Distributing reasoning on WoT edge architectures

Lina Nachabe, Alexandre Bento, Kamal Singh, Lionel Médini, Frederique Laforest

Distributing reasoning on WoT edge architectures

Lina Nachabe[1], Alexandre Bento[2], Kamal Singh[4], Lionel Médini[3], Frédérique Laforest[2]
[1]*Mines Saint-Étienne, Institut Henri Fayol, F-42023 Saint-Étienne France*
[2]*Univ Lyon, INSA Lyon, CNRS, UCBL, LIRIS, UMR5205, F-69621 Villeurbanne, France*
[3]*Univ Lyon, UCBL, CNRS, INSA Lyon, LIRIS, UMR5205, F-69622 Villeurbanne, France*
[4]*Univ Lyon, Univ Jean Monnet, CNRS, LaHC, UMR5516, F-42000 Saint-Étienne, France*
Email: lina.nachabe@emse.fr[1], firstname.lastname@liris.cnrs.fr[2,3], kamal.singh@univ-st-etienne.fr[4]

***Abstract*—Enabling and automating interoperability in the Semantic Web of Things involves complex semantic reasoning tools to process knowledge graphs. To improve performance and energy efficiency, such tools should be deployed as close as possible to the devices, ideally on any available edge node. However, edge nodes often lack sufficient resources, especially memory. We propose a method to distribute reasoning in this context. We then evaluate three algorithms to plan the distribution over a network of heterogeneous nodes. These algorithms take into account architectural constraints such as the position of sensors and actuators and the available resources on each node, while minimizing costly data exchanges on the network.**

I. Introduction

Typical Internet of Things (IoT) infrastructures rely on sensors and actuators to measure and influence physical phenomena, but also on cloud servers to collect and process data [1]. Such centralized infrastructures can cause latency and bandwidth consumption, as well as bottlenecks and single points of failure [2]. The edge computing approach proposes to distribute data collection and processing on IoT nodes close to sensors and actuators [3]. Such IoT nodes can be the sensors and actuators themselves, but also intermediate network equipment. Today, some IoT nodes offer storage and processing, but still have limited resources in terms of memory, computing power, and battery capacity [4].

The Web of Things (WoT) extends the IoT by leveraging web protocols and standards, interconnection of IoT infrastructures and even open application marketplaces. The integration of semantic web technologies in the WoT unifies device and data descriptions with ontologies, paving the way to interoperability through easy data access, sharing and integration [5], [6]. Semantic technologies allow a particular form of data processing called *semantic reasoning*, that allows to logically infer additional knowledge from rules and knowledge. Hence, the *Semantic Web of Things* (SWoT) offers the perspective of semantic interoperability through producing, consuming, exchanging and reasoning about knowledge graphs.

To take full advantage of the combination of distributed edge infrastructures and SWoT, the reasoning process should be distributed on edge nodes. However, the large majority of existing reasoners are too expensive, in terms of memory and processing, to be directly embedded on resource-constrained nodes. Among the few proposals, the low-memory embedded reasoner LiRoT [7] has been designed to save device resources and validated on Arduino Due[1] and ESP32[2] devices. Still, one of the remaining fundamental challenges is how to decompose and distribute reasoning tasks over a set of heterogeneous nodes with respect to their limited computational and memory capabilities [8].

This paper proposes a method to distribute reasoning tasks among nodes of an edge IoT infrastructure. This is done by generating a distribution plan which distributes graphs onto nodes, taking into consideration the semantic dependencies between graphs, the available memory on each node, as well as an estimation of the memory needed to produce each graph. Our contributions are summarised as follows:

- We introduce a formal model of reasoning distribution across edge WoT nodes.
- We propose an approach that generates a plan for distributed reasoning deployment over multiple nodes, while both (i) respecting memory consumption constraints while deploying reasoning algorithms, and (ii) attempting to minimize data exchanges on the network. Given the NP-hard nature of the problem, we propose three polynomial-time algorithms.
- We provide a performance comparison of our proposed algorithms and investigate the influence of distribution on data exchange efficiency.

Our paper is divided as follows: Section 2 presents a literature overview of reasoning distribution approaches. Section 3 provides an example. Section 4 states the problem as a mathematical model. Section 5 approximates the memory required to process reasoning tasks on each IoT node. Section 6 presents our algorithms for distributing graphs among nodes and Section 7 evaluates them.

II. State of the art

A. Distributing reasoning

Different kinds of semantic reasoning algorithms exist, the most easily treatable being rule-based ones [9]. To guaranty decidability [10], DL-safe rules are rules whose consequences only contain terms and individuals that already exist in the ontology, so that reasoning does not produce new individuals [11]. But even then, the reasoning process complexity keeps polynomial in worst case and a trade-off must be found between processing time and memory space, especially for constrained devices.

[1]https://store.arduino.cc/products/arduino-due
[2]https://www.espressif.com/en/products/socs/esp32

Distributing reasoning on edge nodes can reduce the overall latency by both allowing to perform different reasoning tasks in parallel and reducing the distance traveled by messages [12]. Although some authors propose distributed reasoning for IoT systems [1], [13]–[16], they do not take into consideration the limited capabilities of constrained edge nodes.
Providing the same ruleset to all nodes and distributing data across nodes both frees from necessitating a hierarchical topology and allows for dynamic (re)configuration of the distribution strategy when needed. Distributing a reasoning process then consists in producing graphs on different nodes and exchanging these graphs among nodes.

In our experiments, we equip each node with the same reasoning configuration: the Lightweight Reasoner on Things (LiRoT) [7] that enhances the classical RETE algorithm [17] so that it can be deployed on constrained objects, with a subset of RDFS rules [3] [18] and comparison rules[4].

B. Distribution optimization approaches

Distributing computations on a set of nodes can be viewed as a harder version of problems such as graph partitioning [19] or "Generalized Assignment Problem" (GAP) [20]. GAP is NP-Hard and can generally be formulated as an integer linear programming problem, where the objective function and constraints can be expressed as linear functions of a set of decision variables [21]. However, unlike GAP, our problem adds additional constraints, due to the fact that there exist dependencies among graphs. Therefore reasoning processes cannot be assigned to nodes independently. On top of that, the memory consumption is a non-linear function of the input triples. Different approaches exist for solving non-linear problems. One approach is to divide the non-linear function into several linear sections (piece wise linearization) [22]. We use this approach to propose a distribution of the reasoning process, while respecting the available memory size on each node.

III. Running example

We illustrate the rest of the paper with the example of a classroom equipped with an edge architecture formed of four ESP32 microcontroller-based nodes: N_1 is placed on the door and equipped with a temperature sensor, N_2 is placed on the board and equipped with a CO2 level sensor, a presence sensor and a luminosity sensor, N_3 is connected to a window, and equipped with a sensor that measures the outdoor temperature and an actuator, and N_4 is connected to the room heater and to a presence detector.

Two goals are pursued using this architecture:

- open or close the window to control both the room temperature and CO2 level, when people are present in the room
- turn on the heater to control the room temperature when the room temperature is lower than the comfort temperature but higher than outside, when people are present in the room, and off otherwise

The sensors observe their environment and provide values accordingly [5] . For example, let the temperature measured be 8.5°C and the CO2 level 1050 parts per million (ppm), the proximity sensor return its max value (meaning that it did not detect any obstacle) and the luminosity sensor detects a low value of 10 (max value being 255). Such observations can be expressed as RDF triples such as:

```
:TemperatureSensor :hasValue "8.5"^^xsd :decimal .
:CO2Sensor :hasValue "1050"^^xsd :integer .
:ProximitySensor :hasValue "65535"^^xsd :integer .
:LuminositySensor :hasValue "10"^^xsd :integer .
```

A `thermal comfort` property is inferred based on predefined thresholds on temperature values: `TC_Cold` under 16 Celsius degrees, `TC_Hot` over 30 degrees, and `TC_Medium` between the two. The following triple is inferred:

```
:ThermalComfort :hasResult :TC_Cold .
```

An `air quality` property is inferred based on CO2 level thresholds: it is considered `AQ_Good` if the level is below 960 ppm, `AQ_Bad` above 1760 ppm, and `AQ_Average` otherwise. Similarly, a following triple is deduced:

```
:AirQuality :hasResult :AQ_Average .
```

From the thermal comfort and air quality properties, as well as external data (e.g. from a weather API), the reasoning process also respectively infers the appropriate window status (open/close) and heater status (on/off), to be translated into commands to the window (resp. heater) actuators:

```
:Window :hasStatus :WA_Closed .
:Heater :hasStatus :HA_On .
```

IV. Problem formalization

IoT network. We consider an architecture where each node is a constrained device that manages a set of sensors and/or actuators and hosts a network interface and a reasoner[6]. All nodes embed the same reasoner and the same set of rules.
Let $\mathbb{N} = \{N_1, N_2, \cdots, N_n\}$ be the set of n nodes in an IoT edge network where each node is connected to all others. Let the memory capacity of node N_i be M_i and $\mathbb{M} = \{M_1, \cdots, M_n\}$ the set of memory capacities in the network.

Graphs. Let $\mathbb{G} = \{G_1, G_2, \cdots, G_g\}$ be the set of the considered g graphs. Each graph G_j is characterized by its

[3]https://www.w3.org/TR/rdf12-schema/

[4]Using predicates such as $greaterThan$ and $lessThan$

[5]Sensor observations are actually expressed according to the SSN ontology, which is way more detailed than the content of this example. We herein only provide simplified but sufficient RDF data for the reader to understand the reasoning process. They should bear in mind however that the number of triples required to perform some reasoning tasks may be several orders of magnitude higher than what is presented in this example.

[6]As per our experimentation, we use the LiRoT reasoner [7].

size S_j and $\mathbb{S} = \{S_1, S_2, \cdots, S_g\}$. By extension, we define the sizes of the union of two graphs and of a set of graphs $\Gamma \subset \mathbb{G}$ as the sum of the sizes of the distinct triples in these graphs: $S_{G_1 \cup G_2} = S_{G_1} + S_{G_2} - S_{G_1 \cap G_2}$, and:

$$S_\Gamma = \sum_j S_j - \sum_{j,k}^{j \neq k} S_{G_j \cap G_k}, \forall j \mid G_j \in \Gamma, \forall k \mid G_k \in \Gamma \quad (1)$$

We define three disjoint sets of graphs:

1) *Input graphs*: $\mathbb{G}_{input}$ is the set of graphs built by the lifting of sensors raw data, user input or APIs. For example, an input graph describes the observation of the temperature at 8.5. Each input graph is only produced once, by the node hosting the considered data source.
2) *Intermediate graphs*: $\mathbb{G}_{inter}$ is the set of graphs produced by reasoners after applying rules on other graphs, and aimed at being consumed to produce other graphs. For example, the graph describing the thermal comfort property of a room is an intermediate graph produced from a temperature graph and used to produce actuation decisions. Each intermediate graph can be produced by any node possessing sufficient resources and having produced/received its antecedent graphs.
3) *Final graphs*: $\mathbb{G}_{final}$ is the set of graphs produced by a reasoner to describe a final goal, such as "open the window". One and only one final graph is produced by a reasoning task chain. As for intermediate graphs, they can be produced by any node, but final graphs are consumed on a definite node: the node hosting the targeted actuator.

$$\mathbb{G} = \{\mathbb{G}_{input} \cup \mathbb{G}_{inter} \cup \mathbb{G}_{final}\} \quad (2)$$

The Input Graph Production matrix P of size $n * g$ describes on which nodes input graphs are actually produced, based on the locations of the sensors. For a node i and a graph G_j:

$$P_{i,j} = \begin{cases} 1 & \text{if } G_j \in \mathbb{G}_{input} \text{ and} \\ & \text{node } i \text{ hosts the sensor that produces } G_j \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

The transposed graph production matrix of the running example is:

$$P^T = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (4)$$

Graph dependency. Reasoners consume graphs to infer new ones. A graph G_j depends on another graph G_h if and only if there exists a triple of G_j that can only be produced by applying a rule and a triple of G_h validates one of the premises of this rule. In this case, G_h is called an *antecedent* of G_j. The set of antecedents of G_j is noted A_j. For later convenience, we also denote $A_j^+ = G_j \cup A_j$.
Let D be a graph dependency matrix of size $g * g$ defined as:

$$D_{j,h} = \begin{cases} 1 & \text{if } G_h \in A_j \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

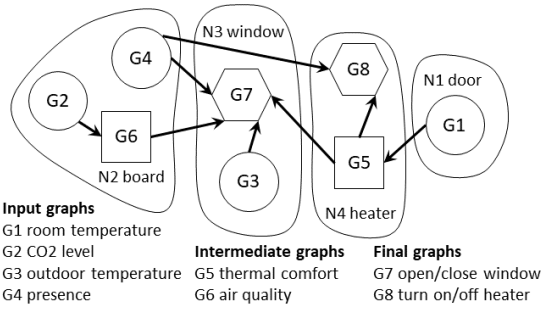

Fig. 1. Possible distributed reasoning workflow for the running example.

The dependency matrix D of the running example is:

$$D = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \end{bmatrix} \quad (6)$$

Reasoning Workflow. A reasoning workflow is a deterministic plan to produce a final graph from a set of input graphs. It is composed of several stages, the first one being applied to input graphs, and the last producing the final graph. At each stage except the last, multiple reasoning tasks may be conducted in parallel on distinct nodes. We herein assume that all graphs are produced at least once during the reasoning workflow.

V. Memory cost for reasoning on an IoT node

In this section, we estimate both the amount of working memory required by a reasoner to produce an intermediate or final graph, and the number of graphs that can be produced on a given node. To do so, we consider a common node setup composed of software performing storage and reasoning, sensing, network communication, etc. The memory required for running this setup is a constant that can be measured by starting a node without loading any graph in it, and is noted M_{setup}.

The amount of working memory required to produce graph G_j is defined as the sum of:

- the memory size of G_j and of the union of all its antecedent graphs: in worst case, if all antecedents are disjoint, it equals $S_{A_h^+} = S_j + \sum_h S_h, \forall h \mid G_h \in A_j$
- the memory required for reasoning over these antecedent graphs: it consists of the variables and data structures used to perform reasoning and depends on the reasoning algorithm[7], on the numbers of distinct conditions in the ruleset (denoted R_{cond}, constant), and on the number of triples in the graphs. Thus, by saturating all conditions for a RETE network with all triples of the antecedents,

[7]For the sake of simplicity, we herein base our model on the regular RETE algorithm, even though the algorithm implemented in the LiRoT reasoner provides several optimizations that reduce this cost.

removing the matches of identical triples and merging permutations, the worst case scenario leads to the order of $\sum_{R_{cond}}({}^{S_{A_j}}C_{R_{cond}}) \cdot t$, where t is the size of a pointer to a triple.

If multiple reasoning tasks are to be conducted on the same node, M_{setup} should only be counted once, as well as each antecedent graph. The total amount of memory consumption is of the order:

$$M_{setup} + S_{\bigcup_j A_j^+} + \sum_{R_{cond}} ({}^{S_{A_j}}C_{R_{cond}}) \cdot t \tag{7}$$

As expected, in worst case, RETE produces an exponential space complexity wrt. the number of triples. Recall that the number of rules is considered constant in our case. We consider an approximation of formula 7 as a linear function of the number of explicit triples, of the form $f_{mem} : S_{A_j} \mapsto K.S_{A_j} + M_{setup}$. We use the piece-wise linearization technique [22] where different non-linear regions are approximated by different linear pieces. For our WoT use case which involves a low number of triples, we assume a linear curve for the region. This is supported by different experiments conducted in [7], [23] which observed approximately linear curves for memory consumption vs. number of triples, up to 1000 triples and with the RDFS ruleset. We rely on an empirical method for determining K for the first linear piece.

As explained earlier, the number of RDFS individuals is fixed, and so is the maximum number of triples that can be produced for a given application. Hence, it is possible to run a set of reasoning tests that will produce all possible results. Measuring the maximum amount of consumed memory M_{max} will provide a reference value for K, as the ratio M_{max} divided by number of input triples $S_{\bigcup_j A_j}, \forall j \mid G_j \in \mathbb{G}$ [8]. We can then approximate the first linear piece with:

$$S_{A_j} \leq \frac{M_{max}}{S_{A_{\mathbb{G}}}}.S_{A_j} + M_{setup} \tag{8}$$

Hence, for any set of graphs Γ, a node i having a maximum amount of memory M_i can produce all elements of Γ as long as:

$$S_{\bigcup_j A_j} \leq S_{A_\Gamma \frac{M_i - M_{setup}}{M_{max}}}, \forall j \mid G_j \in \Gamma \tag{9}$$

We substitute the constant $K = \frac{M_{max}}{S_{A_\Gamma}}$:

$$M_i \geq K.S_{\bigcup_j A_j} + M_{setup},\ \forall j \mid G_j \in \Gamma \tag{10}$$

VI. Approach for distributed reasoning

This section presents several approaches to determine, for each reasoning process and each graph, which node will produce it, while minimizing the overall amount of data exchanged on the network, and according to the inputs defined in the previous sections.

[8]This can be done either by realizing all computations on the same - powerful enough - machine, or by running each reasoning process independently and taking the maximum value of this ratio [7].

A. Algorithm characteristics

1) Decision variables: We introduce two sets of Boolean decision variables $\mathbb{X}, \mathbb{Y}$, defined as:

$$x_{i,j} = \begin{cases} 1, & \text{if } N_i \text{ should obtain } G_j \text{ over the network} \\ 0, & \text{otherwise} \end{cases} \tag{11}$$

$$y_{i,j} = \begin{cases} 1, & \text{if } N_i \text{ should produce } G_j \\ 0, & \text{otherwise} \end{cases} \tag{12}$$

2) Objective function: As (wireless) communications are energy-intensive on small battery-powered devices, we aim at taking into consideration and minimizing the amount of data exchanged among nodes.

Let $Cost$ be total size of the data exchanged among nodes throughout a whole reasoning process. It is the sum of the sizes of all graphs that should be received by all nodes. Thus, the objective function of our approach is to minimize:

$$Cost = \sum_{i=1}^{n} \sum_{j=1}^{g} x_{i,j} S_j \tag{13}$$

3) Constraints: The constraints are derived from the memory costs for inferring graphs, see section V.

Capability. Input graphs are only produced by the nodes that have the required capability, *e.g.* only nodes hosting temperature sensors can produce a temperature observation graph.

$$y_{i,j} \leq P_{i,j} \ \ \forall i \ \forall j \tag{14}$$

Existence & unicity. Each graph should be produced at least on one node.

$$\sum_{i=1}^{n} y_{i,j} \leq 1 \ \forall j \tag{15}$$

Dependency. To produce a graph, a node should either produce or receive all of its antecedents.

$$y_{i,j}\, D_{j,h} \leq x_{i,h} \ \ + \ \ y_{i,h} \ \ \forall i \ \forall j \ \forall h \tag{16}$$

Memory. The memory consumed on each node should not exceed the node available memory (see Section V).

As stated before, our problem resembles a harder variant of NP-hard problems such as graph partitioning and GAP. There is no polynomial-time algorithm to solve it unless $P \neq NP$. Thus, we propose 3 algorithms to find approximate solutions: 2 greedy (*Greedy* and *GreedyPlus*) and 1 heuristic (*Heur*) algorithms.

B. Greedy algorithms

Intuitively, the *Greedy* algorithm 1 tries to produce a given intermediate or final graph on the node which already has the most required bytes (sum of the sizes of antecedent graphs) for this given graph. First the input graphs are placed on the sensor nodes where they are produced (line 18). Values of $\mathbb{X}$ are updated, as well as the specific cost of producing a given graph on each node (function $ProductionCost$), that only counts the antecedent graphs that need to be obtained over the network. The idea is to save the data corresponding to the antecedent

graphs already available on the local node. This is calculated using matrix O that represents the "economy" of not transmitted bytes. The algorithm iterates over all graphs according to their topological order in the dependency workflow (lines 20 - 31), and attempts to find the best node for each graph. Then it checks if the node has sufficient memory available. If yes then it allocates the graph in question on this node, otherwise it moves to the next best node.

Greedy is fast but not always optimal because it cannot see the combinations and allocations that could happen in further iterations.

GreedyPlus iterates further (lines 35 - 46), for every node, over either the graphs obtained from the network or the graphs that obtain their antecedents from the current node. For such graphs, it estimates the potential global savings if these graphs are rather produced on the current node. If the current node has sufficient memory and if it turns out advantageous, then it shifts the production of that graph to this node.

C. Linear Optimization algorithm

We now propose an approach to solve our problem using linear optimization tools.

1) Optimal solver: The optimal solver (OPT-solver) is our baseline solver for the equations formulated in Section VI-C. OPT-solver relies on the python PuLP library[9].

PuLP allows to define any linear programming problem by creating its variables and related equations. We did so for all inputs and equations 13 to 16 and 10. While running its core solver, it does an exhaustive search and outputs the optimal values of the decision variables as well as the value of the objective function, in our case: $\mathbb{X}, \mathbb{Y}$ and $Cost$. OPT-Solver runs in exponential time.

2) Heuristic algorithm: The *Heur* algorithm (Algorithm 2) implements our linear optimization approach. After fixing the input graphs on the nodes where they are produced, the heuristic algorithm first solves a relaxed version of the linear program (line 5). The solution obtained is fractional whereas we need binary values. It thus tries to heuristically round off the values of decision variables. It first sets the value of the highest decision variable to 1, then solves again the relaxed version of the linear program and moves on to the next variable. In case of infeasability, it backups initial values, sets the value of the decision variable in question to 0 and processes with the next potential variable.

Heur relies on the PuLP library as well. We implemented 2 functions to query the PuLP library: `FixValueInSolver()` is used to prevent PuLP from modifying the value of a decision variable in further iterations, and `SolveRelaxedLP()` obtains continuous decision variables from PuLP, instead of binary.

Its complexity is O(NGO_{LP} + N^2G) where O_{LP} is the complexity of solving the relaxed linear program.

D. Solutions for the running example

We ran the *Heur* algorithm for our example with two different sets of values: in the first one, all constrained nodes possess

[9]https://pypi.org/project/PuLP/

Algorithm 1: Greedy and GreedyPlus

```
1  Input: ℕ, 𝔾, 𝕄, 𝕊, 𝔸, D, P
2  Output: 𝕏, 𝕐
3  Function ProductionCost(𝕏, 𝕐):
4      o_{j,h} ← 0, ∀j, ∀h /* initialize economy matrix O   */
5      Cost ← 0 /* initialize total cost of solution  */
6      foreach i = 1 ··· n do
7          foreach j = 1 ··· g do
8              foreach h = 1 ··· g do
                   /* if node has a graph h then it has
                      d_{j,h} · S_h bytes that we need        */
9                  if y_{i,h} OR x_{i,h} then o_{i,j} ← o_{i,j} + d_{j,h} · S_h;
10                 if y_{i,j} · d_{j,h} = 1 & y_{i,h} ≠ 1 then
11                     x_{i,h} ← 1
12     foreach i = 1 ··· n do
13         foreach j = 1 ··· g do
14             Cost ← Cost + x_{i,j} · S_j
15     return 𝕏, O, Cost;
16 Function Greedy():
17     x_{j,h} ← 0, ∀j, ∀h /* initialize 𝕏                 */
18     𝕐 ← P /* initialize 𝕐                               */
19     𝕏, O ← ProductionCost(𝕏, 𝕐)
20     foreach j = 1 ··· g do
21         foreach iter = 1 ··· n do
               /* get best node index for graph j         */
22             i* ← arg max_i o_{i,j}
23             if M_{i*} ≥ K.S_{⋃_j A_j} + M_{setup}, ∀j (see eq. 10) then
24                 y_{i*j} ← 1
25                 y_{i,j} ← 0, ∀i ≠ i*
26                 break
27             else
28                 o_{i*,j} ← 0 /* to get next best node    */
29         if y_{i,j} = 0, ∀i then return infeasible;
30         𝕏, O ← ProductionCost(𝕏, 𝕐)
31     return 𝕏, 𝕐, Cost
32 Function GreedyPlus():
33     /* Run Greedy then see if we can improve more */
34     𝕏, 𝕐, Cost ← Greedy()
35     foreach i = 1 ··· n do
36         foreach j = 1 ··· g do
37             if (x_{i,j} = 1 OR y_{i,h} = 1, ∀h | h ∈ A_j) &
                  M_i ≥ K.S_{⋃_j A_j} + M_{setup}, ∀j then
                   /* see if Cost reduces when j produced
                      locally.                               */
38                 𝕏_{backup}, 𝕐_{backup} ← 𝕏, 𝕐
39                 x_{i,j} ← 0
40                 y_{i,j} ← 1
41                 y_{i⁻,j} ← 0, ∀i⁻ ≠ i
42                 𝕏, Cost_{new} ← ProductionCost(𝕏, 𝕐)
43                 if Cost_{new} > Cost then
44                     𝕏, 𝕐 ← 𝕏_{backup}, 𝕐_{backup}
45                     𝕏, Cost ← ProductionCost(𝕏, 𝕐)
46                 else Cost ←Cost_{new} ;
47     return 𝕏, 𝕐
```

Algorithm 2: Heuristic algorithm

Input: $\mathbb{N}, \mathbb{G}, \mathbb{M}, \mathbb{S}, \mathbb{A}, D, P$
Output: $\mathbb{X}, \mathbb{Y}$
foreach $j = 1 \cdots inputgraphs$ **do**
  `FixValueInSolver(`$y_{n,j} \leftarrow P_{n,j}$`)`
`SolveRelaxedLP()`
foreach $j = inputgraphs \cdots g$ **do**
  $i^* \leftarrow \arg\max_i(\mathbb{Y})$
  $iter \leftarrow 0$ `/* to iterate over nodes */`
  while $iter < n$ **do**
    $\mathbb{X}_{backup}, \mathbb{Y}_{backup} \leftarrow \mathbb{X}, \mathbb{Y}$
    `FixValueInSolver(`$y_{i^*j} \leftarrow 1$`)`
    $\mathbb{X}, \mathbb{Y} \leftarrow$ `SolveRelaxedLP()`
    if *feasible* **then**
      $\forall i \neq i^*$ `FixValueInSolver(`$y_{i,j} \leftarrow 0$`)`
      break;
    else
      $\mathbb{X}, \mathbb{Y} \leftarrow \mathbb{X}_{backup}, \mathbb{Y}_{backup}$
      $y_{i^*,j} \leftarrow 0$
      $i^* \leftarrow$ next best in $\arg\max_i(\mathbb{Y})$
    $iter \leftarrow iter + 1$
    continue;
  if *infeasible* **then return** Infeasible;
return $\mathbb{X}, \mathbb{Y}$

80KB of memory and the size of graphs is 750 B; in the second, the memory sizes vary among nodes and the size of graphs is 1850 B. The graphs received and produced on each node are respectively shown on Tables I and II.

Node	Node Memory	Graphs produced	Graphs received
N_1	80KB	G_1, G_5	-
N_2	80KB	G_2, G_4, G_6	-
N_3	80KB	G_3 G_7, G_8	G_4 G_5, G_6
N_4	80KB	-	-

TABLE I
DISTRIBUTION PLAN FOR THE RUNNING EXAMPLE WITH EQUAL MEMORIES. COST IS 2250 B

Node	Node Memory	Graphs produced	Graphs received
N_1	200KB	G_1, G_5, G_7, G_8	G_3, G_4, G_6
N_2	150KB	G_2,G_4, G_6	-
N_3	150KB	G_3	-
N_4	200KB	-	-

TABLE II
DISTRIBUTION PLAN FOR THE RUNNING EXAMPLE WITH VARIOUS MEMORIES. COST IS 5550 B

VII. NUMERICAL RESULTS AND EVALUATION

We evaluate the performance of our algorithms in terms of quality of results and processing time. We performed 1000 simulations, with the following varying parameters: K = 15, M_{setup} = 44KB (corresponding to LiRoT [7]), the node memory randomly varied between 100KB and 300KB, the number of nodes randomly varied between 5 and 50, the number of workflow stages varied from 2 to 5, the number of graphs randomly varied between 10 to 50, the input graph size randomly varied between 300 to 3000 bytes, and the other graph sizes randomly varied between 300 to 500 bytes. These simulations were performed using python scripts on a PC with 64G RAM and Intel(R) Xeon(R) W-2223 CPU @ 3.60GHz.

Some algorithms may not render results in certain cases:

- Some reasoning stages may require more memory than any node can provide. In this case, no algorithm can find a solution. However, there are some cases when some algorithms fail, while OPT-solver can find a solution. The percentage of cases when a solution is found is as follows: OPT-solver 98.8%, Heur 96.7%, Greedy and GreedyPlus 97.1% for the parameters considered in this paper.
- Our algorithms run in polynomial time whereas OPT-Solver runs in exponential time. As the problem is NP-hard, in some cases the OPT-solver kept running for more than 11 hours and was aborted. Otherwise it generally finished in 10 to 100s.

Gain vs. centralized. We first compare the network efficiency gain obtained by our 3 algorithms. The gain is the sum of the input and final graph sizes divided by $Cost$. It is relative to a centralised approach which has a network efficiency of 1.0 as it requires to send/receive all input/final graphs to/from a centralised reasoner.

$$Gain = \frac{\sum_j S_j \mid j \in \mathbb{G}_{input} \cup \mathbb{G}_{final}}{Cost} \tag{17}$$

Figure 2 shows the complimentary cumulative distribution function (CCDF) of the network efficiency gain. Note that CCDF shows the probability (Y-axis) of achieving a performance value greater than or equal to a given point (X-axis).

We can observe that the gain exceeds 1.0 in 80% (OPT-solver) to 60% (Greedy) cases. Thus, our distributed algorithms are better in the above cases since they save bandwidth as compared to a centralized approach. GreedyPlus is more than 2 times as efficient in 10% of cases, more than 1.5 times as efficient in 26% of cases and more than 1.0 times as efficient in 68% cases. For other cases the gain was less than 1.0: this is because the distributed algorithm pays the cost of distributing the workflow as compared to a centralized solution. This happens when the workflows are complex, memories are too low and some graphs are required multiple times for multiple workflows.

Gain vs. OPT-solver. We evaluate the performances of our 3 algorithms compared to the baseline by comparing their costs to that of OPT-solver, using the ratio:

$$Ratio = \frac{Cost(Algorithm)}{Cost(Opt - solver)} \tag{18}$$

Figure 3 depicts the CCDF. GreedyPlus generally performs best: its ratio is higher than 0.8 in approx. 70% cases. Next come Heur and Greedy which reach 0.8 or above in only approx. 50% cases.

Table III provides the execution times of the different algorithms. Greedy is the fastest and is able to find a solution in maximum 2.16s. GreedyPlus is slightly slower, as expected, as it runs post-processing steps after Greedy. Surprisingly, Heuristic is even slower than OPT-solver in some cases. Heuristic calls `SolveRelaxedLP()`. In general, relaxed linear

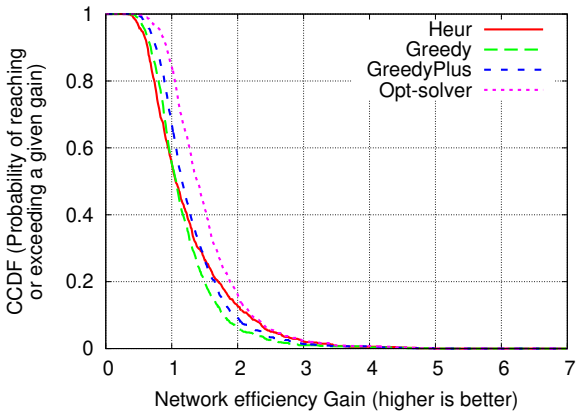

Fig. 2. CCDF of network efficiency Gain.

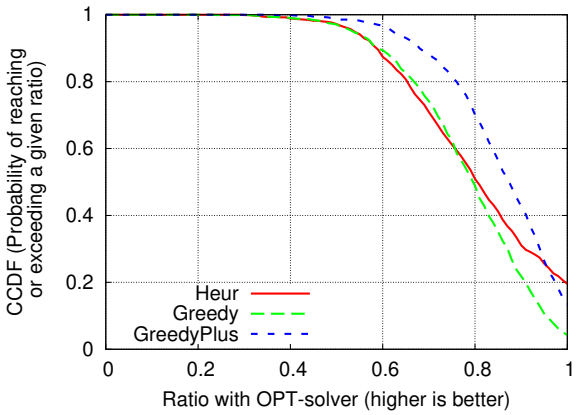

Fig. 3. CCDF of ratio (Cost(Greedy)/Cost(Optimizer)

programming problems can be solved in polynomial time. Thus, while Heuristic is a polynomial time algorithm, we suppose that the implementation of the interface functions with the PuLP library is not as optimized as the solver itself, and that the important number of calls to these functions slows down the algorithm. Lastly, OPT-solver took a huge amount of time in solving a few number of cases and had to be abandoned after running for 11 hours.

From above, we find that GreedyPlus represents a good compromise in terms of speed and performance.

VIII. Conclusion

This paper proposes an approach to distribute reasoning processes over constrained WoT nodes in an edge architecture, while minimizing data exchanges among nodes. We formulate this problem as a linear programming problem. We propose three polynomial time algorithms: two "greedy" ones and one heuristic based on a linear optimization solver. Our evaluations show that distributing computations can actually lead to reducing the network bandwidth consumption, compared to a centralized approach. We also find that the most advanced greedy algorithm provides a good compromise between performance and speed. In the future, we intend to study different optimization objectives and introduce additional parameters.

Algorithm	Min.	Avg.	Max.	Complexity
Opt-solver (abandoned)	0.11s	91.6s	4731.4s (40184.2s)	Exponential
Heuristic	0.42s	20.31s	200.7s	$O(NGO_{LP} + N^2G)$
Greedy	0.004s	0.43s	2.16s	$O(N^2G + NG^3)$
GreedyPlus	0.025s	1.53s	10.96s	$O(N^2G^3)$

TABLE III
Comparison of execution times different algorithms

IX. Acknowledgements

The Agence Nationale de la Recherche, France, supports this work in project CoSWoT[10] with grant ANR-19-CE23-0012.

References

[1] N. Seydoux, K. Drira, N. Hernandez, and T. Monteil, "Towards cooperative semantic computing: a distributed reasoning approach for fog-enabled swot," in *OTM 2018 Conferences: On the Move to Meaningful Internet Systems. Valletta, Malta, October 22-26, 2018*. Springer, 2018, pp. 407–425.

[2] T. Alam, B. Rababah, A. Ali, and S. Qamar, "Distributed intelligence at the edge on iot networks," *Annals of Emerging Technologies in Computing (AETiC)*, vol. 4, no. 5, pp. 1–18, 2020.

[3] A. Le-Tuan, C. Hayes, M. Hauswirth, and D. Le-Phuoc, "Pushing the scalability of rdf engines on iot edge devices," *Sensors*, vol. 20, no. 10, p. 2788, 2020.

[4] J. Pan and J. McElhannon, "Future edge cloud and edge computing for internet of things applications," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 439–449, 2017.

[5] V. Charpenay, S. Käbisch, and H. Kosch, "Introducing thing descriptions and interactions: An ontology for the web of things." in *SR+SWIT @ International Semantic Web Conference*, 2016, pp. 55–66.

[6] H. Baqa, M. Bauer, S. Bilbao, A. Corchero, L. Daniele, I. Esnaola, I. Fernández, Ö. Frånberg, R. G. Castro, M. Girod-Genet *et al.*, "Semantic iot solutions-a developer perspective," 2019.

[7] A. Bento, L. Médini, K. Singh, and F. Laforest, "Do arduinos dream of efficient reasoners?" in *Extended Semantic Web Conference 2022, Hersonissos, Crete, Greece, May 29–June 2, 2022*. Springer, 2022, pp. 289–304.

[8] P. Li, "Semantic reasoning on the edge of internet of things," *University of Oulu, Finland*, 2016.

[9] "Owl 2 rl profile specification," W3C Recommendation, 2012. [Online]. Available: https://www.w3.org/TR/owl2-profiles/#OWL_2_RL

[10] S. Mehla and S. Jain, "Rule languages for the semantic web," in *Emerging Technologies in Data Mining and Information Security*, A. Abraham, P. Dutta, J. K. Mandal, A. Bhattacharya, and S. Dutta, Eds. Singapore: Springer Singapore, 2019, pp. 825–834.

[11] J. Mei and H. Boley, "Interpreting swrl rules in rdf graphs," *Electronic Notes in Theoretical Computer Science*, vol. 151, no. 2, pp. 53–69, 2006.

[12] X. Su, P. Li, J. Riekki, X. Liu, J. Kiljander, J.-P. Soininen, C. Prehofer, H. Flores, and Y. Li, "Distribution of semantic reasoning on the edge of internet of things," in *2018 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, 2018, pp. 1–9.

[13] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz *et al.*, "Owl 2 web ontology language profiles," *W3C recommendation*, vol. 27, no. 61, 2009.

[14] A. Almeida and D. López-de Ipiña, "A distributed reasoning engine ecosystem for semantic context-management in smart environments," *Sensors*, vol. 12, no. 8, pp. 10 208–10 227, 2012.

[15] M. Intizar, P. Patel, S. K. Datta, and A. Gyrard, "Multi-layer cross domain reasoning over distributed autonomous iot applications," *Open Journal of Internet of Things*, vol. 3, 2017.

[16] A. I. Maarala, X. Su, and J. Riekki, "Semantic data provisioning and reasoning for the internet of things," in *2014 International Conference on the Internet of Things (IOT)*. IEEE, 2014, pp. 67–72.

[17] C. L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," in *Readings in Artificial Intelligence and Databases*. Elsevier, 1989, pp. 547–559.

[10] https://coswot.gitlab.io/

[18] J. Urbani and F. Harmelen, "Rdfs/owl reasoning using the mapreduce framework," 09 2023.
[19] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel computing*, vol. 26, no. 12, pp. 1519–1534, 2000.
[20] R. Dautov, H. Song, and N. Ferry, "A light-weight approach to software assignment at the edge," in *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2020, pp. 380–385.
[21] T. Öncan, "A survey of the generalized assignment problem and its applications," *INFOR: Information Systems and Operational Research*, vol. 45, no. 3, pp. 123–141, 2007.
[22] M.-H. Lin, J. G. Carlsson, D. Ge, J. Shi, J.-F. Tsai *et al.*, "A review of piecewise linearization methods," *Mathematical problems in Engineering*, vol. 2013, 2013.
[23] W. Tai, J. Keeney, and D. O'Sullivan, "Resource-constrained reasoning using a reasoner composition approach," *Semantic Web Journal*, vol. 6, no. 1, pp. 35–59, 2015.