A Generic Domain Pruning Technique for GDL-Based DCOP Algorithms in Cooperative Multi-Agent Systems

Md. Mosaddek Khan
School of Electronics and Computer Science, University of Southampton
Southampton, UK
mmk1g14@ecs.soton.ac.uk

Long Tran-Thanh
School of Electronics and Computer Science, University of Southampton
Southampton, UK
ltt08r@ecs.soton.ac.uk

Nicholas R. Jennings
Departments of Computing and Electrical and Electronic Engineering, Imperial College London, London, UK
n.jennings@imperial.ac.uk

ABSTRACT

Generalized Distributive Law (GDL) based message passing algorithms, such as Max-Sum and Bounded Max-Sum, are often used to solve distributed constraint optimization problems in cooperative multi-agent systems (MAS). However, scalability becomes a challenge when these algorithms have to deal with constraint functions with high arity or variables with a large domain size. In either case, the ensuing exponential growth of search space can make such algorithms computationally infeasible in practice. To address this issue, we develop a generic domain pruning technique that enables these algorithms to be effectively applied to larger and more complex problems. We theoretically prove that the pruned search space obtained by our approach does not affect the outcome of the algorithms. Moreover, our empirical evaluation illustrates a significant reduction of the search space, ranging from 33% to 81%, without affecting the solution quality of the algorithms, compared to the state-of-the-art.

KEYWORDS

Distributed Problem Solving; DCOP; GDL; Maximization Operation

ACM Reference Format:
Md. Mosaddek Khan, Long Tran-Thanh, and Nicholas R. Jennings. 2018. A Generic Domain Pruning Technique for GDL-Based DCOP Algorithms in Cooperative Multi-Agent Systems. In *Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018), Stockholm, Sweden, July 10–15, 2018,* IFAAMAS, 9 pages.

1 INTRODUCTION

Distributed Constraint Optimization Problems (DCOPs) are a widely studied framework for coordinating interactions in cooperative multi-agent systems. DCOPs have gained such popularity because of their ability to optimize a global objective function that can be described as the aggregation of a number of distributed constraint cost functions. Each of these functions, representing a local constraint, can be defined by a set of variables held by the corresponding agents related to that constraint. In more detail, each agent holds one or more variables, each of which takes values from a finite domain. The agent is responsible for setting the value of its own variable(s) but can communicate with other agents to potentially influence their choice. The goal of a DCOP solution approach is to set every variable to a value from its domain and minimize the number of constraint violations.

Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018), M. Dastani, G. Sukthankar, E. André, S. Koenig (eds.), July 10–15, 2018, Stockholm, Sweden.

For over a decade, a number of algorithms have been developed to solve DCOPs, and they have been applied to many real world applications. These algorithms can be broadly classified into exact and non-exact approaches. The former always finds a globally optimal solution. However, finding an optimal solution is an NP-hard problem that exhibits an exponentially increasing coordination overhead as the system grows [10, 11, 16]. On the contrary, the latter approaches sacrifice some solution quality for scalability and have, therefore, been used more in practice [3, 9, 17].

Among the non-exact approaches, Generalized Distributive Law (GDL) based algorithms, such as Max-Sum [3] and Bounded Max-Sum (BMS) [13], have received particular attention. Agents in this class of algorithms calculate and propagate utilities (or costs) for each possible value assignment of their neighbouring agents' variables. Thus, the agents explicitly share the consequences of choosing non-preferred states with the preferred one during inference through a graphical representation such as factor graphs or junction trees [4, 6, 7]. Eventually, this information helps these algorithms to achieve good solution quality for large and complex problems. Moreover, unlike many other DCOP solution approaches, GDL-based algorithms explicitly support more than one variable per agent [2, 4]. As a consequence, these algorithms can easily be deployed to any DCOP setting without depending on an additional reformulation technique. Furthermore, they make efficient use of constrained computational and communication resources [3, 13]. This is achieved by following a message passing protocol in which the agents continuously exchange messages to compute an approximation of the impact that each of the agents' actions have on the global optimization function [6]. This involves building a local objective function (expounded in Section 2). Once the function is built, each of the agents picks the value of a variable that maximizes the function.

Despite these aforementioned advantages, scalability remains a widely acknowledged challenge for GDL-based algorithms [5, 12]. Specifically, they perform repetitive maximization operations for each constraint function to select the locally best configuration of the associated variables, given the local utility function and a set of incoming messages. To be precise, a constraint function that depends on n variables having domains composed of d values each, will need to perform d^n computations for a maximization operation. As the system scales up, the complexity of this step grows exponentially.

Over the past few years, a number of efforts have tried to improve the scalability of GDL-based message passing algorithms by reducing the cost of the maximization operator. In particular, [12]

and [8] reduce the domain size of variables associated with constraint functions for task allocation domains where agents' action choices are strictly divided into working on a task or not. However, this method is completely application dependent, because it can only be applied to a specific problem formulation of task allocation domain. Moreover, [14] perform a branch and bound search with constraint functions that ensure the upper and lower bound can be evaluated with a subset of variable values. However, the bounding function they propose to achieve this is solely focused on coordinating mobile sensors. Hence, it is not directly applicable to general DCOP settings. A more general approach to reduce the cost of the maximization operator, called Generalized Fast Belief Propagation (G-FBP), is proposed in [5]. In this approach, they select and sort the top $cd^{\frac{n-1}{2}}$ values of the search space, presuming the maximum value can be found from these ranges. Here, c is a constant. Nevertheless, they also admit that they cannot guarantee in advance whether the presumption is true or false, and in the latter case G-FBP incurs a significant penalty in terms of the computational cost (as we shall see in the empirical results).

Against this background, this paper proposes a **G**eneric **D**omain **P**runing technique, that we call $\mathcal{GDP}$, that is applicable to all DCOP settings. To be exact, $\mathcal{GDP}$ operates as a part of the maximization operator, proveably without affecting its solution quality (see Lemma 3.2). In other words, we improve the computational efficiency of all GDL-based algorithms by reducing the search space over which the maximization operation is computed. We empirically evaluate the performance of our approach, and we observe a significant reduction of search space, ranging from 33% to 81% by using this technique. More importantly, we show the relative performance gain of $\mathcal{GDP}$ gets better with an increase in the variables' domain size and the constraint functions' arity, in which the maximization operator acts on.

The remainder of this paper is structured as follows. We describe the problem in more detail in the section that follows. Then, in Section 3, we discuss the complete process of $\mathcal{GDP}$ with a worked example. We end this section by providing theoretical analyses. Afterwards, in Section 4, we present the empirical results of our method compared to the current state-of-the-art, and Section 5 concludes.

2 BACKGROUND AND PROBLEM FORMULATION

Formally, a DCOP can be defined by a tuple $\langle X, D, F, A, \delta \rangle$ [10], where X is a set of discrete variables $\{x_0, x_1, \ldots, x_m\}$ and $D = \{D_0, D_1, \ldots, D_m\}$ is a set of discrete and finite variable domains. Each variable x_i can take value from the states of the domain D_i. F is a set of constraint functions $\{F_1, F_2, \ldots, F_L\}$, where each $F_i \in F$ is a function dependent on a subset of variables $\mathbf{x}_i \in X$ defining the relationship among the variables in $\mathbf{x}_i$. Thus, the function $F_i(\mathbf{x}_i)$ denotes the value for each possible assignment of the variables in $\mathbf{x}_i$. Notably, the dependencies between the variables and the functions generate a bipartite graph, called a factor graph, which is commonly used as a graphical representation of such DCOPs [6]. In a factor graph, each constraint function $F_i(\mathbf{x}_i)$ is represented by a square node and is connected to each of its associated variable nodes $\mathbf{x}_i$ (denoted by circles) by an individual edge. Hence, $|\mathbf{x}_i|$ is the arity of $F_i(\mathbf{x}_i)$ in this particular graphical representation of DCOP. The nodes (variables and functions[1]) of a factor graph G are being held by a set of agents $A = \{A_1, A_2, \ldots A_k\}$. This mapping of nodes to agents is represented by $\delta : \eta \rightarrow A$. Here, η stands for the set of nodes within the factor graph. Each variable/function is being held by a single agent. However, each agent can hold several variables/functions. The corresponding agent acts (i.e. generates and transmits messages) on behalf of the nodes they hold, and is responsible for assigning values to the variables they hold. Within the model, the objective of a DCOP algorithm is to have each agent assign values to its associated variables from their corresponding domains in order to either maximize or minimize the aggregated global objective function, which eventually produces the value of each variable, X^* (Equation 1).

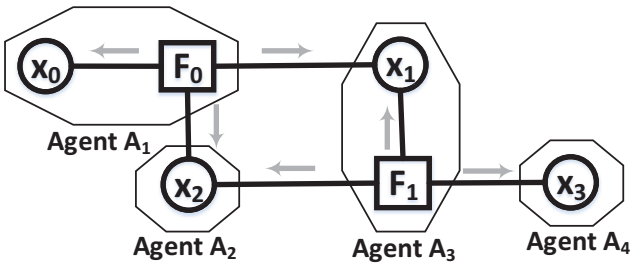


Figure 1: A sample factor graph representation of a DCOP, with two function/factor nodes $\{F_0, F_1\}$ and four variable nodes $\{x_0, x_1, x_2, x_3\}$, illustrating a global objective function $F(x_0, x_1, x_2, x_3)$. In the figure, variables are denoted by circles, factors are squares and agents are octagons. Here, the grey arrows are used to highlight the factor-to-variable messages of GDL-based algorithms, each of which requires the maximization operation to be performed.

$$X^* = \arg\max_X \sum_{i=1}^{L} F_i(\mathbf{x}_i) \vee X^* = \arg\min_X \sum_{i=1}^{L} F_i(\mathbf{x}_i) \qquad (1)$$

For example, Figure 1 illustrates the relationship among variables, functions and agents of a factor graph representation of a sample DCOP. Here, we have a set of four variables $X = \{x_0, x_1, x_2, x_3\}$, a set of two functions/factors $F = \{F_0, F_1\}$, and a set of four agents $A = \{A_1, A_2, A_3, A_4\}$. Moreover, $D = \{D_0, D_1, D_2, D_3\}$ is a set of discrete and finite variable domains, each variable $x_i \in X$ can take its value from the domain D_i. In this example, agent A_1 holds a function node F_0 and a variable node x_0. Similarly, nodes F_1 and x_1 are being held by agent A_3. While agents A_2 and A_4 hold variable nodes x_2 and x_3, respectively. In this particular setting, four agents A_1, A_2 A_3 and A_4 participate in the optimization process in order to either maximize or minimize a global objective function $F(x_0, x_1, x_2, x_3)$. Here, the global objective function is an aggregation of two local functions $F_0(x_0, x_1, x_2)$ and $F_1(x_1, x_2, x_3)$. In the factor graph, F_0 is associated (i.e. connected) with three variable nodes, and as such, the arity of the constraint function F_0 is 3. Similar to F_0, the arity of constraint function F_1 is 3 in this particular example.

[1]The term function is also known as factor, and they are used interchangeably throughout this paper.

In general, GDL-based inference algorithms follow a message passing protocol to exchange messages among the nodes of the factor graph representation of the aforementioned DCOP formulation [1, 3, 6]. Notably, both the Max-Sum and BMS algorithms use Equations 2 and 3 for their message passing, and they can be directly applied to the factor graph. Specifically, the variable and function nodes of a factor graph continuously exchange messages (variable x_i to function F_j (Equation 2) and function F_j to variable x_i (Equation 3)) to compute an approximation of the impact that each of the agents' actions have on the global objective function by building a local objective function $Z_i(x_i)$. In Equations 2–4, M_i stands for the set of functions connected to variable x_i and N_j represents the set of variables connected to function F_j. Once the function is built (Equation 4), each agent picks the value of a variable that maximizes the function by finding $\arg\max_{x_i}(Z_i(x_i))$.

$$Q_{x_i \to F_j}(x_i) = \sum_{F_k \in M_i \setminus F_j} R_{F_k \to x_i}(x_i) \tag{2}$$

$$R_{F_j \to x_i}(x_i) = \max_{\mathbf{x}_j \setminus x_i} [F_j(\mathbf{x}_j) + \sum_{x_k \in N_j \setminus x_i} Q_{x_k \to F_j}(x_k)] \tag{3}$$

$$Z_i(x_i) = \sum_{F_j \in M_i} R_{F_j \to x_i}(x_i) \tag{4}$$

As discussed previously, due to the potentially large parameter domain size and constraint functions with high arity, the maximization operator of the factor-to-variable message is the main reason GDL-based algorithms can be computationally expensive. This can be visualized from an example where a function node has five variable nodes connected to it, meaning the arity of the function is $n = 5$. Here, we assume each of the variables can take its value from 9 possible options (i.e. states of the domain), implying that the domain size is $d = 9$ for each of the variables. In this case, the function node has to perform 9^5 or $59,049$ operations to generate a message for one of its neighbouring variable nodes. Now, each of the function nodes in a factor graph has to generate and send a single message to each of its neighbours to complete a single round of message passing [1, 6]. For example, function node F_0 of Figure 1 has to send a distinct message (grey arrow) to each of its neighbouring variable nodes x_0, x_1 and x_2. Each of these messages includes the expensive maximization operator. Under such circumstances, it is possible to significantly reduce the computational cost of this step. Meanwhile, it is essential to ensure that this reduction process does not limit the algorithms' applicability, as well as not affecting the solution quality. We deal with the issue that arises from the trade-off in the section that follows.

3 THE GENERIC DOMAIN PRUNING TECHNIQUE

$\mathcal{GDP}$ (Algorithm 1) works as a part of Equation 3, which represents a function-to-variable message of a GDL-based algorithm, in order to reduce the search space over which the maximization needs to be computed. This algorithm requires as inputs a sending function node $F_j(\mathbf{x}_j)$ whose utility depends on a set of variable nodes ($\mathbf{x}_j$) associated with it (i.e. neighbours), a receiving variable node $x_i \in \mathbf{x}_j$ and all the incoming messages from the neighbour(s) of F_j apart from the receiving node x_i, denoted as $\mathcal{M}_{\mathbf{x}_j \setminus x_i}$. Finally, $\mathcal{GDP}$ returns a pruned range of values for each state of the domains of the variables over which the maximization operation needs to be performed to generate the message from the function node F_j to the variable node x_i (i.e. $R_{F_j \to x_i}(x_i)$).

In more detail, $\mathbb{S}$ stands for a set $\{\mathbf{s_1}, \mathbf{s_2}, \ldots, \mathbf{s_r}\}$ representing each state of the domains corresponding to $\mathbf{x}_j$ (line 1 of Algorithm 1). This implies that $\mathbb{S}$ is the union ($\cup$) of those sets of states, each of which corresponds to the domain of a variable in $\mathbf{x}_j$. Line 2 sorts the local utility of the sending function node F_j independently by each state $\mathbf{s_i} \in \mathbb{S}$. This sorting can be carried out at runtime of a message passing algorithm without incurring an additional delay (discussed shortly in Conjecture 3.1). Then the total number of incoming messages received by F_j is represented by $\mathbf{n}$ (line 3). Note that, a complete worked example of $\mathcal{GDP}$ is depicted in Figure 2 where we use a part of the factor graph of Figure 1 to show a factor-to-variable (i.e. F_1 to x_3) message computation (Figure 2a), as well as the operation of $\mathcal{GDP}$ on it (Figure 2b). Here, the local utility of the sending function node F_1 is shown in a table at the left side of Figure 2a, which is based on three domain states $\{R, B, G\}$ (for simplicity red, blue and green colours are used to distinguish the values of the states, respectively) and three neighbouring variable nodes x_1, x_2 and x_3. Moreover, the direction of two incoming messages (i.e. $\mathbf{n} = 2$) received by F_1, $\{122, 130, 136\}$ and $\{90, 81, 75\}$, from the variable nodes x_1 and x_2 respectively, are indicated using the black arrows. Then, the grey arrow indicates the desired function-to-variable message $R_{F_1 \to x_3}(x_3) = \{256, 263, 258\}$, and the complete calculation is depicted in a table at the right side of Figure 2a.

At this point, line 4 computes m which is the summation of the maximum values of each of the messages $\mathcal{M}_k \in \mathcal{M}_{\mathbf{x}_j \setminus x_i}$ received by the sending function F_j, other than the receiving variable node x_i. Here, $\mathcal{M}_k$ is one of the $\mathbf{n}$ messages received by F_j. In the worked example of Figure 2b, since the maximum of the received messages $\{122, 130, 136\}$ (i.e. $\mathcal{M}_1$) and $\{90, 81, 75\}$ (i.e. $\mathcal{M}_2$) by F_1 are 136 and 90 respectively, the value of $m = 136 + 90 = 226$. Now, the for loop in lines $5-14$ generates the range of the values for each state $\mathbf{s_i} \in \mathbb{S}$ from where we will always find the maximum value for the function F_j, and discard the rest. To this end, the function $sortedVal_{\mathbf{s_i}}(F_j(\mathbf{x}_j))$ gets the sorted value of $\mathbf{s_i}$ from line 2, and stores them in an array $\mathcal{V}_i$ (line 6). Then, line 7 finds p, which is the maximum of the local utility values for the state $\mathbf{s_i}$ (i.e. $\max(\mathcal{V}_i)$). In the worked example, the sorted values of domain state R are stored in $\mathcal{V}_R$, depicted in the right side of Figure 2b. Hence, the value of $p = \max(\mathcal{V}_R) = 40$. Afterwards, line 8 computes b, which is the summation of the corresponding values of p from the incoming messages of F_j (i.e. $val_p(\mathcal{M}_k)$). In the example, the values corresponding to p (i.e. 40) from two incoming messages are 130 and 75, thus the value of $b = 130 + 75 = 205$. This can be seen in the first row of the rightmost table of Figure 2b. The rows related to the computation for the state R are summarized into this table from the rightmost table of Figure 2a, which depicts the complete computation of the function F_1 to variable x_3 message based on domain states R, B and G. Having obtained the value of m and b from lines 4 and 8 respectively, line 9 gets the base case t, which is a subtraction of b from m (i.e. $t = m - b = 226 - 205 = 21$).

Algorithm 1: Generic Domain Pruning- $\mathcal{GDP}(F_j(\mathbf{x}_j), x_i, \mathcal{M}_{\mathbf{x}_j \setminus x_i})$

Input: $F_j(\mathbf{x}_j)$ - Local utility of the sending function node F_j, where $\mathbf{x}_j$ is the set of variable nodes associated with F_j;
$x_i \in \mathbf{x}_j$ - the variable node which is going to receive a message from F_j;
$\mathcal{M}_{\mathbf{x}_j \setminus x_i}$ - Received messages by F_j from all of its neighbouring variable nodes $\mathbf{x}_j$, other than x_i.

Output: Pruned range of values of the states over which maximization needs to be performed to generate the message from F_j to x_i.

1 Let $\mathbb{S} = \{\mathbf{s_1}, \mathbf{s_2}, \ldots, \mathbf{s_r}\}$ be the states corresponding to the domains of $\mathbf{x}_j$

2 Sort the local utility $F_j(\mathbf{x}_j)$ independently by each state $\mathbf{s_i} \in \mathbb{S}$

3 $\mathbf{n} \leftarrow |\mathcal{M}_{\mathbf{x}_j \setminus x_i}|$

4 $m \leftarrow \sum_{k=1}^{\mathbf{n}} \max(\mathcal{M}_k)$, where $\mathcal{M}_k \in \mathcal{M}_{\mathbf{x}_j \setminus x_i}$ is one of the $\mathbf{n}$ messages received by F_j

5 **foreach** $\mathbf{s_i} \in \mathbb{S}$ **do** // for each state corresponding to the variables $\mathbf{x}_j$ that associate with F_j

6 $\quad \mathcal{V}_i \leftarrow sortedVal_{\mathbf{s}_i}(F_j(\mathbf{x}_j))$

7 $\quad p \leftarrow \max(\mathcal{V}_i)$

8 $\quad b \leftarrow \sum_{k=1}^{\mathbf{n}} val_p(\mathcal{M}_k)$

9 $\quad t \leftarrow m - b$

10 $\quad \mathbf{q} \leftarrow binarySearch(\mathcal{V}_i, \lambda)\ where\ \lambda = \max_c\{c \in \mathcal{V}_i : c \leq (p - t)\}$

11 $\quad$ **if** $\mathbf{q} == p - t$ **then**

12 $\quad\quad$ **result** $prunedRange_{\mathbf{s}_i}([p, \mathbf{q}])$

13 $\quad$ **else**

14 $\quad\quad$ **result** $prunedRange_{\mathbf{s}_i}([p, \mathbf{q}))$

Line 10 searches for a value λ in the sorted list $\mathcal{V}_i$ and stores it in a variable $\mathbf{q}$. In this context, λ stands for a value $c \in \mathcal{V}_i$ that is either equal to the value of $p - t$ or immediately smaller than $p - t$. In other words, λ is the maximum of those values in $\mathcal{V}_i$ that are not greater than $p - t$. To this end, we use the binary search method because the list that needs to be searched is already sorted. Now, when the value of $\mathbf{q}$ is equal to $p - t$, the desired maximum value for the state $\mathbf{s}_i$ must always be found by considering the rows corresponding to the values in the range $[p, \mathbf{q}]$, denoted by $prunedRange_{\mathbf{s}_i}([p, \mathbf{q}])$ (lines $11 - 12$)[2]. Otherwise, the value of $\mathbf{q}$ is less than $p - t$, and the desired maximum value for the state $\mathbf{s}_i$ must always be found by considering the rows corresponding to the values in the range $[p, \mathbf{q})$, denoted by $prunedRange_{\mathbf{s}_i}([p, \mathbf{q}))$ (lines $13 - 14$)[2]. In the worked example of Figure 2b, the value of $p - t$ is 19, given $p = 40$ and $t = 21$. The target is to obtain the value of $\mathbf{q}$ from the list $\mathcal{V}_R$. In the third column of the rightmost table that illustrates the computation for the state R, we see that the value of $\mathbf{q}$ is 13 because this is the closest smaller (or equal) value of 19 (i.e. $p - t$). Since $\mathbf{q}$ is not equal to $p - t$ in this instance, according to lines $13 - 14$ the desired maximization for R must be found by considering the rows corresponding to the values in the range $[40, 13)$ or $[40, 39]$. That means, only considering the top two rows are sufficient to obtain the desired value of R; hence, it is not necessary to consider the remaining 7 rows for this particular instance. To be exact, the value for the state R after maximization is 256, which is obtained from the row corresponding to the local utility value of 39. In this way, $\mathcal{GDP}$ reduces the computational cost of the expensive maximization operator. The grey colour is used to mark the discarded rows of the table. We can see that even for such a small instance, having domain size $d = 3$ and arity $n = 3$, $\mathcal{GDP}$ prunes more than 75% of the search space during the maximization of a state in computing the function-to-variable message.

As argued above, it is important to ensure that combining $\mathcal{GDP}$ with Equation 3 does not make the computation of a function-to-variable message prohibitively expensive. Given the sorting operation of line 2 does not incur an additional delay (see Conjecture 3.1), the time complexity of $\mathcal{GDP}$ involves two parts. This includes the for-loop of line 5 and the binary search of line 10. Hence, we determine that the overall time complexity of $\mathcal{GDP}$ is $\mathcal{O}(r \log |\mathcal{V}_i|)$. In this context, r stands for the number of states of the variables' domain associated with the sending function node (line 5). Then, $|\mathcal{V}_i|$ is the size of the array $\mathcal{V}_i$, hence $\log |\mathcal{V}_i|$ is the time complexity to do the binary search of line 10 on $\mathcal{V}_i$. Taken together, $\mathcal{GDP}$ is able to reduce the search space significantly at the expense of a quasi-linear computation cost of its own.

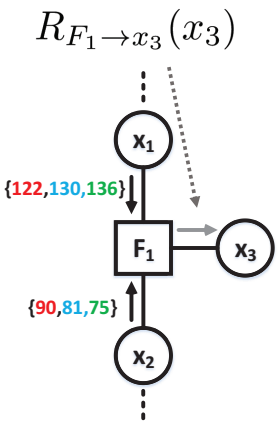
Conjecture 3.1. *The sorting operation performed in line 2 of Algorithm 1 does not incur an additional delay during the computation of a factor-to-variable message.*

Discussion. The message passing protocol followed by GDL-based algorithms operates directly on a factor graph (acyclic or cyclic) representation of a DCOP, and it can be classified into the following two categories [1, 6]:

(1) *Synchronous message passing approach. A message is sent from a node v on an edge e to its neighbouring node w if and only if all the messages are received at v on edges other than e, summarized for the node associated with e.* This implies that a node in a factor graph is not permitted to send a message to its neighbour until it receives messages from all its other neighbours. Here, for w to be able to generate and

[2] See Lemma 3.2 and its proof.

Utility (cost) table for the function node, F_1 (x_1, x_2, x_3)			
x_1	x_2	x_3	F_1
R	R	R	5
R	R	B	10
R	R	G	12
R	B	R	13
R	B	B	12
R	B	G	35
R	G	R	9
R	G	B	38
R	G	G	11
B	R	R	12
B	R	B	14
B	R	G	38
B	B	R	10
B	B	B	3
B	B	G	9
B	G	R	40
B	G	B	14
B	G	G	13
G	R	R	10
G	R	B	37
G	R	G	12
G	B	R	39
G	B	B	13
G	B	G	14
G	G	R	11
G	G	B	12
G	G	G	4



$Q_{x_1 \to F_1}(x_1)$	$Q_{x_2 \to F_1}(x_2)$	F_1		$R_{F_1 \to x_3}(x_3)$
122	90	5	217	
122	90	10	222	
122	90	12	224	
122	81	13	216	
122	81	12	215	
122	81	35	238	
122	75	9	206	
122	75	38	235	
122	75	11	208	
130	90	12	232	
130	90	14	234	
130	90	38	258	{256,263,258}
130	81	10	221	
130	81	3	214	
130	81	9	220	
130	75	40	245	
130	75	14	219	
130	75	13	218	
136	90	10	236	
136	90	37	263	
136	90	12	238	
136	81	39	256	
136	81	13	230	
136	81	14	231	
136	75	11	222	
136	75	12	223	
136	75	4	215	

(a) Computation of a factor-to-variable message (i.e. F_1 to x_3).

Domain pruning of state R during the computation of $R_{F_1 \to x_3}(x_3)$

$M_1 = Q_{x_1 \to F_1}(x_1) = \{122, 130, 136\}$

$M_2 = Q_{x_2 \to F_1}(x_2) = \{90, 81, 75\}$

$V_R = sortedVal_R(F_1(x_1, x_2, x_3)) = \{40, 39, 13, 12, 11, 10, 10, 9, 5\}$

$m = max\ (M_1) + max\ (M_2) = 136 + 90 = 226$

$p = max\ (V_R) = 40;\ Therefore, b = 130 + 75 = 205$

Given $m = 226,\quad t = m - b = 226 - 205 = 21$

Target – find the row that contains utility value $p - t$, if not then closest smaller value of $p - t$.

Here, $p - t = 40 - 21 = 19$

Thus, the desired value ***q*** is *13*, which is in the third row from the top. Use binary search to find this value

Computation for state R

$Q_{x_1 \to F_1}(x_1)$	$Q_{x_2 \to F_1}(x_2)$	F_1	*"Sum"*
130	75	40	245
136	81	39	256
122	81	13	216
130	90	12	232
136	75	11	222
136	90	10	236
130	81	10	221
122	75	9	206
122	90	5	217

"Max"

Now, if $\boldsymbol{q} == p - t$, *then the maximum value for R must be found from the rows within the range* $[p, \boldsymbol{q}]$.

Else If $\boldsymbol{q} < p - t$, *then the maximum value for R must be found from the rows within the range [p,* $\boldsymbol{q}$*).*

Here, $\boldsymbol{q} < p - t$ (i.e. $13 < 19$), hence we have to look only for the rows within the range $[40, 39]$ or $[40, 13)$, and discard the rest.

(b) Complete operation of $\mathcal{GDP}$ on $R_{F_1 \to x_3}(x_3)$.

Figure 2: Worked example of $\mathcal{GDP}$ in computing a factor-to-variable message, F_1 to x_3 or $R_{F_1 \to x_3}(x_3)$, within the factor graph shown in Figure 1. In this example, for simplicity, we show that part of the original factor graph which is necessary for this particular message computation. In the figure, red, blue and green coloured values are used to distinguish the domain states R, B and G respectively for each of the variables involved in the computation, and arrows between the nodes of the factor graph are used to indicate the direction of the corresponding messages.

send messages to all its other neighbours, it depends on the message from v. To be exact, w cannot compute and transmit messages to its neighbours other than v until it has received all essential messages, including the message from v. In this process, a single round of the message passing process will complete once each of the nodes is able to send a message to all of its neighbours.

(2) *Asynchronous message passing approach.* Nodes of a factor graph are initialized randomly, and outgoing messages can be updated at any time and in any sequence. The message passing needs to continue for a number of rounds[3] to either converge or produce an acceptable approximate solution.

Based on both of these versions of the message passing protocol, the three following cases are seen. We are going to illustrate that, for all of these cases, Conjecture 3.1 is true.

- *Case* 1*:* It is always preferable for acyclic factor graphs to use the synchronous version of message passing [1, 3]. This is because it requires only one round of message passing to generate the optimal solution in such factor graphs. Therefore, it is redundant to use the asynchronous alternative. In this case, only a very small number of nodes act (i.e. generate and transmit messages) initially, while the rest of the nodes have to wait for their required messages to arrive before they can start generating message(s). Given a sorting operation is not computationally expensive, we propose to apply $\mathcal{GDP}$ to those nodes which are not initially active. Thus, they can utilize the waiting time to complete the sorting operation without incurring an additional delay.
- *Case* 2*:* A key GDL-based DCOP algorithm, namely Bounded Max-Sum, deals with cyclic factor graph representations of DCOPs by initially removing the cycles from the original factor graph using a preprocessing step. During this step, the agents experience an additional waiting time. Then, it applies the synchronous version of message passing on the transformed acyclic graph to provide a bounded approximate solution of the problem [13]. In this case, the sorting operation can be carried out during the agents' waiting time of the preprocessing step. Hence it does not incur an additional delay.
- *Case* 3*:* The so-called *loopy message passing* [3] is another way to deal with the cyclic factor graph representation of DCOPs. It uses the asynchronous approach as the message passing protocol. As mentioned above, this version of message passing requires several rounds to either converge or produce an approximate solution. We propose to enforce the fact that the first round must follow the synchronous approach, so that the sorting operation can be completed using the same way as *Case* 1. The following rounds can then proceed with the asynchronous message passing approach without loss of its own characteristics. □

Lemma 3.2. *During a function-to-variable message computation, the desired maximum value for a state* $\mathbf{s_i} \in \mathbb{S}$ *must always be found from the rows corresponding to the values ranging from* $\mathbf{q}$ *to* p*.*

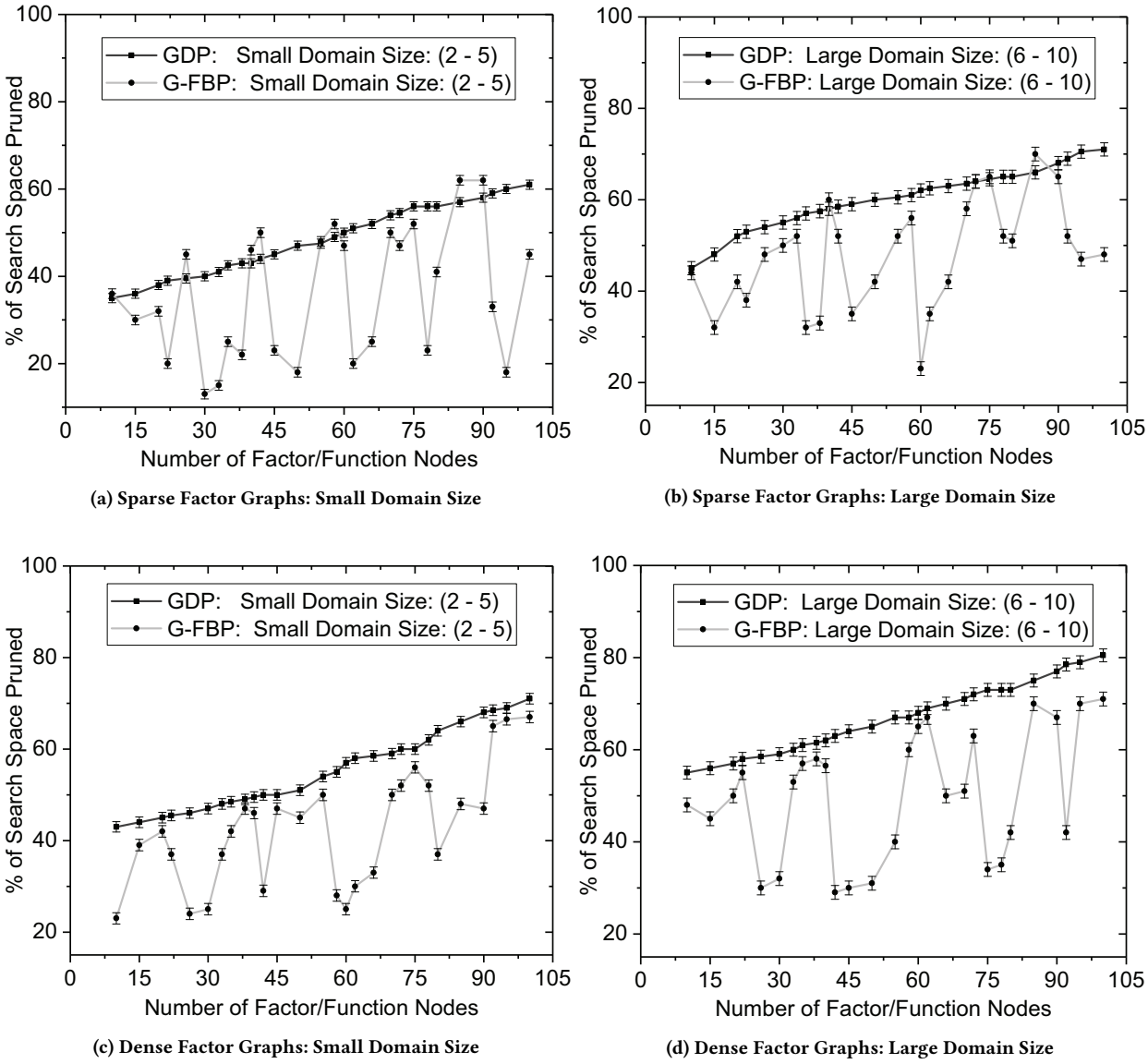
Proof. We prove this by contradiction. Assume there exists a row r_a that resides outside the range from which the maximum value for $\mathbf{s_i}$ can be found. As we know, a function-to-variable message depends on two inputs– the local utility table of the function and the incoming messages from its neighbours. In this context, p is the maximum utility corresponding to $\mathbf{s_i}$, and is within our proposed range. Therefore, to be able to find r_a, we have to rely on the only remaining input, that is the incoming messages from the neighbours of the sending function node. To this end, let's consider two parameters from this remaining input. The first is the summation of the maximum values from each of the incoming messages, denoted as m. The second is the summation of values from the incoming messages corresponding to p, denoted as b. Given p is the maximum of the first input, the value $t = m - b$ is significant because this is the maximum difference the remaining input can make. In $\mathcal{GDP}$, the value of $\mathbf{q}$ is chosen in such a way that it covers the difference. As a consequence, there exists no such row as r_a. □

4 EMPIRICAL EVALUATION

We now empirically evaluate how much speed-up can be achieved using $\mathcal{GDP}$ and compare this with the performance of G-FBP[4]. In so doing, we run our experiments on two different types of factor graph representation (i.e. sparse and dense) of a benchmarking graph colouring problem. Specifically, we consider factor graphs having a number of function nodes ranging from 10 to 100, and that each of the factor graphs is generated by randomly connecting a number of variable nodes per function node. Specifically, this number of variable nodes connected to each function node, termed the arity n of a function, has been chosen based on the following parameters: the value of n for each function node is randomly chosen from the ranges $1-4$ and $5-10$ to generate sparse and dense factor graphs, respectively. Thus, the differences in the arity of the function nodes for two different types of factor graphs are distinguished by the terms sparse and dense in our experiments. Moreover, we categorize domain size d of the variable nodes into two distinct classes. On the one hand, for a setting with *"small domain size"* we consider the size between 2 to 5 for each of the variable nodes in a factor graph. On the other hand, we consider them between 6 to 10 for a setting with *"large domain size"*. This classification has been done in order to observe the performance of $\mathcal{GDP}$ and G-FBP from a very small (e.g. $d^n = 2^3$) to a large (e.g. $d^n = 10^5$) search space. It is worth noting that we make use of the Frodo framework [15] to generate local utility tables (i.e. cost function) for the function nodes of a factor graph. To get the results based on the aforementioned setting, we initially compute the percentage of the search space pruned (i.e. speed-up) by the algorithms for a function node by taking the average of the speed-ups of all the messages sent by that function node. Afterwards, we take the average of the speed-ups of all the nodes in a factor graph. Finally, we report the results of each factor graph averaged over 50 test runs in Figure 3, recording standard errors to ensure statistical significance. All of the experiments were performed on a simulator implemented in an Intel *i*7 Quadcore 3.4GHz machine with 16GB of RAM. Note, both the algorithms, $\mathcal{GDP}$ and G-FBP, operate

[3] A detailed description regarding how many rounds are required is beyond the scope of this paper. See [3] for more details.

[4] See Section 1 for the detailed reasoning behind the selection of this benchmark.



(a) Sparse Factor Graphs: Small Domain Size

(b) Sparse Factor Graphs: Large Domain Size

(c) Dense Factor Graphs: Small Domain Size

(d) Dense Factor Graphs: Large Domain Size

Figure 3: Empirical results: $\mathcal{GDP}$ vs G-FBP– for the factor graph representations of different instances of the graph colouring problem. In the figure, we use the terms sparse and dense to indicate the disparity in arity among the function nodes. Error bars are calculated using standard error of the mean.

only on the function-to-variable messages of a factor graph in order to reduce the computation cost of the maximization operator. Therefore, experimenting with other typical DCOP parameters and metrics, such as communication cost, message size and completion time, is beyond the scope of this paper [5, 14].

Figures 3a–3b and Figures 3c–3d illustrate the performance of $\mathcal{GDP}$ and G-FBP for sparse and dense factor graphs of $10 - 100$ function nodes, respectively. In the figures, the black lines depict the results of $\mathcal{GDP}$, while the results of G-FBP are shown using the grey lines. More precisely, the black line of Figure 3a shows the results of $\mathcal{GDP}$ obtained from the factor graphs having variable nodes with *small domain size*. For the same algorithm, the results of the factor graphs having *large domain size* variable nodes are shown using the black line of Figure 3b. It can be clearly seen from those two black lines of both the figures that $\mathcal{GDP}$ always performs better when the variables take their values from a larger domain size, given that the value of the arity n remains identical. Moreover, the performance of $\mathcal{GDP}$ increases steadily with the number of function nodes for both the cases. This trend indicates that $\mathcal{GDP}$ performs correspondingly better when the scale of the factor graph becomes larger. Note that neither all the nodes, nor all the function-to-variable messages experience similar performance

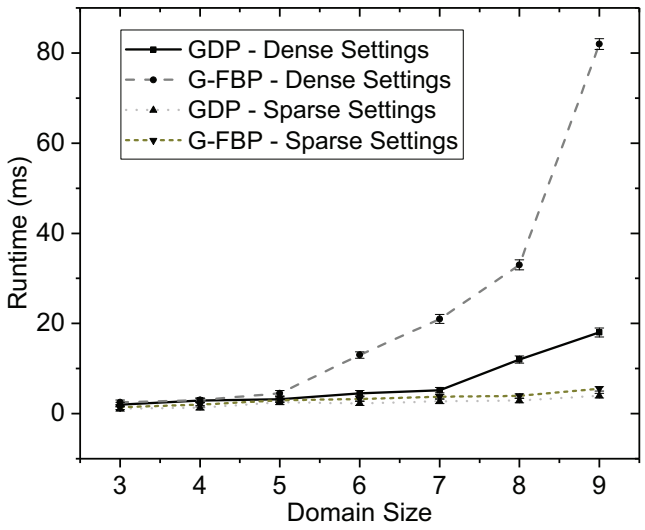


Figure 4: Comparative cost of $\mathcal{GDP}$ and G-FBP in terms of their runtime on top of the maximization operator. Error bars are calculated using standard error of the mean.

from the proposed approach, due to their differences in the content of the utility tables and incoming messages.

In more detail, $\mathcal{GDP}$ running over sparse factor graphs having $10-40$ function nodes and variables with *small domain size* prunes around $33-42\%$ of the search space during the computation of the maximization operation. On the other hand, $\mathcal{GDP}$ prunes around $40-48\%$ in the dense factor graph (Figure 3c) with a similar setting. This indicates that our approach performs significantly better in dense factor graphs, where the value of arity n is larger, compared to the sparse factor graphs. A similar trend is observed in the larger factor graphs of Figures 3a and 3c. For instance, having $75-100$ function nodes and *small domain size* variable nodes, $\mathcal{GDP}$ prunes around $55-61\%$ and $60-70\%$ of the search space for sparse and dense factor graphs, respectively. On the other hand, it is observed from the results reported in Figure 3 that $\mathcal{GDP}$ always performs better when the domain size of the variable nodes are larger, given the remaining parameters are identical. In the sparse setting, we observe around $60-72\%$ reduction of search space by our approach when applied on the factor graph of $65-100$ function nodes and *large domain size* of the variable nodes (Figure 3b). Notably, the performance gain from $\mathcal{GDP}$ reaches its maximum level (i.e. $70-81\%$) in the dense factor graph with similar setting (Figure 3d). This is important because it gives us a clear indication that $\mathcal{GDP}$ is able to prune the maximum amount of search space when the values of n and d becomes larger.

As mentioned already, the grey lines of Figures 3a – 3d illustrate the results of G-FBP for the same settings as $\mathcal{GDP}$. It can be seen from the results that the performance obtained from G-FBP fluctuates throughout all the cases. The insight behind this trend is due to the fact that G-FBP is based on an intuition that the maximum value can be found from the partially sorted top $cd^{\frac{n-1}{2}}$ values (see Section 1 for details). When this presumption is false, it incurs a significant penalty in terms of the computation cost (i.e. search space). As a consequence, although we observe a good reduction of the search space by G-FBP for a number of nodes in a factor graph, its overall performance for a complete factor graph is neither guaranteed, nor consistent. Taken together, the aforementioned results clearly show a significant reduction of search space by $\mathcal{GDP}$ while computing the maximization of function-to-variable messages within a factor graph. In contrast, although G-FBP prunes more of the search space for a number of instances, its overall performance is worse than $\mathcal{GDP}$ most of the time because of the consistency issue. This highlights a key shortcoming of G-FBP is that it is not consistent in pruning the search space, while our approach performs better consistently with the growth of the arity and domain size.

In the final experiment, we analyse whether $\mathcal{GDP}$ and G-FBP are prohibitively expensive in terms of their execution time. We have to check this because both the algorithms trade this time in order to generate the pruned search space. To this end, Figure 4 illustrates this result for both the sparse and dense settings defined in the previous experiment. The results reported in the figure are generated by taking the average of ten different messages for each of the domain sizes (d) ranging from 3 to 9. On the one hand, it can be clearly seen that the runtime of $\mathcal{GDP}$ (dotted-light-grey line) and G-FBP (short-dash-dark-yellow line) are very small and comparable for all the values of d for sparse settings. On the other hand, when the value of d is more than 5, $\mathcal{GDP}$ (black line) requires comparatively less time than G-FBP (dash-grey line) in the dense settings. Although $\mathcal{GDP}$'s runtime is smaller, none of the algorithms incur such delays that would make them prohibitively expensive to deploy. This is expected because from their complexity analysis we find that both of them require quasi-linear time (see previous section and [5]).

5 CONCLUSIONS

We presented a new algorithm, $\mathcal{GDP}$, that significantly reduces the computation cost of the maximization operator in the widely used GDL-based DCOP algorithms. We observe a significant reduction in the search space of around $33\%-81\%$ from our empirical evaluation. This is significant because by reducing the computation cost of the expensive maximization operator, we are able to accelerate the overall optimization process of this class of DCOP algorithms. Moreover, our empirical evidence clearly demonstrates that the performance of $\mathcal{GDP}$ improves with an increase in the parameters upon which the maximization operator acts. Given this, by using $\mathcal{GDP}$, we can now use GDL-based algorithms to efficiently solve large real world DCOPs. In addition, we provide a theoretical proof regarding the accuracy of our approach, which is also applicable on generic DCOP settings as opposed to some previous approaches that tend to be restricted to specific application domain(s). Significantly, rather than being a preprocessing step, we have incorporated $\mathcal{GDP}$ into a function-to-variable message of GDL-based algorithms so that they can work jointly. This particular phenomenon provides an opportunity to use existing application dependent approaches on top of $\mathcal{GDP}$ to further reduce the computational cost of the maximization operator. This will be investigated in future work.

REFERENCES

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

[2] J. B. Cerquides, A. Farinelli, P. Meseguer, and S. D Ramchurn. 2013. A tutorial on optimization for multi-agent systems. *Computer Journal* 57 (2013), 799–824.

[3] A. Farinelli, A. Rogers, A. Petcu, and N. R. Jennings. 2008. Decentralised coordination of low-power embedded devices using the max-sum algorithm. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, Vol. 2. 639–646.

[4] F. Fioretto, E. Pontelli, and W. Yeoh. 2018. Distributed constraint optimization problems and applications: A survey. *Journal of Artificial Intelligence Research* 61 (2018), 623–698.

[5] Y. Kim and V. Lesser. 2013. Improved max-sum algorithm for DCOP with n-ary constraints. In *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. 191–198.

[6] F. R. Kschischang, B. J Frey, and H.A. Loeliger. 2001. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory* 47, 2 (2001), 498–519.

[7] A. R. Leite, F. Enembreck, and J. A. Barthès. 2014. Distributed constraint optimization problems: review and perspectives. *Expert Systems with Applications* 41, 11 (2014), 5139–5157.

[8] K. S. Macarthur, R. Stranders, S. D. Ramchurn, and N. R. Jennings. 2011. A distributed anytime algorithm for dynamic task allocation in multi-agent systems. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence*. 701–706.

[9] R. T. Maheswaran, J. P. Pearce, and M. Tambe. 2004. Distributed algorithms for DCOP: A graphical-game-based approach. In *Proceedings of the ISCA 17th International Conference on Parallel and Distributed Computing Systems (ISCA PDCS)*. 432–439.

[10] P. J. Modi, W. Shen, M. Tambe, and M. Yokoo. 2005. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence* 161, 1 (2005), 149–180.

[11] Faltings B. Petcu, A. 2005. A scalable method for multiagent constraint optimization. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*. 266–271.

[12] S. D. Ramchurn, A. Farinelli, K. S. Macarthur, and N. R. Jennings. 2010. Decentralized coordination in robocup rescue. *Computer Journal* 53 (2010), 1447–1461.

[13] A. Rogers, A. Farinelli, R. Stranders, and N.R. Jennings. 2011. Bounded approximate decentralised coordination via the max-sum algorithm. *Artificial Intelligence* (2011), 730–759.

[14] R. Stranders, A. Farinelli, A. Rogers, and N. R Jennings. 2009. Decentralised coordination of mobile sensors using the max-sum algorithm. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, Vol. 9. 299–304.

[15] L. Thomas, O. Brammert, and S. Radoslaw. 2009. FRODO 2.0: An Open-Source Framework for Distributed Constraint Optimization. In *Proceedings of the IJCAI'09 Distributed Constraint Reasoning Workshop (DCR'09)*. 160–164. https://frodo-ai.tech.

[16] W. Yeoh, A. Felner, and S. Koenig. 2010. BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. *Journal of Artificial Intelligence Research* 38 (2010), 85–133.

[17] R. Zivan, S. Okamoto, and H. Peled. 2014. Explorative anytime local search for distributed constraint optimization. *Artificial Intelligence* 212 (2014), 1–26.