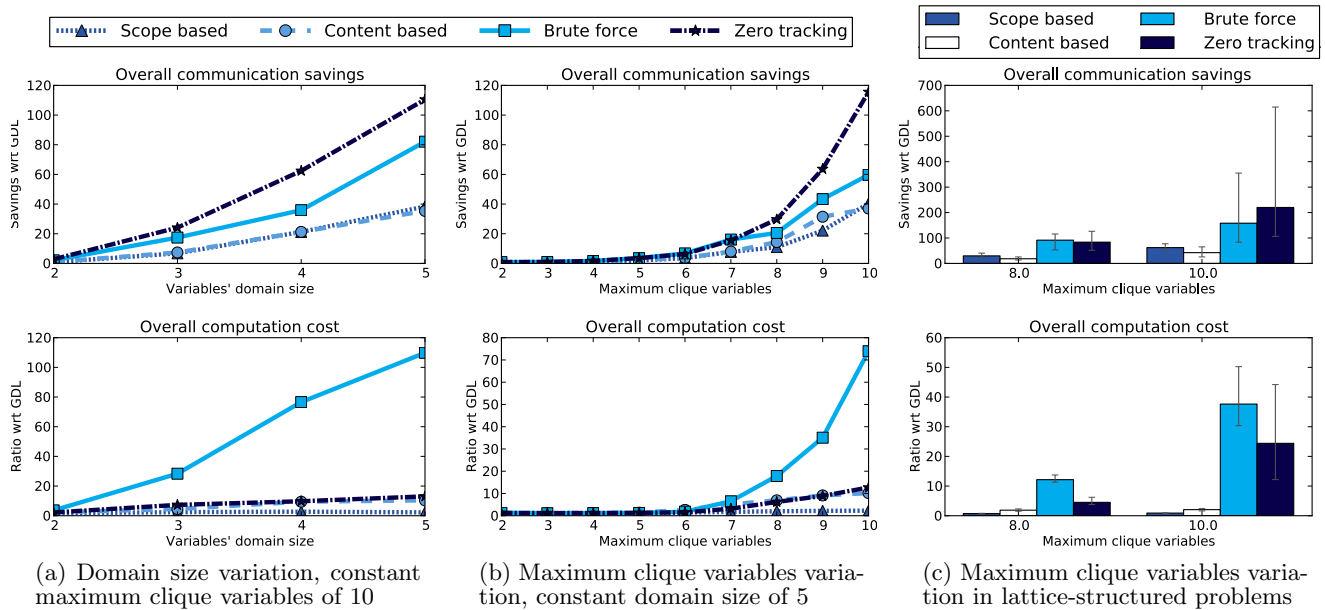
Figure 6: Performance evaluation results.

8. CONCLUSIONS

We addressed the issue of optimal DCOP solving in communication-constrained scenarios, using GDL with function filtering. We first reviewed current state-of-the-art approaches to message approximation, presenting them in a common framework that we named bottom-up approximations. Next, we proposed top-down approximations, a new class of methods designed to reduce communication costs. We then presented two realizations of this novel approach: (1) brute-force decomposition, a naive implementation with high computational cost; and (2) zero-tracking decomposition, which greatly reduces the amount of computation. Finally, we empirically evaluated their performance, showing that top-down approximations always achieve larger communication savings than bottom-up ones. In fact, zero-tracking decomposition does so while keeping the computational cost at bay, becoming the method of choice for optimally solving DCOPs in communication-constrained scenarios.

9. ACKNOWLEDGMENTS

10. REFERENCES

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

[2] I. Brito and P. Meseguer. Distributed cluster tree elimination. In *IJCAI DCR Workshop*, pages 16–30, 2009.

[3] I. Brito and P. Meseguer. Improving dpop with function filtering. In *AAMAS*, pages 141–148, 2010.

[4] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *Communications, IEEE Transactions on*, 32(4):396 – 402, apr. 1984.

[5] R. Dechter and I. Rish. A scheme for approximating probabilistic inference. *Proceedings of Uncertainty in Artificial Intelligence (UAI'97)*, pages 132–141, 1997.

[6] B. Faltings, D. Parkes, A. Petcu, and J. Shneidman. Optimizing streaming applications with selfinterested users using M-DPOP. In *COMSOC'06*, pages 206–219, 2006.

[7] A. Farinelli, A. Rogers, and N. Jennings. Bounded approximate decentralised coordination using the max-sum algorithm. In *IJCAI DCR Workshop*, pages 46–59, 2009.

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

[9] C. Kiekintveld, Z. Yin, A. Kumar, and M. Tambe. Asynchronous algorithms for approximate distributed constraint optimization with quality bounds. In *AAMAS*, pages 133–140, 2010.

[10] R. T. Maheswaran, M. Tambe, E. Bowring, J. P. Pearce, and P. Varakantham. Taking dcop to the real world: Efficient complete solutions for distributed multi-event scheduling. In *AAMAS*, pages 310–317, 2004.

[11] 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.

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

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

[14] E. Rollon and R. Dechter. New mini-bucket partitioning heuristics for bounding the probability of evidence. In *AAAI*, pages 1199–1204, 2010.

[15] M. Vinyals, J. A. Rodríguez-Aguilar, and J. Cerquides. Constructing a unifying theory of dynamic programming dcop algorithms via the generalized distributive law. *JAAMAS*, pages 1–26, 2010.

[16] M. Vinyals, J. A. Rodríguez-Aguilar, and J. Cerquides. Egalitarian utilities divide-and-coordinate: Stop arguing about decisions, let's share rewards! In *ECAI*, pages 1025–1026, 2010.

[17] W. Zhang, G. Wang, Z. Xing, and L. Wittenburg. Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks. *Artificial Intelligence*, 161(1-2):55–87, 2005.