24th European Conference on Artificial Intelligence - ECAI 2020
Santiago de Compostela, Spain

Resilient Distributed Constraint Optimization in Physical Multi-Agent Systems

Pierre Rust[1] and **Gauthier Picard**[2] and **Fano Ramparany**[1]

Abstract. We model the notion of k-resilient distribution of computation graphs supporting agent decisions, over dynamic physical multi-agent systems (e.g. IoT systems). We devise a self-organizing repair method, namely DRPM[DMCM], based on distributed optimization to repair the distribution as to ensure the system still performs collective decisions and remains resilient to upcoming changes. Resilience is based on the concept of replicas of computations so that those hosted by disappearing agents can activate on other agents. We focus on a particular type of reasoning process to repair: distributed constraint optimization (DCOP), where computations are decision variables and constraints distributed over a set of agents. We provide a full stack of mechanisms to install resilience in operating stateless DCOP algorithms, which results in an robust approach using MGM-2 to repair any stateless DCOP algorithm at runtime. We experimentally evaluate the performances of our methods on different topologies (uniform or problem-dependent) operating DCOP algorithms (A-MaxSum and A-DSA) to solve classical benchmarks (random graph, graph coloring) while agents are disappearing.

1 INTRODUCTION

We consider the problem of distributing a set of *computations* supporting decisions over a set of agents embodied in physical devices (or *nodes*), like robots, sensors or autonomous cars. Coordinated decisions are organized in a computation graph, where vertices represent computations and edges represent a dependency relation between computations. This appears in many decision models, like factor graphs and constraints graphs when solving Distributed Constraints Optimization Problems (DCOP) [6], or computation graph algorithms such as those addressed by Pregel or other BSP-based frameworks [12]. While these frameworks usually target high-performance cluster computing, we consider here Internet-of-Things, edge computing or robot swarm scenarios, where computations run on distributed, highly heterogeneous nodes and where a central coordination might not be desirable or even not possible [3].

Such systems must be able to cope with agents failures: when an agent stops responding, other agents in the system must run the orphaned computations. We define the notion of k-resilience, which characterizes systems able to provide the same functionalities or decisions even when up to k nodes disappear. As far as we know, only [19, 20][3] addressed the problem of adapting decisions distribution at runtime and proposed a model for computing such distribution.

The contributions structure the paper as follows. Section 2 defines the notion of optimal distribution of graph-based computations over a given physical infrastructure. Section 3 expounds the notion of k-resilience and the life cyle of a k-resilient system. Section 4 presents DRPM, a solution method to deploy replicas of decision at runtime, to ensure k-resilience. We devise a distributed repair method, namely DMCM, based on a DCOP model using replicated computations to adapt the decision deployment following failures in the physical multi-agent system in Section 5. The general framework of our approach is summarized in Figure 1. We evaluate experimentally our algorithmic contributions on different topologies of multi-agent systems whose functionality is to operate distributed constraint reasoning while agents are leaving the system, in Section 6. Collective problems to solve are classical DCOP benchmarks. We notably execute asynchronous versions of Max-Sum and DSA algorithms in such dynamic settings. Finally, we conclude the paper on some perspectives.

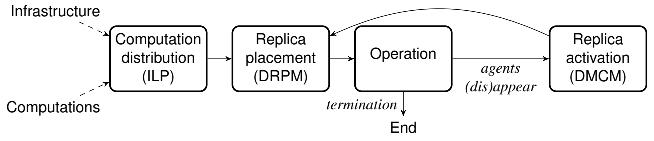


Figure 1: DRPM[DMCM] life cycle in a glance.

2 ASSIGNING COMPUTATIONS TO AGENTS

The placement of decision-related *computations* on physical agents has an important impact on the performance characteristics of the global system: some placements improve response time, as studied by [10], some others favor communication load between agents and some others optimize for other criteria like QoS or running cost.

Let $G = \langle \mathbf{X}, D \rangle$ be a computation graph, where $\mathbf{X}$ is the set of computations x_i, and D is the set of edges (i, j) representing the dependencies between computations (which implies that messages between neighbors computations are passed along these edges). Let $\mathbf{A}$ be the set of agents which can host the computations $x_i \in \mathbf{X}$. We note $\mu : \mathbf{X} \mapsto \mathbf{A}$ the function that maps computations to agents and $\mu^{-1}(a_m)$ the set of computations hosted on agent a_m. An agent can only host a limited quantity of computations, constrained by agent's *capacity* $\mathbf{w_{max}}(a_m) \geq 0$, and computation's *weight*, $\mathbf{w}(x_i) \geq 0$. We note x_i^m the boolean value in $\{0, 1\}$ stating whether computation x_i is hosted on agent a_m.

Definition 1 *Given a set of agents* $\mathbf{A}$ *and a set of computations* $\mathbf{X}$, *a* ***distribution*** *is a mapping function* $\mu : \mathbf{X} \mapsto \mathbf{A}$ *that assigns each computation to exactly one agent and respects the agents' capacity.*

[1] Orange Labs, France, email: {pierre.rust,fano.ramparany}@orange.com
[2] Mines Saint-Etienne, France, email: picard@emse.fr
[3] This paper is an extended version of an AAMAS'19 poster [20]

Finding a distribution is a constraint satisfaction problem with:

$$\forall x_i \in \mathbf{X}, \quad \sum_{a_m \in \mathbf{A}} x_i^m = 1 \tag{1}$$

$$\forall a_m \in \mathbf{A}, \quad \sum_{x_i \in \mu^{-1}(a_m)} \mathbf{w}(x_i) \leq \mathbf{w_{max}}(a_m) \tag{2}$$

When communication is constrained (like in IoT), distribution should generate as little load as possible and favor the cheapest communication links. We model communication costs with a matrix: $\mathbf{route}(m, n) \geq 0$ is the communication cost between agents a_m and a_n. These costs may represent the financial cost of using a communication link or other characteristics like throughput or latency and infinite costs can be used when there is no communication link between agents. Let $\mathbf{msg}(i, j) \geq 0$ be the size of the messages between x_i and x_j, the communication cost between x_i on a_m and x_j on a_n is $\forall x_i, x_j \in \mathbf{X}, \forall a_m, a_n \in \mathbf{A}$:

$$\mathbf{c_{com}}(i, j, m, n) = \begin{cases} \mathbf{msg}(i, j) \cdot \mathbf{route}(m, n) & (i, j) \in D, m \neq n \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where there is no communication cost between computations hosted on the same agent. Costs to host some computation on an agent are modeled as a **host** function assigning a cost for each pair (a_m, x_j). These hosting costs model some affinity or repulsiveness between an agent and a computation, or financial infrastructure costs. Notice that one can easily force a computation to be hosted on a specific agent by assigning an infinite hosting cost for all other agents. The characteristics of many real-world systems can be accounted for, thanks to these flexible definitions of communication and hosting costs.

The quality of a distribution can then be evaluated using the following function, with $\omega \in [0, 1]$ a penalty parameter to be tuned depending on the scenario (prefering communication cost over hosting cost, and vice versa):

$$\omega \cdot \sum_{(i,j) \in D} \sum_{(m,n) \in \mathbf{A}^2} \mathbf{c_{com}}(i, j, m, n) \cdot x_i^m \cdot x_j^n \tag{4}$$

$$+ (1 - \omega) \cdot \sum_{(x_i, a_m) \in \mathbf{X} \times \mathbf{A}} x_i^m \cdot \mathbf{c_{host}}(a_m, x_i) \tag{5}$$

Definition 2 *An **optimal distribution** is a distribution μ that minimizes the cost of communication between agents and the cost of hosting computations as stated in eq. (4-5).*

Assigning computations to agents as stated in Definition 2 is equivalent to graph partitioning [18], which typically falls under the category of NP-Hard problems [2] and can be mapped to a binary Integer Linear Program (ILP) using linearization [5] of (4) and (5) as the minimization objectives, (2) as constraints, and adding constraints so that computation are only assigned to one agent. It may however drastically stress centralized modern solvers and can only be used, when bootstrapping the system, to compute an initial distribution for relatively small instances. For larger problems, a greedy heuristic (which we do not include here due to space constraints) can be used and yields close to optimal results.

3 k-RESILIENCE PRINCIPLES

Now consider the case of a dynamic system where agents may disappear. In centralized settings, when some agents fail, one could use the ILP or greedy heuristic to compute a new optimal distribution. But, such a centralized computing may not be possible or desirable, depending on the requirements of the application scenario.

We define the notion of k-resilience as the capacity for a system to repair itself and operate correctly even when up to k agents disappear. This means that after a recovery period, all computations must be active on exactly one agent and communicate one with another as specified by the graph $\mathbf{G}$.

Definition 3 *Given a set of agents $\mathbf{A}$, a set of computations $\mathbf{X}$, and a distribution μ, the system is **k-resilient** if for any $F \subset \mathbf{A}, |F| \leq k$, a new distribution $\mu' : X \rightarrow \mathbf{A} \backslash F$ exists.*

Our approach for k-resilience is based on *replication* and *relocation*. Assuming initial deployment and replica placement have been performed at system bootstrap, and the graph is dense enough and remains connected, even with very prohibitive costs (e.g. backup wireless connections) the system will execute the following repair cycle all along its lifetime (see Figure 1): (a) Detect agents' failures; (b) Activate replicas of missing computations; (c) Place new replicas for missing computations, and continue nominal operation.

3.1 Replication

Replication ensures that the definitions of the computations (which contain the definition of the system itself) are not lost and is inspired by distributed databases [24, 23]. Indeed, one pre-requisite to k-resilience is to still have access to the definition of every computation after a failure. Our approach is to keep k *replicas* (copies of definitions) of each active computation on different agents. Provided that the k replicas are placed on different agents, no matter the subset of up to k agents that fails there will always be at least one replica left after the failure. Here, we apply these ideas except we keep replicas of computation definitions instead of data records, which implies that computations must be *stateless* or that their state must be restorable. Let's note that given the capacity constraints on the agents, keeping k replicas is not enough to warrant k-resilience and there might be no possible distribution. The maximum k value for which k-resilience can be achieved depends on the system and especially on agent's capacities. Additionally, the k-resilience characteristic of the repaired system should be restored, as long as there are enough nodes available. Notice that this approach to resilience has an impact on privacy, as computations might be shared with any agent in the system.

We call *Distributed Replica Placement Method* (DRPM) our method for replication, described in Section 4.

3.2 Relocation

Relocation consists in assigning a computation, whose host agent has left the system, to a another agent. As the computation's definition is required to run it, the agents on which a computation could be relocated are the agents holding a replica for that computation. Of course, the repaired system is still subject to the same conditions as for the initial distribution. Thus, the relocation problem aims at selecting the agents in a way that minimizes the communication and hosting costs, while honoring the capacity constraints. Starting the relocation process requires that agents are aware of failures in the system; here we assume that an agent holding a replica for a computation c_i monitors the agent hosting that computation and detects its departure (using *keep alive* messages). Notice that this knowledge is still local and that no entity needs to be aware of the status of all agents in the system.

Our approach, called *DCOP Model for Computation Migration* (DMCM) is exposed in Section 5.

4 REPLICA PLACEMENT

The problem of assigning replicas to hosts could be considered as an optimization problem, close to Definition 2. Ideally, we should optimize replica placement for communication and hosting costs. This would ensure that when agents fail, replicas are available on good candidate agents. However, the search space for this optimization is prohibitively large. In a k-resilient system with n agents, there is $\sum_{0<i\leq k}\binom{n}{i}$ potential failure scenarios (up to k agents out of n can fail simultaneously). With m computations, the number of possible *replica configurations* is $m \cdot \binom{n}{k}$ (for each of the m computations, select k agents to host the replicas). In the end, the problem of optimally distributing the k replicas of each computation on a given set of agents having different costs and capacities can be cast into a quadratic multiple knapsack problem (QMKP) [21], which is NP-hard. Moreover, even assuming we could compute every possible replicas placement, it would still not be obvious which would be better. Indeed, defining the optimality for replica placement is very problem dependent. Thus, given that complexity, we opt for a distributed heuristic approach, described in the next section.

4.1 Distributed Replica Placement Method

We propose here a distributed method, namely DRPM, to determine the hosts of the k replicas of a given computation x_i. The intuition behind DRPM is that the initial distribution is either optimal or of very good quality. Therefore, by placing replicas on closest neighbors, with respect to communication and hosting costs, DRPM ensures that the distribution after repair will maintain a good quality, as in case of failure a computation is relocated to one of the agents holding it's replica. If some privacy is required, one can use infinite hosting costs to exclude some agent as potential host for a replica.

DRPM is a distributed version of iterative lengthening (uniform cost search based on path costs) that finds the k best paths by searching in a graph induced by computations dependencies. It outputs a distribution of k replicas (and the path costs to their hosts) with minimum costs over a set of interconnected agents. If it is impossible to place the k replicas, due to memory constraints, DRPM places as much computations as possible and outputs the best resilience level it could achieve. One hosting agent, called *initiator*, *iteratively* asks each of his lowest-cost neighbors, in increasing cost order, until all replicas are placed. Candidate hosts are considered iteratively in increasing order of cost, which is composed of both communication cost (all along the path between the original computation and its replica) and the hosting cost of the agent hosting the replica.

Let's first define the graph specifying the communication costs which will be developed during the search process:

Definition 4 (route–graph) *Given a computation graph $\langle \mathbf{X}, D\rangle$, the* $\mathbf{route}$*–graph is the edge-weighted graph $\langle \mathbf{A}, E, w\rangle$ where $\mathbf{A}$ is the set of vertices, E is the set of edges with $E = \{(a_m, a_n) | \exists (x_i, x_j) \in D, \text{and } \mu(x_i) = a_m, \mu(x_j) = a_n\}$ and $w : E \to \mathbb{R}^+$ is the weight function $w(a_m, a_n) = \mathbf{route}(m, n)$.*

Contrary to classical approach of routing like OSPF [9], as to take into account both communication and hosting costs in the path costs, the $\mathbf{route}$–graph is extended into a $\mathbf{route}+\mathbf{host}$–graph with extra leaf vertices attached to each agent in the neighboring graph, except the original host of the computation, with an edge weighted using the hosting cost of the agent, as in Figure 2.

Definition 5 (route+host–graph) *Given $\langle \mathbf{A}, E, w\rangle$ a* $\mathbf{route}$*–graph, a computation x_i, and a mapping μ, the* $\mathbf{route}+\mathbf{host}$*–graph is the edge-weighted graph $\langle \mathbf{A}', E', \mathbf{cost}\rangle$ where $\mathbf{A}' = \mathbf{A} \cup \widetilde{\mathbf{A}}$ is the set of vertices where $\widetilde{\mathbf{A}} = \{\tilde{a}_m | a_m \in \mathbf{A}, a_m \neq \mu(x_i)\}$ is a set of extra vertices (one for each element in $\mathbf{A}$ except the host of x_i), $E' = E \cup \{(a_m, \tilde{a}_m) | a_m \in \mathbf{A}\}$ is the set of edges and $\mathbf{cost} : E' \to \mathbb{R}^+$ is the weight function s.t. $\forall a_m, a_n \in \mathbf{A}$, $\mathbf{cost}(a_m, a_n) = w(a_m, a_n)$, $\forall \tilde{a}_m \in \widetilde{\mathbf{A}}$, $\mathbf{cost}(a_m, \tilde{a}_m) = \mathbf{c_{host}}(a_m, x_i)$.*

A $\mathbf{route}+\mathbf{host}$–graph is a search graph, expanded at runtime and explored for a particular computation x_i. Each agent operates as many DRPM as computations to replicate over several $\mathbf{route}+\mathbf{host}$–graph's. For a given $\mathbf{route}+\mathbf{host}$–graph, each agent may encapsulate two vertices (one in $\mathbf{A}$ and its image in $\widetilde{\mathbf{A}}$) and may receive messages concerning their two vertices, and even self-send messages. Additionally, when assessing if an agent can host a replica, we ensure that it only accepts if it has enough capacity to activate any subset of size k of its replicas, using a predicate `can_host?`. Of course this constraint is stronger than what might be actually needed, so, this distribution is not optimal with respect to hosting cost, since one agent reject hosting a computation whilst it may finally have enough memory to host it. Even communication-wise, the algorithm may results on a suboptimal distribution. However, if `can_host?` is provided by an oracle or if memory is not a real constraint, and replica placement only concerns one computation, the distribution would be optimal with respect to communication and hosting costs, since our algorithm implements an iterative lengthening search [17, p.90]. Otherwise, DRPM is not guaranteed to be optimal.

Example 1 *Figure 2 shows a sample* $\mathbf{route}+\mathbf{host}$*–graph with 4 agents (in gray), where a_1 search for hosting computation x_i. For $k = 2$, DRPM places a replica on a_2 (cost of $1+1 = 2$) and another on a_3 (cost of $1+3+1 = 5$) if enough capacity on these two agents, since the minimum cost path to host on a_4 is higher ($1 + 5 = 6$).*

DRPM makes use of two message types (`REQUEST` and `ANSWER`) with the same fields: (i) `current`: path of the request, as a list containing all vertices messages have been passed through from the initiator vertices to the one receiving the current message, (ii) `budget`, `spent`: remaining budget for graph exploration and budget already spent on the current path, (iii) `known`: map assigning cost to already discovered paths to unvisited vertices which bookkeeps the cheapest paths so far, (iv) `visited`: list of already visited vertices, (v) `k`: the remaining number of replicas to host, (vi) x_i: computation that must be replicated. `REQUEST` messages propagate down along the graph from the initiator, and correspond to the development of the graph. `ANSWER` messages trace the solutions (if any) back to the initiator.

At the beginning, the agent requiring a computation replication initializes `known` with the paths to its direct neighbors in the $\mathbf{route}+\mathbf{host}$–graph and sends itself a `REQUEST` message with a

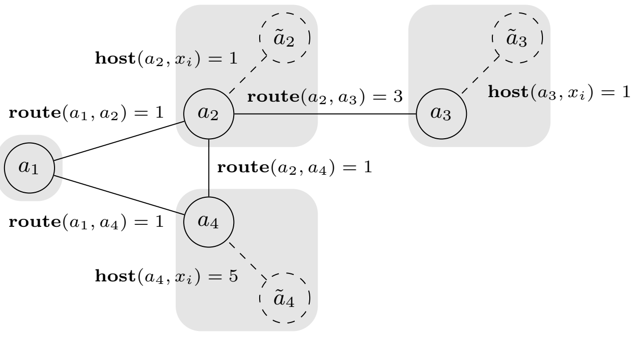

$\mathbf{host}(a_2, x_i) = 1$
$\tilde{a}_2$
$\mathbf{route}(a_2, a_3) = 3$
$\tilde{a}_3$
$\mathbf{host}(a_3, x_i) = 1$
$\mathbf{route}(a_1, a_2) = 1$
a_2
a_3
a_1
$\mathbf{route}(a_2, a_4) = 1$
$\mathbf{route}(a_1, a_4) = 1$
a_4
$\mathbf{host}(a_4, x_i) = 5$
$\tilde{a}_4$

Figure 2: A sample $\mathbf{route}+\mathbf{host}$–graph with 4 agents.

Algorithm 1: Handler for `REQUEST` messages

```
   Data: current, budget, spent, known, visited, k, $x_i$
1  known $\leftarrow$ known $\setminus$ current
2  if me $\notin$ visited then
3      visited $\leftarrow$ visited $\cup\{$me$\}$
4      if can_host?($x_i$) then
5          k $\leftarrow$ k $- 1$
6          add $x_i$ to memory
7          if $k = 0$ then
8              $a_p \leftarrow$ predecessor of me in current
9              send ANSWER(current, budget+cost(me, $a_p$),
                spent-cost(me, $a_p$), known, visited, k,
                $x_i$) to $a_p$
10             return
11 $p \leftarrow \text{argmin}_{e \in \{\text{paths in known starting with current}\}}$ known$[e]$
12 if $p \neq \emptyset$ then
13     $a_n \leftarrow$ successor of me in $p$
14     if cost(me, $a_n$) $\leq$ budget then
15         current $\leftarrow$ current + $a_n$
16         send REQUEST(current, budget-cost(me, $a_n$),
            spent+cost(me, $a_n$), known, visited, k, $x_i$)
            to $a_n$
17         return
18 foreach $a_n \in \{a_m \mid (a_m, \text{me}) \in E', a_m \notin \text{visited}\}$ do
19     if spent + cost(me, $a_n$) <
         $\min_{e \in \{\text{paths in known leading to } a_n\}}$ known$[e]$ then
20         known[current + $a_n$] $\leftarrow$ spent + cost(me, $a_n$)
21 $a_p \leftarrow$ predecessor of me in current
22 send ANSWER(current, budget+cost(me, $a_p$),
    spent-cost(me, $a_p$), known, visited, k, $x_i$) to $a_p$
```

Algorithm 2: Handler for `ANSWER` messages

```
   Data: current, budget, spent, known, visited, k, $x_i$
1  if k $= 0$ then
2      if me is root of current path then
3          terminate with target number of replicas placed
4      else
5          $a_p \leftarrow$ predecessor of me in current
6          send ANSWER(current, budget+cost(me, $a_p$),
            spent-cost(me, $a_p$), known, visited, k, $x_i$)
            to $a_p$
7  else
8      $p \leftarrow \text{argmin}_{e \in \{\text{paths in known starting with current}\}}$ known$[e]$
9      if me is root of current path then
10         if $p \neq \emptyset$ then
11             budget $\leftarrow$ budget + known$[p]$
12             $a_n \leftarrow$ successor of me in $p$
13             current $\leftarrow$ current + $a_n$
14             send REQUEST(current,
                budget-cost(me, $a_n$), cost(me, $a_n$), known,
                visited, k, $x_i$) to $a_n$
15         else
16             terminate with fewer replicas than requested
17     else
18         if $p \neq \emptyset$ then
19             $a_n \leftarrow$ successor of me in $p$
20             if cost(me, $a_n$) $\leq$ budget then
21                 current $\leftarrow$ current + $a_n$
22                 send REQUEST(current,
                    budget-cost(me, $a_n$),
                    spent+cost(me, $a_n$), known, visited,
                    k, $x_i$) to $a_n$
23         $a_p \leftarrow$ predecessor of me in current
24         send ANSWER(current, budget+cost(me, $a_p$),
            spent-cost(me, $a_p$), known, visited, k, $x_i$)
            to $a_p$
```

budget equals to the cheapest known path. Then, agents handle messages as explained in the next paragraph. The protocol ends when all possible replicas have been placed (at most k).

When receiving a `REQUEST` message (see Alg. 1), either the agent can host a replica (line 2), and thus decreases the number of replicas to place, or forwards the request to other agents. In the first case, if all replicas have been placed, the agent answers back to its predecessor with a `ANSWER` message (line 9). When looking for other agents to host replicas, if there exists a minimum cost known path starting with the currently explored path which is reachable with the current budget (line 11), the agent forwards the request to its successor in this path with an updated cost and budget (line 16). If there is no such path, the agent fill out the `known` map of known paths with new paths leading to its neighbors in the **route**+**host**–graph, when they improve the existing known paths, and sends this back via an `ANSWER` message to its predecessor so that it will explore new possibilities (line 22).

When receiving an `ANSWER` message (see Alg. 2), the message can either notify that all replicas have been placed or that there exists at least one replica left to place. In the former case, if the agent is the initiator, it terminates the algorithm, whilst having all the requested replicas placed (line 3), otherwise it forwards the answer back to its predecessor, until it reached the initiator (line 6). In the later case, if the agent is the initiator it increases the budget and send a request to the closest neighbor if any (line 14); otherwise that means that there is no more path to explore and that all replicas cannot be placed, therefore the agent terminates (line 16). If the agent is not the initiator, but there exists some reachable path within current budget, it requests replication to its successor in the best known path, as when handling `REQUEST` messages (line 22). Finally, if there is no such path, it simply forwards the answer to its predecessor in the current path (line 24).

Example 2 *In Figure 2, a_1 initiates and considers two paths $[a_1 \to a_2]$ and $[a_1 \to a_4]$ with same cost, and thus sets a budget equals to 1. a_1 sends a `REQUEST` to a_2, his first successor in the first path. a_2 replies with an `ANSWER` containing two new paths $[a_1 \to a_2 \to \tilde{a}_2]$ of cost 2, and $[a_1 \to a_2 \to a_3]$ of cost 4. Path $[a_1 \to a_2 \to a_2 \to a_4]$ of cost 2 is not considered because the best known path leading to a_4 in `known` costs 1. a_1, upon receipt of the message from a_2, explores the best way by sending a message to a_4, who adds a path to `known`, $[a_1 \to a_4 \to \tilde{a}_4]$ of cost 6 and answers back to a_1. a_1 sends a `REQUEST` to a_2 to explore the best way forward. a_2 can host the replica, because this path has a leaf $\tilde{a}_2$. a_2 can even continue the exploration, because the next path in `known` goes through it and leads to a_3. So a_2 sends a `REQUEST` to a_3, but this time the number of replicas to place is no longer 2 but 1. a_3 adds the path $a_1 \to a_2 \to a_3 \to \tilde{a}_3$, which is the best available at this time, and so answers that it can host the last replica. a_3 sends an `ANSWER` to a_2, which forwards it to a_1. At reception, as all replicas have been placed, a_1 ends the placement.*

Globally, each agent is responsible for placing k replicas of all the computations it currently hosts, and thus executes DRPM once for each of its computations. These multiple DRPM runs can be either sequentially or concurrently executed, but their result depends on message reception order. Note that even when running multiple DRPM concurrently, an agent has only one message queue and handle incoming messages sequentially, which prevents him from accepting replicas that would exceed its capacity.

Theorem 1 *DRPM terminates.*

Proof. For $k = 1$, since costs are additive and monotonic, and paths to unvisited vertices are bookkept, DRPM terminates like classical iterative lengthening, with the minimum cost path or empty path if not enough memory in agents to host x_i. For $k > 1$, it attempts to place each replica sequentially, searching first for the best path (as for $k = 1$), then operates the same process for a second best path, and so on until either (i) k replicas are placed or (ii) there is not enough memory to host the n^{th} replica . Bookkeeping ensures the same path will not be considered twice, and thus consecutive search iterations output different paths with increasing path costs. In case (i), DRPM terminates when k replicas have been placed on the k best hosts; and in case (ii), it terminates when $k' < k$ replicas have been placed, with k' the maximum number of replicas that can be placed. □

Theorem 2 *DRPM requires $\mathcal{O}(b^l)$ messages to terminate, with b the branching factor, $l = d/e$ number of iterations, d the depth of the tree, and $0 < e \leq 1$ the normalized step cost.*

Proof. DRPM's worst case is that the only possible hosts for replica are the last explored ones or there is no possible host. The number of explored nodes is $(l)b + (l-1)b^2 + \ldots + (1)b^l$ which is $\mathcal{O}(b^l)$. It requires twice messages (request and answer), which is still $\mathcal{O}(b^l)$. □

5 DECENTRALIZED REPAIR METHOD

Computations being replicated, we now introduce DMCM, the DCOP Model for Computation Migration, which is implemented by agents, following a failure of up to k agents in the system.

5.1 DCOP Formulation

We note A_d the set of up to k agents that leave the system simultaneously. $X_c = \cup_{a_m \in A_d} \mu(a_m)$ denotes the set of orphaned candidate computations x_i that must be relocated For each of these computations, $A_c^i = \rho(x_i) \backslash A_d$ is the set of candidate agents that could host x_i, i.e. agents holding a replica for x_i. The set of all candidate agents, regardless of computations, is noted $A_c = \cup_{x_i \in X_c} A_c^i$ and X_c^m denotes the set of computations that agent a_m could host. Deciding which agent $a_m \in A_c$ hosts each computation $x_i \in X_c$ can be mapped to an optimization problem similar to the one presented in Section 2, restricted to A_c and X_c: communication and hosting costs should be minimized while honoring the capacity constraints of agents. To ensure each candidate computation is hosted on exactly one agent, we rewrite constraints (1) for each $x_i \in X_c$:

$$\sum_{a_m \in A_c^i} x_i^m = 1 \tag{6}$$

Similarly, capacity constraints (2) can be reformulated as:

$$\sum_{x_i \in X_c^m} \mathbf{w}(x_i) \cdot x_i^m + \sum_{x_j \in \mu^{-1}(a_m) \backslash X_c} \mathbf{w}(x_j) \leq \mathbf{w_{max}}(a_m) \tag{7}$$

The hosting cost objective in (5) can be similarly formulated using one soft constraint for each candidate agent a_m:

$$\sum_{x_i \in X_c^m} \mathbf{c_{host}}(a_m, x_i) \cdot x_i^m \tag{8}$$

Finally, the communication costs in (4) is represented with a set of soft constraints. For an agent a_m, the communication cost incurred by hosting a computation x_i can be formulated as the sum of the cost of the cut edges (x_i, x_j) from the computation graph $\langle \mathbf{X}, D \rangle$, (i.e. where $\mu^{-1}(x_j) \neq a_m$). Let's note N_i the neighbors of x_i in the computation graph. When a neighbor x_n is not a candidate computation (i.e. it might not be moved and $x_m \in N_i \backslash X_c$), the communication cost of the corresponding edge is simply given by $\mathbf{c_{com}}(i, j, m, \mu^{-1}(x_m))$. For neighbors that might be moved, communication cost depends on the candidate agent that is chosen to host it and can be written as $\sum_{a_n \in A_c^j} x_j^n \cdot \mathbf{c_{com}}(i, j, m, n)$. With this we can write the communication cost soft constraint for agent a_m:

$$\sum_{(x_i, x_j) \in X_c^m \times N_i \backslash X_c} x_i^m \cdot \mathbf{c_{com}}(i, j, m, \mu^{-1}(x_j))$$
$$+ \sum_{(x_i, x_j) \in X_c^m \times N_i \cap X_c} x_i^m \cdot \sum_{a_n \in A_c^j} x_j^n \cdot \mathbf{c_{com}}(i, j, m, n)) \tag{9}$$

We can now formulate the repair problem as a DCOP $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mu \rangle$ where $\mathcal{A}$ is the set of candidate agents A_c, $\mathcal{X}$ and $\mathcal{D}$ are the set of decision variables x_i^m and their domain and $\mathcal{C}$ is composed of constraints (6), (7), (8), and (9) applied for each agent $a_m \in A_c$. (6) and (7) result in infinite costs when violated, while (8) and (9) directly define costs to be minimized. The mapping function μ assigns each variable x_i^m to agent a_m.

5.2 Implementing Repair using a DCOP Solver

Now that our repair problem has been expressed as a DCOP, we discuss its resolution using a DCOP solution method. Several solution methods for DCOPs exist, like search algorithms [11, 13] and inference algorithms [14, 22, 6], to cite a few. In brief, using these message passing protocols (synchronous or not), agents coordinate to assign values to their variables.

In our case, we opt for MGM-2, a 2-coordinated variant Maximum Gain Message (MGM) [11], a lightweight, fast and iterative method. In MGM, each agent first assigns random values to its variables and send the information to all its neighbors. Using all neighbors' values, an agent calculates the maximum gain if it changes its value and sends it to all its neighbors. Then, using all neighbors' gains, the agent changes its value if its gain is the largest. This process repeats until a termination condition is met. MGM monotonic property fits very well our needs: once the hard constraints (6) and (7) have been satisfied they will not be broken while optimizing the soft constraints (8) and (9). By comparison, a stochastic algorithm like DSA for example, would not provide this guarantee.

The 2-coordination provided by MGM-2 allows for required coordination: to move a computation from agent a_m to a_p, the binary variable x_i^m must take 0 as a value, while *simultaneously*, x_i^p must switch from 0 to 1. We argue that, as moving a computation involves a coordinated decision of exactly two agents, MGM-2 always eventually find a solution satisfying all hard constraints, if such solution exist. Indeed, moving a computation from an agent where it violates the capacity constraint to an agent with enough capacity will always represent the maximum gain and be the best offer. However, as offerers and receivers are selected at random, MGM-2 takes an indeterminate number of cycles to find a valid solution (less than 15 in our experiment). However, MGM-2 is still a local search algorithm and may yield a sub-optimal solution for the soft constraints.

By applying MGM-2 on DRPM[DMCM], we obtain a full solution for k-resilience called DRPM[MGM-2].

5.3 Focus on Distributed Constraint Reasoning

The general framework we propose can be applied to several settings, especially in distributed constraint reasoning and optimization. Such decision problems can be represented as graphs, like constraint graphs (CG) or factor graphs (FG), where computations can be decision variables or constraints. Solving such problems requires operating distributed algorithms over the graph by exchanging messages, which suits perfectly DRPM[MGM-2]. But, all distributed solution methods may not suit well, especially those which are sensitive to message or information loss. Since agents may disappear, algorithms based on question-answer messages (e.g. DPOP [14], AFB [8]), might be stuck in a deadlock or an inconsistent state. Moreover, algorithms hardly relying on information acquired and stored by agents might not be able to restart after a reparation (e.g. ADOPT [13]). Thus, suitable algorithms to be equipped with DRPM[MGM-2] are asynchronous algorithms with small memory usage and robust to message loss. Here, we identify two algorithms as relevant candidates: Max-Sum and A-DSA.

Max-Sum is an inference-based algorithm based on belief propagation [6], operating on factor graphs by performing a marginalization process of the cost functions, and optimizing the costs for each given variable. The value assignments take into account their impact on the marginalized cost function. If a computation (variable or factor) disappears, it might quickly restore its past state from messages received from its neighboring agents. Max-Sum can be implemented as an asynchronous algorithm (A-MaxSum).

A-DSA [7] is an asynchronous version of the Distributed Stochastic Algorithm (DSA) [25], a local search DCOP algorithm. At start, agents assign a random value to their variable(s) and send this information to their neighbors. Thus each agent can evaluate if the local quality of its own partial assignment could be improved by selecting a new value, in which case it decides randomly, with an probability p, to select the corresponding value and send its updated state to its neighbors. A-DSA is asynchronous, each agent evaluates *periodically* if it could improve its partial assignment.

6 EXPERIMENTAL EVALUATIONS

We analyze here the quality of repaired distributions using DRPM[MGM-2], and the impact of repair on the performance of computations operated of the set of agents. We choose to illustrate our DCOP-based repair framework applied to two DCOP solution methods (A-MaxSum and A-DSA) where CGs and FGs are computation graphs to be deployed and repaired at runtime.

6.1 Experimental Setup

We run the experiments using the multi-threaded DCOP library pyDCOP [16]. We generate instances composed each of: a problem definition, a topology (the infrastructure), and a disturbance scenario.

We study two different types of DCOPs: (i) random graph coloring problem and (ii) scale free graph coloring. Random graph coloring problem are generated by creating a random graph with density $p = 0.3$. For scale free graph coloring problem, we generate a graph using the Barabasi-Albert model [1] (starting for a 2-node connected graph) . In both cases, each node is mapped to a variable and each edge is mapped to a binary constraint whose costs are generated by sampling from the uniform discrete distribution $U[0-9]$.

Depending on the chosen solution method (A-DSA or A-MaxSum), the DCOP is encoded into either a CG (A-DSA) or a FG (for A-MaxSum). For a same problem $\langle \mathbf{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mu \rangle$, this results in placing either $|\mathbf{V}| = |\mathcal{X}|$ computations for a CG or $|\mathbf{V}| = |\mathcal{X}| + |\mathcal{C}|$ computations for a FG, since constraints are mapped to computations in FGs.

Once the computation graph is generated, we generate two different multi-agent infrastructures: uniform and problem-dependent. An infrastructure is made of $|\mathbf{A}|$ agents, each holding one decision variable ($|\mathbf{A}| = |\mathcal{X}|$), and is defined by $\mathbf{c_{host}}$, $\mathbf{route}$, $\mathbf{w_{max}}$, $\mathbf{w}$ and $\mathbf{msg}$ as follows. The uniform infrastructure considers systems where communication costs are uniform: $\forall a_m, a_n, \mathbf{route}(a_m, a_n) = 1$. In the problem-dependent case, route costs $\mathbf{route}(a_m, a_n)$ are defined in a way that respects the structure of the computation graph: agents with many neighbors have a low communication while agents few neighbors have an higher communication cost, as in many physical infrastructures like IoT. More precisely, $\mathbf{route}(a_m, a_n) = \frac{1+||N(a_m)|-|N(a_n)||}{|N(a_m)|+|N(a_n)|}$ where $|N(a_i)|$ is the number of neighbors of a_i in the computation graph In all cases, we set (i) hosting costs $\mathbf{c_{host}}(a_m, x_j) = 0$ if the computation x_j is initially hosted by agent a_m, $\mathbf{c_{host}}(a_m, x_j) = 10$ otherwise; (ii) the capacity of each agent depends on the weight of its decision variable and is set to a large value, to ensure that all replicas can be hosted and k-resiliency is possible, even after several repairs: $\mathbf{w_{max}}(a_i) = 100 * \mathbf{w}(x_i)$; (iii) finally, $\mathbf{w}$ and $\mathbf{msg}$ depends on the solution method used. For A-MaxSum, $\mathbf{msg}(i, j) = |D_j| = 10$ and the weight of variable and factors computations is respectively proportional to the size of the variable's domain and the sum of the size of the linked variable's domains. For A-DSA, $\mathbf{msg}(i, j) = 1$ and $\mathbf{w}(x_i) = |N(a_m)|$.

Initially, each variable is assigned to an agent, and in case of FGs, factors are placed using a greedy heuristic providing near-to-optimal solutions to the ILP in Section 2. We use $\omega = 0.5$ (hosting costs and communication costs are equally considered).

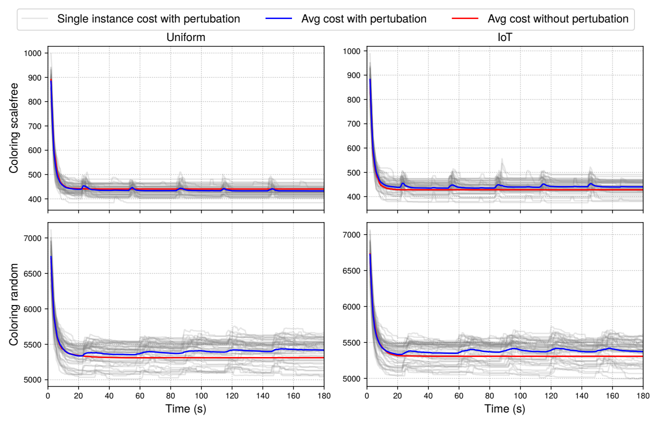
We generate disturbance scenarios as sequences of perturbation events happening every 30 seconds, starting at $t = 20s$. At each such event, k randomly chosen agents disappear as to analyze the impact on DCOP solution methods, and observe the k-resilience of our system. We generate 20 instances (infrastructure and problem), and run each instance 5 times. We also solve the same problems (5 runs for each of the 20 instances) without any disturbance, as to assess the impact of repair methods on the quality of the solution returned by A-DSA or A-MaxSum. Results are averaged over all instances and experiments are performed on an Intel core i7 CPU with 16GB RAM.

6.2 Impact of Repairing A-DSA Operations

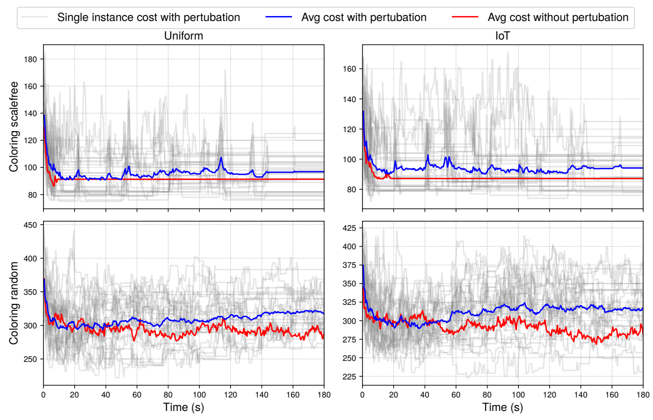
Let us look at the runtime state of the current A-DSA with and without disturbance, on the different infrastructures and problems. We generate problems with $|\mathbf{A}| = |\mathcal{X}| = 100$, and use $k = 3$.

Figure 3a shows the cost of the solution found by A-DSA over time. The cost of each of the 100 runs is displayed in transparent grey, the overall shapes illustrates the fact that the system's behavior is consistent across the various instances. We can see that the solutions on the disturbed system degrade when agents are removed, but quickly improve again when the system recovers. Here, the replicas that are activated by the repair process, as opposed to the computation that were hosted on removed agents, do not need accumulated knowledge to recover a consistent state, thank to message passing with neighbors. In A-DSA, computations are stateless, as required by our approach of k-resilience (see Section 3): they gather new information about costs from their neighbors at each message exchange.

The recovering period is shorter on scale free models, while it takes more time on random graphs. Indeed, our graph coloring prob-



(a) A-DSA



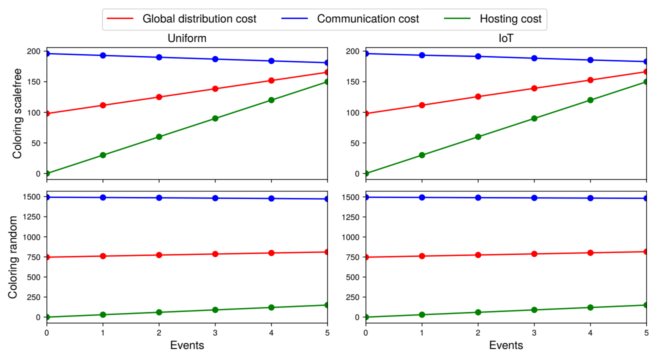
(b) A-MaxSum

Figure 3: Solution cost for A-DSA (3a) and A-MaxSum (3b) at runtime, w/ (blue) and wo/ (red) perturbation, on uniform (left) and problem-dependent (right) infrastructure, when solving scale free (top) and random (bottom) graph coloring, using DRPM[MGM-2].

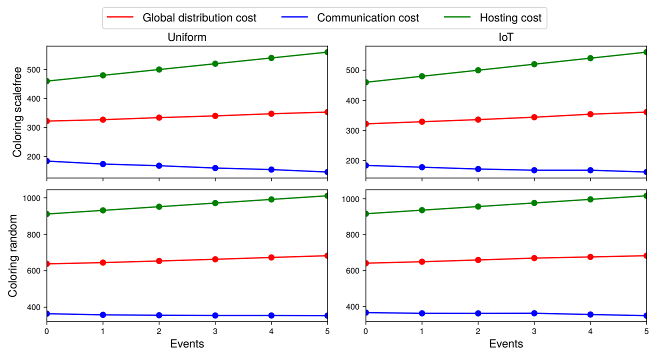
lems are more dense and more difficult to solve, and repair requires more messages and time. For the same duration, agents solving and repairing random graph systems must handle more repair-specific messages (MGM-2 and DRPM) than agents solving scale free graph systems, thus they handle less A-DSA messages to improve the solution cost. This is the same reason why the overall cost is higher in problem-dependent infrastructure: the replica activation and placement processes require more messages. We evaluated the average time to repair a distribution at less than 3 seconds. Another point of attention is the fact that, in some settings, the average cost with perturbation is better than the average cost without perturbation. In fact, removing some computations and relocating them to other agents, and thus forgetting some information about the past neighborhood, might extract A-DSA from some local optima. We can also observe some time lag for some instances to repair the system, mainly due to accumulated message processing latency.

6.3 Impact of Repairing A-MaxSum Operations

We look here at the runtime state of the current A-MaxSum, to analyze how DRPM[MGM-2] changes the operation of the running process, on the different infrastructures and problems. We consider smaller problems, since A-MaxSum operates on FGs which requires



(a) A-DSA



(b) A-MaxSum

Figure 4: Cost of the distributions of computation graphs on which A-DSA (4a) or A-MaxSum (4b) operates, after each event, on uniform (left) and problem-dependent (right) infrastructure, when solving scale free (top) and random (bottom) graph coloring problems, using DRPM[MGM-2] to repair.

more computations to distribute than CGs used by A-DSA. We consider $|\mathbf{A}| = |\mathcal{X}| = 25$, and $k = 2$. On random graph with density of 0.3, this requires on average $25 + 0.3\frac{25\times24}{2} = 125$ computations to manage. In Figure 3b we can see that solution costs on the disturbed system degrade when agents are removed, but improve again when the system recovers, as for A-DSA. However, A-MaxSum operation on very cyclic problems like random coloring is known to be very noisy, even using a high damping factor (here 0.8 as in [4]). Moreover, belief propagation algorithms like A-MaxSum, computations are not stateless: they accumulate information about constraints and preferences from their neighbors. When activating a replica, the new active computation start afresh and an indeterminate number of message rounds are needed to restore information. So, A-MaxSum is more impacted by the perturbations and repair procedure than A-DSA.

6.4 Quality of Repaired Distributions

To evaluate the quality of the repaired distributions of computations, we measure the degradation of the distribution all along the system lifetime. At each event, we assess the cost of the current distribution of the constraint graphs (for A-DSA) and the factor graphs (for A-MaxSum) using equations 4 and 5, against the initial distribution cost (which is optimal, but cannot be computed at runtime). Figure 4 shows the distribution costs for the 100 runs. As the global distribution cost is made of communication and hosting costs, we also plot these two costs independently. In every case, the hosting cost logically increases by $10 \cdot k$ at each perturbation event, as k computations are moved from their initial agent to another (where the hosting cost is 10). On scale free models, hosting costs and communication costs have the same order of magnitude. But, for random graphs, higher density implies that there are more edges in the graph and as a consequence the overall communication cost is higher. In general, the communication costs decrease at each repair. Here, all agents are homogeneous and computations are necessarily moved to more costly agents, communication-wise (there is no agent less expensive or equivalent to the missing ones), and thus communication costs increase incrementally. In other cases computations can move to a less expensive agent, and thus communication costs decrease.

7 CONCLUSIONS

We investigated the resilient distribution of computations over a dynamic set of physical agents, and two distributed algorithms have been devised: (i) a replica placement protocol (DRPM) and (ii) a repair protocol (DRPM[MGM-2]) relying on replicas placed by DRPM and based on DCOP solution method MGM-2. Our contributions have been evaluated experimentally through operation of A-MaxSum and A-DSA on dynamic systems where agents disappear during the optimisation process. On the different settings we investigate, operating these algorithms is not much impacted by our repair method DRPM[MGM-2], and the systems continue providing solutions, whilst missing agents, which demonstrates the resilience of these systems. Since our approach is based on the replication of computations, using problem encoding requiring less computations (choosing constraint instead of factor graphs) is a better choice. The complexity of the repair process, encoded as a DCOP itself, strongly depends on the number of computations and the density of the infrastructure. Moreover, on our experiments, A-MaxSum operation is much more impacted by agent removals and repairing than A-DSA which is very robust to such dynamics.

This paper raised promising results about resilience in operating distributed optimisation processes. We only focused on the worst scenario with agent removals only.We will investigate less stressing scenarios, coming from a broader scope of graph-based computations, like high-performance computing or virtual network functions [15], where agents may be added to replace disappeared ones. Moreover, we proposed to use MGM-2 as the core reparation algorithm, resulting in DRPM[MGM-2] method, but, other lightweight DCOP solution methods might be considered or even designed to the particular case of constraint graph or factor graph reparation. Finally, approaches for preserving information disclosure while ensuring system resilience, and the resulting trade-off between resilience and privacy will be investigated in future research.

REFERENCES

[1] A. Barabási, 'Emergence of scaling in random networks', *Science*, **286**(5439), 509–512, (1999).

[2] *Graph Partitioning*, eds., Charles-Edmond Bichot and Patrick Siarry, John Wiley & Sons, Ltd, 2013.

[3] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, 'Fog computing and its role in the internet of things', in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pp. 13–16, New York, NY, USA, (2012). ACM.

[4] L. Cohen and R. Zivan, 'Max-sum revisited: The real power of damping', in *Autonomous Agents and Multiagent Systems*, eds., Gita Sukthankar and Juan A. Rodriguez-Aguilar, pp. 111–124, Cham, (2017). Springer International Publishing.

[5] Neng Fan and Panos M. Pardalos, 'Linear and quadratic programming approaches for the general graph partitioning problem', *Journal of Global Optimization*, **48**(1), 57–71, (2010).

[6] A. Farinelli, A. Rogers, A. Petcu, and N. R. Jennings, 'Decentralised coordination of low-power embedded devices using the max-sum algorithm', in *International Conference on Autonomous Agents and Multiagent Systems (AAMAS'08)*, pp. 639–646, (2008).

[7] S. Fitzpatrick and L. Meertens, 'Distributed coordination through anarchic optimization', in *Distributed Sensor Networks*, eds., V. Lesser, C. L. Ortiz, and M. Tambe, volume 9, 257–295, Springer US, (2003).

[8] A. Gershman, A. Meisels, and R. Zivan, 'Asynchronous forward bounding for distributed cops', *J. Artif. Int. Res.*, **34**(1), 61–88, (2009).

[9] IETF. Ospf version 2. https://tools.ietf.org/html/rfc2328, 1998.

[10] M. Khan, L. Tran-Thanh, W. Yeoh, and N. R. Jennings, 'A Near-Optimal Node-to-Agent Mapping Heuristic for GDL-Based DCOP Algorithms in Multi-Agent Systems', in *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, pp. 1613–1621, (2018).

[11] R.T. Maheswaran, J.P. Pearce, and M. Tambe, 'Distributed algorithms for dcop: A graphical-game-based approach', in *Proceedings of the 17th International Conference on Parallel and Distributed Computing Systems (PDCS), San Francisco, CA*, pp. 432–439, (2004).

[12] G. Malewicz, M. H. Austern, A. J.C Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, 'Pregel: A system for large-scale graph processing', in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pp. 135–146. ACM, (2010).

[13] P.J. Modi, W. Shen, M. Tambe, and M. Yokoo, 'ADOPT: Asynchronous distributed constraint optimization with quality guarantees.', *Artificial Intelligence Journal*, (2005).

[14] A. Petcu and B. Faltings, 'A scalable method for multiagent constraint optimization', in *International Joint Conference on Artificial Intelligence (IJCAI'05)*, pp. 266–271, (2005).

[15] Q. Pham, K. Singh, A. Bradai, G. Picard, and R. Riggio, 'Single and multi-domain adaptive allocation algorithms for vnf forwarding graph embedding', *IEEE Transactions on Network and Service Management*, 1–1, (2018).

[16] pyDcop library. https://github.com/Orange-OpenSource/pyDcop.

[17] S. Russel and P. Norvig, *Artificial Intelligence: a Modern Approach*, Prentice-Hall, 3rd edn., 2009.

[18] P. Rust, G. Picard, and F. Ramparany, 'Using message-passing DCOP algorithms to solve energy-efficient smart environment configuration problems', in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI-16)*, ed., S. Kambhampati, pp. 468–474. AAAI Press, (2016).

[19] P. Rust, G. Picard, and F. Ramparany, 'On the deployment of factor graph elements to operate max-sum in dynamic ambient environments', in *8th International Workshop on Optimisation in Multi-Agent Systems (OPTMAS 2017, in conjunction with AAMAS 2017)*, (2017).

[20] P. Rust, G. Picard, and F. Ramparany, 'Installing Resilience in Distributed Constraint Optimization Operated by Physical Multi-Agent Systems', in *International Conference on Autonomous Agents and Multiagent Systems*, p. 3, (2019). http://www.ifaamas.org/Proceedings/aamas2019/pdfs/p2177.pdf.

[21] T. Saraç and A. Sipahioglu, 'Generalized quadratic multiple knapsack problem and two solution approaches', *Computers & Operations Research*, **43**(Supplement C), 78 – 89, (2014).

[22] M. Vinyals, J. A. Rodriguez-Aguilar, and J. Cerquides, 'Constructing a unifying theory of dynamic programming dcop algorithms via the generalized distributive law', *Autonomous Agents and Multi-Agent Systems*, **22**(3), 439–464, (2010).

[23] R. Wattenhofer. Principles of Distributed Computing, 2015.

[24] O. Wolfson and A. Milo, 'The multicast policy and its relationship to replicated data placement', *ACM Trans. Database Syst.*, **16**(1), 181–205, (March 1991).

[25] 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), 55–87, (2005).