Improving function filtering for computationally demanding DCOPs

Marc Pujol-Gonzalez, Jesus Cerquides,
Pedro Meseguer, and Juan Antonio Rodriguez-Aguilar

Artificial Intelligence Research Institute (IIIA-CSIC),
Universitat Autònoma de Barcelona,
E-98324 Catalonia (Spain)
{mpujol,cerquide,pedro,jar}@iiia.csic.es
http://www.iiia.csic.es/

Abstract. In this paper we focus on solving DCOPs in computationally demanding scenarios. GDL optimally solves DCOPs, but requires exponentially large cost functions, being impractical in such settings. Function filtering is a technique that reduces the size of cost functions. We improve the effectiveness of function filtering to reduce the amount of resources required to optimally solve DCOPs. As a result, we enlarge the range of problems solvable by algorithms employing function 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 [7], DPOP [8], and its generalization GDL [1, 13]. Since DCOPs are NP-Hard [7], 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 resource constrained. For instance, in wireless sensor networks, memory, bandwith, and computation are severely limited [15]. As computational requirements grow, so does the relevance of resource constraints. Hence, in this paper we focus on computationally demanding scenarios. An approach in these domains is to drop optimality in favor of lower complexity approximate algorithms with weaker guarantees [10]. As an alternative, we aim at reducing resource usage while keeping optimality.

Function filtering [12] is a technique that reduces the size of cost functions by filtering out those tuples that are found unfeasible to be extended into an optimal solution. Function filtering can be applied to GDL, as detailed in [3], where they present the so-called DIMCTEf algorithm. Provided a lower bound on the cost of the best extension of the tuple, and an upper bound on the cost of the optimal solution, a tuple is filtered out when its lower bound is greater than the upper bound. Thus, the amount of filtered tuples strongly depends on

the quality of both lower and upper bounds. The tighter the bounds, the larger the size reduction of cost functions.

In [9], the authors reduce DIMCTEf communication needs whilst keeping memory and computation needs. However, this alternative turns out to be infeasible in computationally demanding scenarios. The purpose of this paper is to improve the effectiveness of function filtering by tightening the lower and upper bounds in [3]. More effective filtering reduces the amount of resources required to optimally solve DCOPs. Empirically, we estimate a reduction of up to 72% on communication costs and up to 32% on computational costs. Furthermore, we also obtain a significant memory reduction, allowing agents to solve up to 75% more problem instances given the same constraints. To summarize, we manage to increase the range of problems that can be solved optimally by algorithms employing function filtering.

The structure of the paper is as follows. Section 2 introduces DCOPs, and Section 3 outlines GDL with function filtering. Section 4 focuses on improving lower bounds, whereas Section 5 focuses on improving upper bounds. Finally, Section 6 draws some conclusions.

2 DCOP

A Distributed Constraint Optimization Problems (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 collection of domains, where $D(x_i)$ is the finite set of x_i's possible values.
- C is a set of cost functions. Each $f_S \in C$ is defined on the ordered set $S \subseteq X$, and specifies the cost of every combination of values of variables in S, namely $f_S : \prod_{x_j \in S} D(x_j) \mapsto \mathbb{R}^+$. S is f_S's *scope*, and $|S|$ is its *arity*.
- A is a set of p agents.
- $\alpha : X \to A$ 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 concepts and operations that will be used throughout the paper.

A *tuple* t_S, with scope S, is an ordered set of values assigned to each variable in $S \subseteq X$. 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 t_X. If a complete tuple's cost is lower than the user-specified threshold, it is a *solution*. A minimum cost solution is *optimal*.

The *projection* $t_S[T]$ of a tuple t_S to $T \subseteq S$ is a new tuple t_T, which only includes the values assigned to the variables in T. 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.

Let $F = \{f_{T_1}, \ldots, f_{T_m}\}$ be a set of functions, *the combination of* F, $\bowtie F$, is the function resulting from the joint combination of every function in F,

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

Therefore, solving a DCOP means finding the tuple t_X that minimizes $(\bowtie C)(t_X)$.

Lower bounds to the cost of extensions of tuples

Let $V \subseteq X$ be a subset of the variables of the problem and t_V a tuple that assigns values to each of the variables in V. An extension of t_V to X is a tuple that keeps the assignments of t_V and assigns new values to the variables in $X \setminus V$. If the cost of each possible extension of t_V is larger than or equal to LB, we say that LB is a lower bound of the cost of the best extension of tuple t_V.

Likewise, a function f_T is a *lower bound* of function f_S, noted $f_T \leq f_S$, iff $T \subseteq S$, and

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

A function g_V is a lower bound of a set of functions F if it is a lower bound of its combination $\bowtie F$. In particular, a function g_V is a lower bound of a problem if it is a lower bound of the combination of its set of cost functions C. Namely, if for each tuple t_V, $g_V(t_V)$ is a lower bound of the cost of the best extension of t_V to the complete problem.

The tightest lower bound is provided by the min-marginal. The *min-marginal* $f_S[T]$ of a cost function f_S over $T \subset S$ is a new cost function f_T, which assigns to each tuple t_T the minimum cost among all the extensions of t_T to S. Formally,

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

Similarly, the *min-marginal of* F *over* V is the min-marginal of the combination of all the functions in F, that is $(\bowtie F)[V]$.

Given a set of functions F, the time to compute the min-marginal of F over V is bounded by $\mathcal{O}(d_T^{|T|})$, where $T = \bigcup_{i=1}^{m} T_i$, and d_T is the maximum domain among the variables in T. In some scenarios, this can be overdemanding. For that reason we introduce a less costly way of computing a lower bound of a set of functions. Specifically, we define $\overset{\text{V}}{\bowtie} F$, *the combination of* F *under* V as the result of combining the min-marginals of each of its functions over V. That is,

$$\overset{\text{V}}{\bowtie} F = f_{T_1}[T_1 \cap V] \bowtie \ldots \bowtie f_{T_m}[T_m \cap V].$$

$\overset{\text{V}}{\bowtie} F$ is a lower bound of F and can be assessed in $\mathcal{O}(\max(\max_{i=1}^{m} d_{T_i}^{|T_i|}, d_V^{|V|}))$ time, which can be way smaller than $\mathcal{O}(d_T^{|T|})$.

3 GDL

Several algorithms can optimally solve DCOPs. In particular, we consider the GDL algorithm [1], following the Action-GDL description [13]. GDL works over a special structure named junction tree (JT), also known as joint tree or cluster tree. Action-GDL runs two phases: (1) costs are sent from the leaves up to the root; (2) optimal assignments are decided and communicated down the tree.

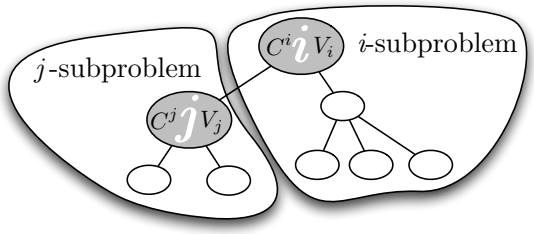


Fig. 1: Subproblems in a junction tree.

In Action-GDL, each node in the JT is controlled by an agent and has a set of variables V_i and a subset of the cost functions of the problem C^i (representing its own stake at the problem). We define the neighbors of i, $neigh(i)$, as the set of nodes linked to i in the JT. Figure 1 shows agent i and j linked by an edge in a JT. Observe that removing the edge connecting i and j splits the JT into two different connected components, which we call subproblems. Formally, we say that the i-subproblem involves every cost function in the component containing i after the edge is removed. Subproblems i and j are coupled by a set of variables they share and must agree upon, namely their *separator* $S_{ij} = V_i \cap V_j$.

Action-GDL determines the optimal solution using the following scheme. First, each leaf node starts by sending a lower bound of the costs of its subproblem to its parent. When a node has received the messages from all its children, it combines them to produce the lower bound of its own subproblem, and sends it to its parent. Once the root has received messages from all its children, it decides the best assignment (minimum cost tuple) for its variables. Then, it broadcasts this assignment to its children, who assess their best assignments and send them down the tree.

A main drawback of Action-GDL is the exponential size of the cost functions exchanged. To reduce communication, these messages can be approximated by means of lower arity functions [5], at the expense of losing optimality. The quality of the solution is expected to grow as the arity increases. We can take advantage of that to build an algorithm that successively finds better solutions by increasing the arity of approximations. As introduced in [9] there are two different schemes to approximate the functions exchanged in a junction tree: top-down and bottom-up. Top-down schemes compute the whole function to approximate,

and subsequently split it into smaller arity functions. Conversely, bottom-up schemes avoid computing the function to approximate, directly assessing smaller arity functions by combining subsets of functions. Since our interest lies in computationally intensive scenarios, the cost of assessing the function to approximate can be prohibitive, and hence in the following we restrict to bottom-up schemes.

3.1 GDL with function filtering

In this section we detail how function filtering can be applied to cope with the exponential growth of cost functions in GDL. Function filtering [12] is a technique that reduces the size of cost functions by filtering out those tuples that are found unfeasible to be extended into an optimal solution. In order to do that, each agent i intending to send a cost function f_U to agent j needs: (1) a lower bound $lb_U(t_U)$ on the value of each tuple t_U; and (2) an upper bound UB on the value of the optimal solution. Provided that, we say that the agent filters f_U with lb_U and UB when it removes those tuples t_U such that $lb_U(t_U) > UB$ (*i.e.* the ones that cannot be extended into an optimal solution), and sends the remaining ones. Obviously, the amount of filtered tuples will strongly depend on the quality of bounds lb_U and UB. The tighter the bounds, the larger the size reduction of cost functions.

Agents are unable to build a reasonably tight lower bound with the information they have in Action-GDL. Nevertheless, computing the lower bound of the best-cost extension of a tuple has been studied for the centralized case in MCTE(r) [4], where r is the highest arity of functions that can be computed. Its extension to the distributed setting is DMCTE(r) [3], where functions of arity greater than r cannot be exchanged. DMCTE(r) works on a JT in three phases. The first one, *cost propagation*, builds a summary of each subproblem of the JT. The second one, *solution propagation*, computes a candidate solution. The third one, *bound propagation*, assesses the cost of the candidate solution.

During the cost propagation phase of DMCTE(r) agents exchange approximate cost functions (one or several functions of arity up to r), first bottom-up and then top-down the JT. After this, each agent i has: (1) a set of functions C^i, containing its stake at the problem; and (2) for each neighbor j a set of functions $G^{j\to i}$. $G^{j\to i}$ stands for a summary of the j-subproblem, namely a *lower bound* on the cost of each tuple in that subproblem. Observe that agent i can assess the cost of an assignment by adding its own costs and the costs of its neighbors' subproblems. Likewise, the agent can assess a lower bound for the costs of a tuple in the complete problem by combining its own cost functions with those received from its neighbors. Formally,

$$lb_{V_i}(t_{V_i}) = (\bowtie F)(t_{V_i}) \tag{1}$$

where $F = C^i \cup \bigcup_{j\in neigh(i)} G^{j\to i}$. However, the lower bound assessed in Equation (1) requires $\mathcal{O}(d_{V_i}^{|V_i|})$ time, where V_i are the agent's variables. Hence, it cannot be computed in computationally demanding scenarios.

As an alternative, [3] proposes that each agent i willing to send function g_U, where $U \subsetneq V_i$, assesses a lower bound $lb_U(t_U)$ by adding: (1) a lower bound on the cost of the best extension of tuple t_U in the j-subproblem to (2) the cost that g_U assigns to t_U. Formally,

$$lb_U(t_U) = (g_U^{j \to i} \bowtie g_U)(t_U), \tag{2}$$

where $g_U^{j \to i} = \overset{\text{U}}{\bowtie} G^{j \to i}$. Henceforth we shall refer to this lower bound as *one-sided lower bound*. Since, in this case, the complexity of combining under U is $\mathcal{O}(d_U^{|U|})$, this turns out to be much cheaper than using Equation (1).

The solution propagation phase of DMCTE(r) works as Action-GDL's. However, the solution obtained after this phase is not necessarily optimal. This is because DMCTE(r) operates with approximate cost functions. Thereafter, DMCTE(r) introduces a third phase to assess the cost of the candidate solution. Such cost is aggregated from the leaves to the root and then communicated down the tree. Observe that the cost of the candidate solution is an upper bound on the cost of the optimal solution.

Computing lower and upper bounds allows us to apply function filtering. Note that, when DMCTE(r) runs with increasing r, the cost of the best solution found so far constitutes an upper bound UB. Furthermore, at each iteration, each agent i willing to send a set of cost functions to agent j can filter each one of them separately. Thus, for each function: first, it uses Equation (2) to assess a lower bound from the last message received from j; and then, it filters the function with the lower bound and UB. The resulting algorithm is known as DIMCTEf [3].

4 Two-sided filtering

Next, we aim at tightening the one-sided lower bound described above. Consider that agent i has already received $G^{j \to i}$ from agent j. After that, it intends to send a set of functions $G^{i \to j}$, summarising the cost information in the i-subproblem, to agent j. Since no cost function appears in both the i-subproblem and the j-subproblem, we can assess a lower bound for the complete problem by adding a lower bound of each of them. Notice that the one-sided lower bound in Equation (2) already assesses the summary of the costs of the j-subproblem from $G^{j \to i}$. Likewise, we can assess the summary of the costs of the i-subproblem from $G^{i \to j}$. Therefore, we can employ the cost summaries of both subproblems to obtain a tighter bound.

Formally, when sending cost function $g_U \in G^{i \to j}$, we compute the lower bound of tuple t_U as:

$$lb_U(t_U) = (g_U^{j \to i} \bowtie g_U^{i \to j})(t_U) \tag{3}$$

where

- $g_U^{i \to j} = \overset{\text{U}}{\bowtie} G^{i \to j}$ is a lower bound on the contribution of the i-subproblem.

– $g_U^{j\to i} = \overset{U}{\bowtie} G^{j\to i}$ is a lower bound on the contribution of the j-subproblem.

Observe that there is no double counting of costs because no cost function appears in both the i-subproblem and the j-subproblem. Henceforth, we will refer to the lower bound in Equation (3) as *two-sided lower bound*. The name stems from the symmetrical use of both subproblems. Hereafter, two-sided filtering refers to filtering employing the two-sided lower bound.

Given

$x\,y$	$g_{xy}^{j\to i}$
$a\,a$	3
$a\,b$	4
$b\,a$	3
$b\,b$	3

,

$x\,y$	f_{xy}
$a\,a$	5
$a\,b$	2
$b\,a$	8
$b\,b$	6

,

$x\,z$	f_{xz}
$a\,a$	4
$a\,b$	3
$b\,a$	5
$b\,b$	2

, and UB=10

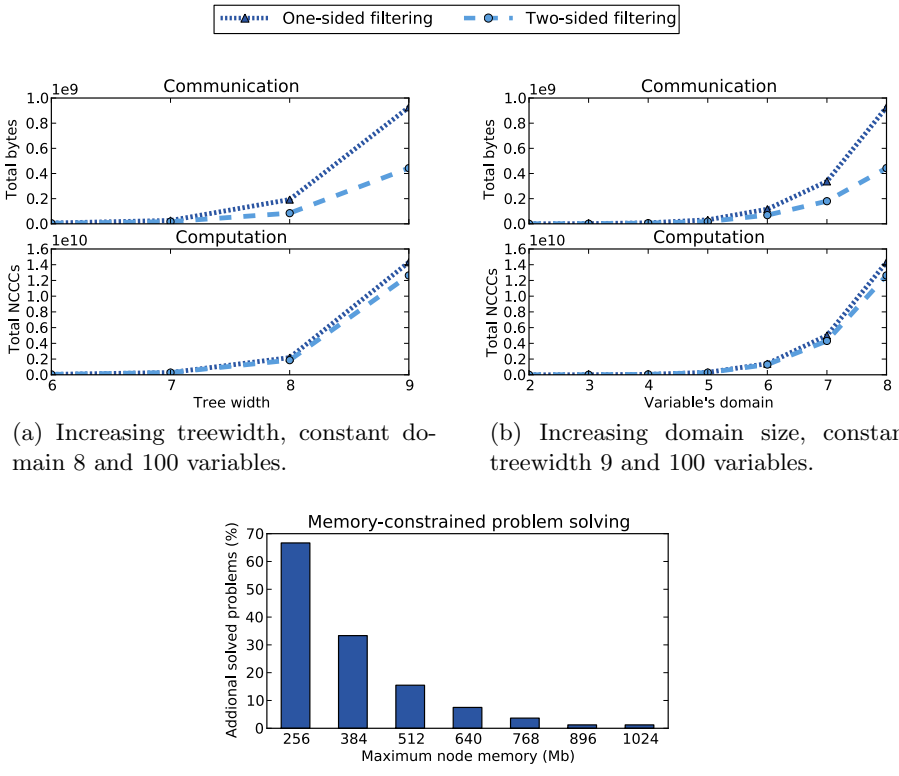
$x\,y$	$f_{xy} \bowtie f_{xz}[x] = g_{xy}^{i\to j}$			**One-sided** $f_{xy} \bowtie g_{xy}^{j\to i}$	**Two-sided** $g_{xy}^{i\to j} \bowtie g_{xy}^{j\to i}$
$a\,a$	5	3	8	$5 + 3$	$8 + 3$ ✗
$a\,b$	2	3	5	$2 + 4$	$5 + 4$
$b\,a$	8	2	10	$8 + 3$ ✗	$10 + 3$ ✗
$b\,b$	6	2	8	$6 + 3$	$8 + 3$ ✗

Fig. 2: Example of one-sided vs. two-sided filtering. Tuples ticked off (✗) are the ones being filtered out.

For instance, consider that agent i has received a set of functions $G^{j\to i}$, which combined under $\{x, y\}$ produces the function $g_{xy}^{j\to i}$ shown in Figure 2. Furthermore, agent i knows that the cost of the optimal solution is smaller than or equal to 10 ($UB = 10$). Now, it wants to send functions $G^{i\to j} = \{f_{xy}, f_{xz}\}$ (in Figure 2) to agent j. Consider that it starts by sending function f_{xy}. Agent i can calculate the one-sided lower bound using Equation (2), filtering out tuple (x=b, y=a) as shown in Figure 2. Alternatively, the agent can compute the two-sided lower bound using Equation (3), by assessing the lower bound on the contribution of its own subproblem, namely $g_{xy}^{i\to j} = \overset{xy}{\bowtie} G^{i\to j} = f_{xy} \bowtie f_{xz}[x]$. Figure 2 shows that two-sided filtering performs better, keeping only the tuple (x=a, y=b) as feasible.

4.1 Empirical evaluation

In this section we empirically compare the performance of DIMCTEf when using one-sided filtering and two-sided filtering. For each experiment, we track the amount of communication used by the algorithm (i.e., the total number of bytes) along with the total amount of serial computation (i.e., the number of non-concurrent constraint checks). Moreover, we performed signed rank tests [14] on all results to ensure that differences between methods are statistically significant ($\alpha = 0.01$).



(a) Increasing treewidth, constant domain 8 and 100 variables.

(b) Increasing domain size, constant treewidth 9 and 100 variables.

(c) Additional problems solved by two-sided filtering when agents have limited memory.

Fig. 3: Experimental results of one-sided filtering against two-sided filtering.

It is well-known that a JT's treewidth is the most important indicator of problem hardness for GDL-based algorithms. Hence, we segmented our experiments according to this parameter, and ensured that all algorithms use the very same JT when solving the same problem instance. As mentioned before, we are specifically interested in hard problems. Thus, our first experiment used the hardest dataset in the DCOP repository, namely the meeting scheduling dataset [6]. We obtained very similar results for both one-sided and two-sided filtering, with very small gains for two-sided filtering.

As a consequence, we decided to design new datasets harder than those typically used in the DCOP literature. We characterized each scenario by three parameters: number of variables, variables' domain size, and treewidth. For each scenario, we generated 100 problems by: (1) randomly drawing problem structures following an Erdös-Rényi $G(n, p)$ model [2]; (2) selecting those structures having the treewidth requested for the scenario; and (3) randomly drawing costs from a $\mathcal{N}(0, 1)$ distribution.

First, we ran an experiment to evaluate the savings as the treewidth increases. We generated scenarios with 100 variables of domain 8, and treewidths ranging from 6 to 9. Figure 3a shows that two-sided filtering reduces, with respect to one-sided filtering, the amount of communication required by a median of 26% for the easier problems (treewidth 6). It achieves even better results for the harder problems (52% for the set with treewidth 9).

Next, we designed an experiment to measure the trend of both filtering styles as the variables' domain sizes increase. Thus, we generated scenarios with 100 variables, treewidth 9 and domain sizes ranging from 2 to 8. Once again, two-sided filtering achieves significant communication savings for all the experiment's problems. Further, as the domain increases, so do the savings with respect to one-sided filtering: starting with a narrow 8% reduction for the binary variables set, and reaching a 52% reduction for the toughest scenario (domain size 8).

Furthermore, note that in all but the easiest experiments (the ones with variables' domains 2 to 4), two-sided filtering performs up to 15% less non-concurrent constraint checks. Because function size is the main limiting factor of GDL-based algorithms, this suggests that two-sided filtering can solve problems that are too hard for one-sided filtering. Therefore, we re-ran the hardest set of problems (domain size 8, treewidth 9), but now limiting the maximum amount of memory available for each agent. Figure 3c shows that, indeed, two-sided filtering solves as much as 67% more problems than one-sided filtering.

5 Improving upper bounds

As mentioned above, the tighter the bounds, the larger the size reduction of cost functions. Previous work has focused on trying to improve the lower bounds sent during the cost propagation phase [11, 9]. In contrast, there is no work addressing the improvement of upper bounds (UB), despite also playing an important role in function filtering. Recall that the cost of any candidate solution is an upper bound on the cost of the optimal solution. Therefore, exploring multiple candidate solutions at the same time, instead of a single one, is expected to lead to better bounds. In this section, we present different approaches to propagate multiple candidate solutions. Then, we experimentally show that the benefits of providing the filtering process with better upper bounds can outweigh the cost of calculating them.

5.1 Centralized exploration

The simplest approach to propagating multiple assignments is to perform the very same procedure as DMCTE(r) does, but with multiple assignments instead. This is, the root node begins by choosing the best m assignments for its variables, and subsequently sends them to its children. Thereafter, each child extends each assignment by choosing its best cost extension (according to its knowledge), and relays them to its own children. The solution propagation phase terminates once each leaf node has received (and extended) its assignments.

Then, agents need to calculate the cost of each solution. With this aim, there is a third phase where: (1) the cost of each assignment is aggregated up the tree; and (2) the best assignment and its cost (the new global UB) are sent down the tree. Firstly, each leaf node i evaluates the cost of each assignment in its problem's stake C^i, and sends the resulting costs to its parent. Subsequently, once a parent node j receives the costs of each assignment from its children, it aggregates them with the costs in its own problem stake C^j. Thereafter, it sends the resulting costs up the tree. After the root has received and aggregated the costs from all its children, it decides the best assignment. Finally, the root sends the best assignment along with its cost down the tree.

The main advantage of this method lays in its simplicity. However, its main drawback is that it offers limited exploration capabilities because: (1) it cannot propagate more than k candidate solutions, where k stands for all possible assignments for the root's variables; and (2) when a node finds several good extensions for a candidate solution, it is enforced to choose only one of them. For instance, say that an agent receives assignment $(x{=}a)$ from its parent, and has to choose a value for variable y. According to its knowledge, extension $(x{=}a, y{=}a)$ costs 1, and so does extension $(x{=}a, y{=}b)$. Because centralized exploration forces the agent to extend each received assignment exactly once, extension $(x{=}a, y{=}b)$ must be discarded. This restriction implies that the root is the only node able to explore new candidate solutions, whereas other nodes simply exploit them.

5.2 Distributed exploration

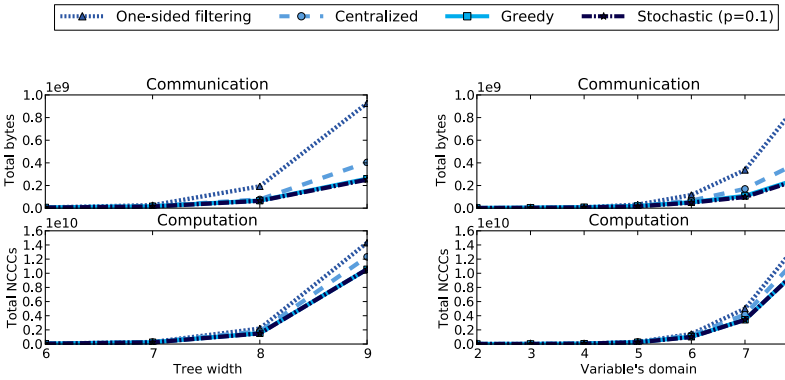
To overcome the limited exploration capabilities of centralized exploration, we need mechanisms allowing any node to explore new assignments.

First, we assume that the number of assignments considered by each agent is bounded by m. Then, we enforce nodes to extend each received assignment at least once. However, we allow each agent to extend any assignment multiple times. Provided a node receives a set of assignments A, it needs to decide the number of new assignments to explore n_e, which cannot exceed $n_{max} = m - |A|$. With this aim, we propose that an agent employs one out of the following strategies:

Greedy. An agent extends as many assignments as possible, namely $n_e = n_{max}$.

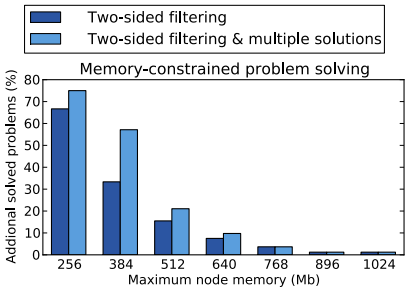
Stochastic. An agent chooses the number of assignments to extend n_e from a binomial distribution $B(n_{max}, p)$, where p is the ratio of assignments to extend. Intuitively, higher p values will favor exploitation, whereas lower p values will favor exploration.

It might happen that the number of extensions requested n_e is larger than the number of possible extensions. In that case the agent will communicate every possible extension. The process to calculate the cost of each solution is analogous to the one described for centralized exploration. The difference lies in the aggregation of costs up the tree. Since an agent may extend a parent's assignment multiple times, it will send up the best cost out of the different extensions.



(a) Increasing treewidth, constant domain 8 and 100 variables.

(b) Increasing domain size, constant treewidth 9 and 100 variables.



(c) Additional problems solved when agents have limited memory (w.r.t. one-sided filtering).

Fig. 4: Experimental results when tightening the upper bound.

5.3 Empirical evaluation

To assess the performance of GDL with two-sided function filtering and the tighter upper bounds obtained by propagating multiple solutions, we ran experiments in the same scenarios we used in Section 4.1. Specifically, we assessed the communication and computation savings obtained by: (1) centralized exploration; (2) greedy distributed exploration; and (3) stochastic distributed exploration. Regarding the stochastic case, we empirically observed that different exploration ratios (different values for p), do not lead to very significant differences when filtering. Here we set p=0.1 because it provided slightly better results.

Note that, the harder the problem, the cheaper to propagate multiple solutions with respect to the cost propagation phase. Hence, we ran our experiments with different numbers of propagated solutions, and found that propagating 1024 solutions yielded the best results overall. This is, propagating less than 1024 solu-

tions slightly decreased the computation and communication used when solving the easier problems, but significantly increased when solving the harder ones. Likewise, propagating more solutions led to no additional savings on harder problems, while increasing costs on easier ones.

Figure 4a shows the evolution of the median results as the treewidth increases. On the one hand, centralized exploration achieves between 1 and 4% extra communication savings on top of two-sided filtering. On the other hand, both greedy and stochastic exploration outperform centralized exploration, consistently saving a median 20% communication cost, for a grand total of up to 72% savings when compared to the state-of-the-art one-sided filtering. Figure 4b displays very similar trends as variables' domain grows. Centralized exploration provides a low reduction in communication, whereas greedy and stochastic exploration strategies obtain up to 24% extra savings with respect to two-sided filtering.

Finally, it is important to note that both greedy and stochastic exploration further reduce the number of non-concurrent constraint checks by as much as 24%. Furthermore, the reduction of computational effort goes up to 32% once multiple solutions propagation strategies are combined with two-sided filtering. Figure 4c reveals the effect of this reduction on the number of problems that can be solved when nodes have limited memory. Specifically, using two-sided filtering with distributed exploration helps solve up to 75% more problems than one-sided filtering.

6 Conclusions

Function filtering [12] is a technique that reduces the size of cost functions by filtering out tuples that are found unfeasible to be extended into an optimal solution. Function filtering can be readily applied to GDL, as detailed in [3]. This paper improves the effectiveness of state-of-the-art function filtering by providing techniques to assess tighter lower and upper bounds. Such improvements lead to significant reductions in the amount of resources required to optimally solve DCOPs. Thus, we can reduce up to 72% on communication costs and up to 32% on computational costs. Furthermore, we also obtain a significant memory reduction, allowing agents to solve up to 75% more problem instances given the same constraints. To summarize, we increased the range of problems that can be solved optimally by algorithms employing function filtering.

References

1. Srinivas M. Aji and Robert J. McEliece. The generalized distributive law. *IEEE Transactions on Information Theory*, 46(2):325–343, 2000.
2. B. Bollobas. *Random Graphs*. Cambridge University Press, 2001.
3. Ismel Brito and Pedro Meseguer. Cluster tree elimination for distributed constraint optimization with quality guarantees. *Fund. Informaticae*, 102:263–286, 2010.

4. Rina Dechter, Kalev Kask, and Javier Larrosa. A general scheme for multiple lower bound computation in constraint optimization. In *Principles and Practice of Constraint Program.*, pages 346–360, 2001.
5. Rina Dechter and Irina Rish. Mini-buckets: A general scheme for bounded inference. *J. ACM*, 50:107–153, March 2003.
6. Rajiv T. Maheswaran, Milind Tambe, Emma Bowring, Jonathan P. Pearce, and Pradeep Varakantham. Taking dcop to the real world: Efficient complete solutions for distributed multi-event scheduling. In *AAMAS*, pages 310–317, 2004.
7. Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe, and Makoto Yokoo. Adopt: asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1-2):149–180, 2005.
8. Adrian Petcu and Boi Faltings. A scalable method for multiagent constraint optimization. In *IJCAI*, pages 266–271, 2005.
9. Marc Pujol-Gonzalez, Jesus Cerquides, Pedro Meseguer, and Juan A. Rodriguez-Aguilar. Communication-constrained dcops: Message approximation in gdl with function filtering. In *AAMAS*, 2011.
10. A. Rogers, A. Farinelli, R. Stranders, and N.R. Jennings. Bounded approximate decentralised coordination via the max-sum algorithm. *Art. Intelligence*, 175(2):730–759, 2011.
11. Emma Rollon and Rina Dechter. New mini-bucket partitioning heuristics for bounding the probability of evidence. In *AAAI*, pages 1199–1204, 2010.
12. Martí Sánchez, Javier Larrosa, and Pedro Meseguer. Improving tree decomposition methods with function filtering. In *IJCAI*, pages 1537–1538, 2005.
13. Meritxell Vinyals, Juan A. Rodríguez-Aguilar, and Jesús Cerquides. Constructing a unifying theory of dynamic programming dcop algorithms via the generalized distributive law. *JAAMAS*, pages 1–26, 2010.
14. F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics*, 1:80–83, 1945.
15. Weixiong Zhang, Guandong Wang, Zhao Xing, and Lars 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.