Alma Mater Studiorum · Bologna
University

Cesena Campus
School of Engineering and Architecture
Department of Electrical Energy and Information Engineering
- Guglielmo Marconi -

Master's Degree Course in
Electronic Engineering and Telecommunications for Energy

STUDY OF DISTRIBUTED LAGRANGIAN HEURISTIC FOR SELF-ADAPTIVE PUBLISH-SUBSCRIBE NETWORK DESIGN PROBLEMS

Optimization Methods for Networks and Energy

Relator:
Marco Antonio Boschetti

Edited by:
Leonardo Urbinati

Academic Year 2021-2022

Contents

1 Introduction **10**
- 1.1 IoT Decentralized Architecture . . . 10
- 1.2 Publish/Subscribe Model & MQTT . . . 11
- 1.3 Peer to Peer . . . 12
- 1.4 Lagrangian Relaxation . . . 13
  - 1.4.1 Mathematical Explanation . . . 14
  - 1.4.2 Subgradient Method . . . 15
- 1.5 Continuous Relaxation . . . 18

2 Problem Description **19**
- 2.1 Problem Scenario . . . 19
- 2.2 First Formulation . . . 21
- 2.3 Second Formulation . . . 22
  - 2.3.1 Lagrangian Relaxation . . . 23

3 Algorithm Implementation **25**
- 3.1 Centralized Version with Formulation 1 . . . 25
- 3.2 Continuous Relaxation of Formulation 1 . . . 28
- 3.3 Centralized Version with Formulation 2 . . . 28
- 3.4 Continuous Relaxation of Formulation 2 . . . 30
- 3.5 Heuristic Algorithm for Network Design Problems . . . 31
  - 3.5.1 Overview of the Implemented Heuristic Algorithm . . . 31
  - 3.5.2 Technical details of the code implementation . . . 36
- 3.6 Lagrangian Relaxation of Formulation 2 . . . 40
  - 3.6.1 Knapsack Problem and Function Implementation . . . 44
- 3.7 Distributed Lagrangian Heuristic . . . 47
  - 3.7.1 Distributed Lagrangian Heuristic - First Approach . . . 56
  - 3.7.2 Distributed Lagrangian Heuristic - Second Method . . . 59
- 3.8 Alternative Distributed Heuristic . . . 63
- 3.9 Auxiliary Functions . . . 68

4 Results Analysis **71**
- 4.1 Simulation Scenarios . . . 71
- 4.2 Simulations with Formulation 1 . . . 79
- 4.3 Simulations with Formulation 2 . . . 88
  - 4.3.1 Analysis of Heuristic Algorithm Results . . . 88
  - 4.3.2 Analysis of the Lagrangian Relaxation for Formulation 2 89
- 4.4 Simulation with Distributed Algorithm . . . 102

4.4.1 Results of Distributed Heuristic Algorithm - First Apporach . . . 103
4.4.2 Results of Distributed Heuristic Algorithm - Second Apporach . . . 105
4.4.3 Dynamic Adaptability of the Distributed Algorithm . . 109
4.4.4 Comparison of results obtained by all algorithms with Formulation 2 . . . 112

5 Conclusions **121**

List of Figures

1 Difference Between Client-server and Publish/Subscribe 11
2 Example of Lagrangian Function 15
3 Step 1 - Sub-gradient Method 16
4 Step 2 - Sub-gradient Method 17
5 Example of sensor network 19
6 Heuristic Algorithm Structure 32
7 Network Example for Dijkstra's Algorithm - step 1 33
8 Network Example for Dijkstra's Algorithm - step 2 34
9 Network Example for Dijkstra's Algorithm - step 3 35
10 Description of First Approach of the Distributed Lagrangian Heuristic Algorithm 58
11 Description of Second Approach of the Distributed Lagrangian Heuristic Algorithm 62
12 Example for Heuristic Distributed Algorithm - step 1 63
13 Example for Heuristic Distributed Algorithm - step 2 63
14 Example for Heuristic Distributed Algorithm - step 3 63
15 Example for Heuristic Distributed Algorithm - step 4 64
16 Example for Heuristic Distributed Algorithm - step 5 64
17 Tiny Instance 72
18 case2 Instance 73
19 case3 Instance 74
20 case4 Instance 75
21 case5 Instance 76
22 case6 Instance 77
23 case7 Instance 78
24 case Tiny solved with Formulation 1 using CPLEX 80
25 case2 solved with Formulation 1 using CPLEX 81
26 case3 solved with Formulation 1 using CPLEX 82
27 case4 solved with Formulation 1 using CPLEX 83
28 case5 solved with Formulation 1 using CPLEX 84
29 case6 solved with Formulation 1 using CPLEX 85
30 case7 solved with Formulation 1 using CPLEX 86
31 case8 solved with Formulation 1 using CPLEX 87
32 Percentage deviation from the optimal value of heuristic algorithm 89
33 Lower Bound trend for instance 'case2' with α=0.1 on 1000 iteratinos 90

34 Upper Bound influnece on Lower Bound values, simulated with instance 'case2', α=0.1 for 2000 iterations 91
35 Parameter α influnece on Lower Bound values, simulated with instance 'case2', for 2000 iterations 92
36 case Timy solved with Formulation 2 using CPLEX 94
37 case2 solved with Formulation 2 using CPLEX 95
38 case3 solved with Formulation 2 using CPLEX 96
39 case4 solved with Formulation 2 using CPLEX 97
40 case5 solved with Formulation 2 using CPLEX 98
41 case6 solved with Formulation 2 using CPLEX 99
42 case7 solved with Formulation 2 using CPLEX 100
43 case8 solved with Formulation 2 using CPLEX 101
44 Example of bad decisions and good decisions made by brokers of the network 105
45 Comparison of approaches 1 and 2 on the average cost of solutions 108
46 C Comparison of approaches 1 and 2 on the average times to obtain solutions 108
47 Optimal Solution Instance 'case5' before node 8 failure 110
48 Situation on Instance 'case5' immediately after node 8 failure . 111
49 New Solution found for Instance 'case5' after node 8 failure . . 111
50 case Timy solved with second approach of Distribute Lagrangian Heuristic 113
51 case2 solved with second approach of Distribute Lagrangian Heuristic 114
52 case3 solved with second approach of Distribute Lagrangian Heuristic 115
53 case4 solved with second approach of Distribute Lagrangian Heuristic 116
54 case5 solved with second approach of Distribute Lagrangian Heuristic 117
55 case6 solved with second approach of Distribute Lagrangian Heuristic 118
56 case7 solved with second approach of Distribute Lagrangian Heuristic 119
57 case8 solved with second approach of Distribute Lagrangian Heuristic 120

List of Tables

1 Example of Realization of Dijkstra's Table - step 1 33
2 Example of Realization of Dijkstra's Table - step 2 34
3 Example of Realization of Dijkstra's Table - step 3 34
4 Example of Realization of Dijkstra's Table - step 4 35
5 Example of Knapsack table 45
6 Results Obtained with Formulation 1 with integrality constraints, and Continous Relaxation 79
7 Analysis of centralized heuristic algorithm 89
8 Results obtained from different algorithms 93
9 Results of costs and mean cost of solutions obtained with first approach 103
10 Results of number of reset and mean number of reset with first approach 104
11 Time estimation and mean time for obtaining solutions with first approach 104
12 Results of costs and mean cost of solutions obtained with second approach 106
13 Results of number of reset and mean number of reset with second approach 107
14 Time estimation and mean time for obtaining solutions with first approach 107
15 Percentage of optimal solutions found 109

Abstract

The Internet of Things (IoT) is an emerging technology that has revolutionized the collection and processing of information through the interconnection of smart objects. "Thing" or "object" in the context of IoT refers to a wide range of device categories, equipment, installations and systems, physical materials and products, works and goods, as well as machines and equipment. In other words, it is any type of object that can be connected to the internet and transmit data for the collection and processing of information. Usually, IoT devices send data to cloud servers, but this can cause connectivity and data transfer problems, especially in areas with limited resources. To overcome these obstacles, researchers are studying edge computing-based solutions, which involve processing data directly at the source, rather than on remote servers. This approach could enable more efficient and effective services, but requires a transition to a decentralized and interconnected IoT framework. However, current IoT infrastructures are not yet ready for this type of transition. The main challenge is to integrate edge computing capabilities into these infrastructures, while ensuring their usability and high availability. The Publish/-Subscribe communication method allows for data exchange between network components without the need for direct connections between them. The use of the MQTT protocol, one of the main protocols used for message exchange between IoT devices, offers numerous advantages such as lightweight and scalability. However, this communication model also has some limitations, such as the vulnerability of the broker, the system's weak point, and the difficulty of managing numerous IoT devices. To overcome these challenges, new solutions based on a peer-to-peer approach are being developed.

This thesis examines the possibility of using multiple distributed MQTT brokers on different interconnected machines as a promising solution to improve system reliability and scalability. However, since this configuration requires careful and coordinated management of the various brokers, so that they act as a single entity, it is necessary to explore the use of a fully-distributed optimization solution based on a Lagrangian relaxation approach. This approach allows finding an optimal solution for a complex optimization problem by breaking it down into simpler sub-problems. In particular, the potential of using Lagrangian relaxation to optimize data distribution between the various MQTT brokers is evaluated, thus ensuring optimal load balancing and reliability for the entire system. The main objective of the project is to evaluate the effectiveness of distributed Lagrangian

heuristic algorithms in the field of communication network management. These algorithms allow network nodes to act autonomously, based only on information about themselves and neighboring nodes they can communicate with, without the need for centralized management. In particular, each node must execute an algorithm that uses Lagrangian penalty calculation and message exchange to make heuristic choices in order to establish effective connections in the network. The goal is therefore to propose algorithms that improve network efficiency and reliability without the need for centralized management. The thesis is divided into five chapters, each of which focuses on specific aspects of the research.

Chapter 1 introduces the IoT field and the publish/subscribe approach, which form the basis of the research work. In addition, Lagrangian relaxation and the subgradient method are presented, which will be used later for the implementation of the algorithms.

Chapter 2 focuses on the problem scenario and the mathematical formulation of the optimization problems that need to be solved. In particular, two optimization models and their relaxations are analyzed.

Chapter 3 presents the algorithms implemented in the research work, which are based on the distributed Lagrangian heuristic approach. The proposed algorithms aim to optimize the distribution of data between MQTT brokers, achieving load balance and optimal reliability for the entire system.

Chapter 4 presents the experimental results obtained by implementing the proposed algorithms in a simulated environment. The results show that the proposed algorithms can effectively improve the efficiency and reliability of the network.

Finally, Chapter 5 summarizes the research work carried out and presents possible applications of the developed algorithms in a real-world context, as well as potential future improvements.

In conclusion, the research work presented in this thesis provides a promising solution to the challenges posed by the increasing complexity and scale of IoT networks. The proposed approach based on distributed Lagrangian heuristic algorithms can effectively improve the efficiency and reliability of the network, without the need for centralized management.

1 Introduction

1.1 IoT Decentralized Architecture

The Internet of Things (IoT) has revolutionized the way in which we collect and process informations. At its core, the IoT is built on the concept of "intelligent" objects that are interconnected and able to exchange the information they possess. The aim is to enable the collection and processing of data from diverse sources with the ultimate goal of providing enhanced services and products to end-users.

Typically, data from IoT devices are transmitted to servers located in the cloud. However, these servers can often be situated far from the devices themselves, resulting in transfer time issues and connection failures. Moreover, connectivity in rural and remote areas can be poor, resulting in poor performance and limited functionality. As a result, researchers are now looking at edge-computing solutions. These solutions enable the deployment of artificial intelligence in areas where connectivity is poor and resources are limited, such as rural areas. Such an approach can potentially provide more efficient and responsive services by processing data at its source, i.e., at the edge. Moving towards a decentralized and interconnected Internet of Things (IoT) framework could be a promising solution. However, the current IoT infrastructures are not yet ready for this transition to decentralization. The primary challenge lies in effectively incorporating edge computing features into existing IoT infrastructures, while maintaining the usability and high availability of the current systems. In addition to this, it is important to understand what additional benefits this new paradigm can offer to the IoT. Decentralization could potentially enhance security, reduce costs, and enable more efficient sharing of resources. As such, the research focus should be on developing a framework that can seamlessly integrate edge computing into the IoT architecture without compromising the usability and high availability of the system. In summary, the Internet of Things (IoT) is a rapidly evolving field that has the potential to revolutionize the way in which we interact with our environment. By shifting to a decentralized and interconnected IoT framework, it is possible to enhance the efficiency and responsiveness of services by processing data at the edge, which is the source of the data. However, achieving this goal would require significant research and development efforts, and a concerted effort to integrate edge computing features into the existing IoT infrastructure.

1.2 Publish/Subscribe Model & MQTT

The Publish/Subscribe (also Pub/Sub) communication pattern has become a fundamental pillar of IoT applications, enabling the exchange of data between different components of a network without direct connections between them. By distinguishing the client into publishers and subscribers, Pub/Sub has revolutionized the traditional client-server model. The broker, responsible for routing and distributing information, is the backbone of this model. Pub/Sub offers greater flexibility and scalability, as it does not require clients to have knowledge of each other's existence (Figure 1).

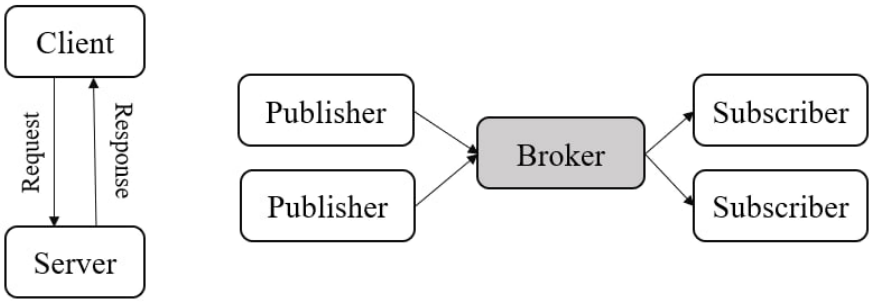

Figure 1: Difference Between Client-server and Publish/Subscribe

One of the main protocols used for exchanging messages between IoT devices is MQTT, which was specifically designed to be lightweight and scalable. MQTT makes it possible to exchange information between publishers and subscribers without direct communication between them, thus making it ideal for IoT devices with limited resources, such as those with long battery life. Despite the advantages of Pub/Sub and MQTT, they have some limitations. The broker represents the weak point of the system, and if it becomes inaccessible, all communications are interrupted. Furthermore, these systems struggle to handle the considerable number of IoT devices expected in the future. These challenges have led to the development of new solutions based on a peer-to-peer approach, where participants are connected through publishers and subscribers that are linked to specific topics.

When choosing a message broker infrastructure, it is crucial to consider various design considerations such as latency, bandwidth, message handling, service availability, service reliability, and security. To overcome these challenges, it is possible to use multiple distributed MQTT brokers on different machines connected via a network, which act as a single entity. This approach ensures high scalability, replication, elasticity, and resiliency to fail-

ures. However, it may increase latency and communication overhead between brokers, which can be problematic in an IoT scenario where deployments are often in constrained or frugal environments with brokers located at the network's edge. Hence, the optimal network configuration that can transmit the maximum number of messages to subscribers or minimize the number of unfulfilled network requests should be discovered to tackle this network design challenge.

1.3 Peer to Peer

The P2P (peer-to-peer) protocol is a communication model in which participants communicate directly with each other, without a central server. In a P2P system, each participant is both a client and a server, and each participant can provide and request resources or services from other participants. The P2P protocol was originally used for file sharing, where participants can share and download files directly from other participants without going through a central server. However, the P2P protocol can be used in many other applications, such as sharing computing resources, distributing real-time content, social networks, gaming networks, and more. There are different types of P2P architectures, including structured and unstructured ones. In structured P2P networks, participants are organized in a well-defined network structure, such as a tree or a distributed hash table, which allows for efficient resource localization. In unstructured P2P networks, participants are organized randomly, making it more difficult to find desired resources. There are also several challenges associated with implementing P2P systems, such as scalability, security, availability, and bandwidth management. However, the P2P protocol remains an important and increasingly used communication model, especially in the era of the Internet of Things and edge computing.

The idea is to create distributed algorithms in which each node works autonomously to create connections with neighboring nodes, in order to create a direct link between the requester and the data sender. To solve this problem, Lagrangian relaxations will be used, which allow the problem to be divided into simpler sub-problems for each node. In the next pages, the concept of Lagrangian relaxation and the subgradient technique will be introduced to provide practical context to the development of the algorithm.

1.4 Lagrangian Relaxation

Given the use of Lagrangian relaxations in the problems that will be presented, a brief and preliminary introduction to these techniques is deemed necessary to establish a sufficient conceptual foundation for their application.

The approach of Lagrangian relaxations is a very useful technique in many fields of optimization, as it allows for the efficient handling of optimization problems with constraints. A common example where it is used is the linear optimization problem with constraints, where the goal is to maximize or minimize a linear function (called objective function) subject to linear constraints. The key concept is to convert the constrained optimization problem into an unconstrained optimization problem by relaxing only certain constraints, particularly those that make the problem challenging to solve, by introducing the constraints as part of the objective function to be minimized or maximized. In this way, standard optimization techniques can be used to solve the problem without directly considering all the constraints. In practice, this is done by associating to each relaxed objective function constraint a Lagrange multiplier. In this way, the constraints become part of the objective function to be minimized or maximized, and the problem becomes a simpler optimization problem. The result of this operation is a new objective function, called the Lagrangian function, which depends on both the original variables of the problem and the Lagrange multipliers. By solving the optimization problem related to the Lagrangian function, one obtains the optimal values of the original variables, together with the optimal values of the Lagrange multipliers. An important aspect to note is that the Lagrange multipliers can be interpreted as the "penalties" associated with the constraints that have been introduced. In other words, the higher the value of the Lagrange multiplier associated with a constraint, the greater the penalty associated with that constraint in the optimization problem. In general, the approach of Lagrangian relaxations is a very powerful and flexible technique that allows for the efficient solution of a wide range of optimization problems with constraints. It is widely used in fields such as engineering, economics, social sciences, and many others, and is an essential tool for any optimization expert.

1.4.1 Mathematical Explanation

Consider an integer linear programming problem **P**

$$z(P) = min\, c^T x \tag{1}$$

s.t.

$$Ax \geq b \tag{2}$$

$$Bx \geq d \tag{3}$$

$$x \in \{0, 1\} \tag{4}$$

Let c be the cost vector, x be the variable vector of the problem, A and B be the matrix of coefficients of the constraints, b and d be the vector of the right-hand sides of the constraints.

Lagrangian relaxation involves relaxing the difficult constraints and introducing them into the objective function by adding a non-negative penalty vector. For example the constraint (2) is satisfied if:

$$Ax - b \geq 0 \tag{5}$$

Then we can reformulate the relaxed problem as:

$$L(\lambda) = min\, c^T x - \lambda(Ax - b) \tag{6}$$

s.t.

$$Bx \geq d \tag{7}$$

$$x \in \{0, 1\} \tag{8}$$

The minus sign is due to the fact that the constraint is satisfied when it assumes a value greater than or equal to zero, and the penalties are assumed non-negative. Since this is a minimization problem, we want to penalize the constraint when it is not satisfied (i.e., when it assumes a negative value). By subtracting the constraint, we are in fact adding a quantity to the original problem, thereby opposing its minimization.

The optimal value of *z(P)* for the original problem will be greater than or at most equal to the optimal value of $L(\lambda)$ for the relaxed problem.

$$z(P) \geq L(\lambda), \qquad \forall \lambda \geq 0 \tag{9}$$

Therefore, the relaxed problem provides us with a lower bound for the optimal value of the original problem.

The Lagrangian penalities that maximize the Lagrangian function $L(\lambda)$ can be found by solving the following problem, called Lagrangian Dual:

$$z(D_L) = \max_{\lambda \geq 0} [L(\lambda)] \tag{10}$$

Since the Lagrangian relaxed problem provides a lower bound, maximizing the estimated Lagrangian provides the best lower bound for the problem. The subgradient method is one of the techniques that can be used to search the optimal (or near-optimal) solution of the Lagrangian Dual.

1.4.2 Subgradient Method

The subgradient method is an iterative method used to maximize or minimize a convex or concave nondifferentiable function, such as the Lagrangian function. Specifically, the subgradient method can be used to compute the Lagrange multipliers associated with the relaxed constraints of an optimization problem. The best bound can be obtained by maximizing the value of the Lagrangian function (Lagrangian Dual). The Lagrangian function is concave, as can be observed by sampling some values of the function. More specifically, the Lagrangian function is piecewise linear, and the envelope of these segments results in a concave function (as shown in Figure 2). The segments that compose this function are associated with specific vectors $\bar{x}$ that characterize the optimal solution.

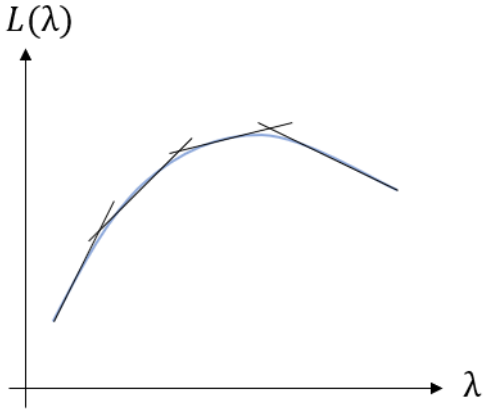

Figure 2: Example of Lagrangian Function

Given lambda, let $\bar{x}$ be the solution of the Lagrangian relaxation L($\bar{\lambda}$), which identifies a point in the function:

$$L(\bar{\lambda}) = c\bar{x} - \bar{\lambda}(A\bar{x} - b) \tag{11}$$

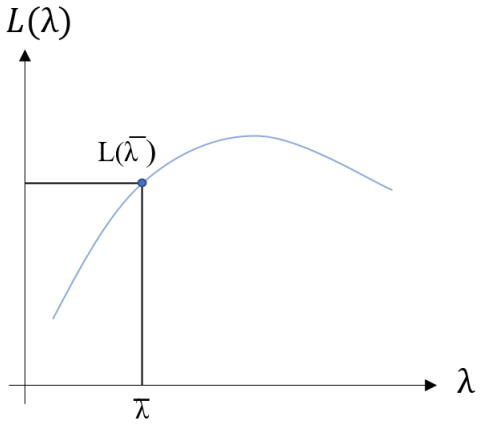

Figure 3: Step 1 - Sub-gradient Method

For a generic lambda, we have:

$$L(\lambda) = min\,(c^T x - \lambda(Ax - b)) \leq c\bar{x} - \lambda(A\bar{x} - b) \tag{12}$$

By subtracting equation (11) from equation (12), we obtain:

$$L(\lambda) - L(\bar{\lambda}) \leq -(\lambda - \bar{\lambda})(A\bar{x} - b) \tag{13}$$

that is:

$$L(\lambda) \leq L(\bar{\lambda}) - (A\bar{x} - b)(\lambda - \bar{\lambda}) \tag{14}$$

It follows that: $s = -(A\bar{x} - b)$ is a subgradient of L(λ) at $\bar{\lambda}$.

For L(λ) to be greater than L($\bar{\lambda}$), it is necessary that:

$$-(A\bar{x} - b)(\lambda - \bar{\lambda}) > 0 \tag{15}$$

So it is necessary to move in the direction of subgradient:

$$\lambda = \bar{\lambda} + \theta s \qquad \theta > 0 \tag{16}$$

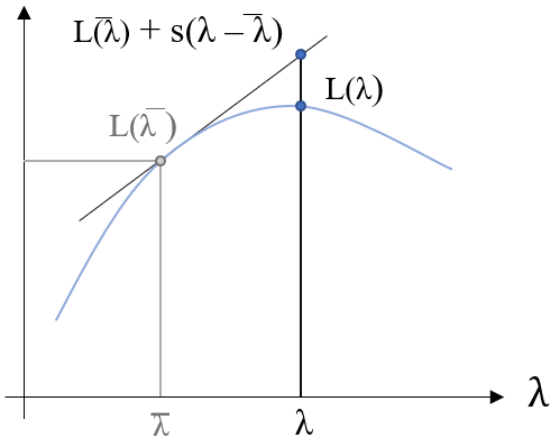

Figure 4: Step 2 - Sub-gradient Method

The choice of the parameter θ is crucial, as a step that is too large may cause the divergence of the method (low values of lower bound far from optimal solution of Lagrangian Dual), while a step that is too short may result in too many steps being required.

1.5 Continuous Relaxation

To understand the concept of continuous relaxation, it's important to first have a basic understanding of optimization problems.
An optimization problem is a mathematical problem where the goal is to find the best solution from a set of possible solutions. This can involve maximizing or minimizing an objective function subject to certain constraints. In some cases, the variables in the optimization problem are restricted to be integers, which are called integer variables. However, solving optimization problems with integer variables can be computationally complex and time-consuming. This is where continuous relaxation comes in. In some cases, it is possible to transform an optimization problem with integer variables into an equivalent optimization problem with continuous variables. This is done by relaxing the integrality constraint on the variables, allowing them to take on any value in a continuous range. This approach is called "continuous relaxation". Continuous relaxation is a common technique used in optimization to deal with the computational complexity associated with solving problems with discrete variables. By allowing the variables to take on fractional values, the relaxed problem can be solved using standard optimization techniques, providing a lower bound on the optimal solution of the original problem. The difference between the optimal solutions of the relaxed problem and the original problem is known as the integrality gap.
Compared to Lagrangian relaxations, which introduce penalties associated with some constraints, continuous relaxations eliminate the integrality constraint by allowing variables to take on fractional values. In some cases, it is possible to derive valid bounds on the integrality gap, which can be used to evaluate the quality of the solution obtained from the relaxed problem. If the continuous relaxation is equivalent to the Lagrangian relaxation, we say that there is an integrality property. It is important to underline that the Lagrangian relaxation could dominate the continuous relaxation and not vice versa. Continuous relaxations are widely used in fields such as operations research, computer science, and engineering, and are an essential tool for any optimization expert. Overall, continuous relaxation is a powerful technique that allows for the efficient optimization of complex problems with integer variables.

2 Problem Description

2.1 Problem Scenario

The scenario presented in this thesis concerns a network of distributed nodes that exchange data (**commodities**) between information generators (**Publishers**) and network elements interested in receiving them (**Subscribers**). In this context, intermediary nodes (**Brokers**) play the role of information distributors, routing them to Subscribers or other Brokers. To ensure efficient exchange of information, various factors are considered, such as:

- Capacity of connections (e.g., available bandwidth);
- Cost of connection usage (e.g., energy consumption);
- Weight of each commodity (e.g., bandwidth occupation).

In the network, not all nodes are visible to each other, but there are subgroups of nodes that have limited access to information from others. This can affect the network's ability to exchange information.

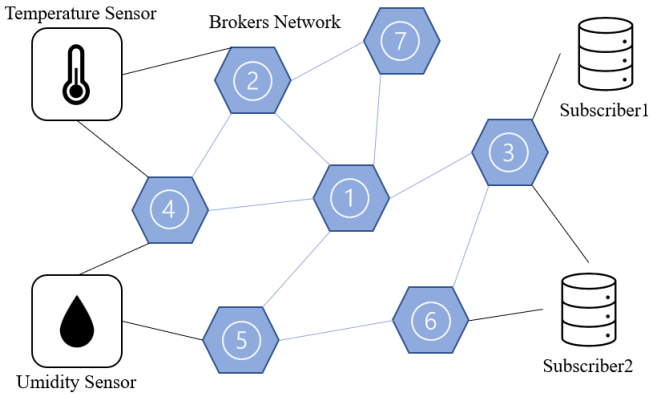

Figure 5: Example of sensor network

Figure 5 presents a simple network model for the Pub/Sub problem, where there are two sensors (humidity and temperature) acting as publishers and two data storage systems acting as subscribers. The network consists of 7 brokers that communicate with each other and with the commodity generators/requesters.
Brokers in the network do not store data themselves, but only the tags requested by clients or other brokers with which they are connected. Their

role is to forward data that is compatible with the stored requests. Brokers are also able to request data from other nodes in order to forward the data over the network and create dynamic paths between data producers and externally connected nodes. It is also assumed that commodity flows cannot be split and recombined at the destination. In summary, the problem can be seen as a network designing problem. This particular version of the problem is Known as Publish/Subscribe network design problem (described at subsection 1.2). The main objective of the project is to minimize the overall cost of the network in terms of connections made, while maximizing the number of commodities provided to requesting Subscribers. To achieve this goal, two main approaches are considered, which will be explored in detail in the following chapters of the paper.

To better introduce the objective functions analyzed in this thesis, it is necessary to provide a brief explanation of the terminology used for the variables involved:

- K, set of commodities
- S, set of source nodes
- s_k, node that produce the commodity k ($s_k \subseteq$ S)
- T, set of destination nodes
- T_k, set of destination that requires commodity k ($T_k \subseteq$ T)
- B, set of brokers
- N, set of nodes (B $\cup$ S $\cup$ T)
- A, set of directed arcs between nodes
- Γ_i, set of end-nodes of arcs outgoing from node i
- Γ_i^{-1}, set of end-nodes of arcs incoming to node i
- x_{ij}^k, integer variable indicating how many subscribers requesting commodity k are served through the arc (i, j).
- $\xi_{i,j}^k$, binary variable equal to 1 if commodity k is transmitted along the arc i,j, 0 otherwise.
- $u_{i,j}$, the total capacity of the arc (i, j).

- $c_{i,j}$, the cost for using the arc (i, j).
- a^k, bandwidth occupation of the commodity k.
- b_i^k, which is $b_i^k \geq 0$ for nodes that produce or emit the commodity k, $b_i^k = -1$ for nodes that require the commodity k, 0 otherwise.

2.2 First Formulation

The first formulation of the problem is aimed at maximizing the number of satisfied requests. This is formalized through a standard minimum-cost maximum flow structure, where the feasible flow distributions that achieve the maximum satisfaction goal are compared and the one with the lowest cost is chosen. The problem of maximizing can be easily transformed into the problem of minimizing the number of unsatisfied requests by adding dummy arcs with high cost that enter the sink nodes, which will be used only when no other feasible options are available to satisfy their request.

Objective Function:

$$\textbf{minimize} \sum_{k \in K} \sum_{i \in N} \sum_{j \in \Gamma_i^{-1}} c_{ji} x_{ji}^k \tag{17}$$

Subject to:

$$\sum_{j \in \Gamma_i^{-1}} x_{jt}^k = 1 \qquad \forall t \in T^k, k \in K \tag{18}$$

$$\sum_{j \in \Gamma_i^{-1}} x_{ji}^k = \sum_{j \in \Gamma_i} x_{ij}^k \qquad \forall i \in B, k \in K \tag{19}$$

$$M\xi_{ij}^k \geq x_{ij}^k \qquad (i, j) \in A, k \in K \tag{20}$$

$$\sum_{k \in K} a^k (\xi_{ij}^k + \xi_{ji}^k) \leq u_{ij} \qquad \forall i \in N, j \in \Gamma_i \tag{21}$$

$$x_{ij}^k \geq 0 \qquad \forall (i, j) \in A, k \in K \tag{22}$$

$$\xi_{ij}^k \in \{0, 1\} \qquad \forall (i, j) \in A, k \in K \tag{23}$$

The objective function of the problem is represented by Equation (17). To optimize the problem, we assign non-negative costs to the arcs between

nodes in the network. The goal is to minimize the cost of flow within the network and discourage direct flow between source and destination nodes using dummy arcs. Equation (18) ensures that every destination node for a commodity k receives that commodity. Equation (19) enforces the constraint that every commodity entering a broker node i must also exit the node, thus ensuring the conservation and continuity of flow within the network. Constraints (20) links the variables x_{ij}^k and ξ_{ij}^k, ensuring that each x_{ij}^k is 1 when the corresponding arc has a nonzero flow for commodity k. Finally, Constraints (21) imposes the capacity constraint on the arcs, while constraints (22) and (23) enforce the integrality constraints on the decision variables.

2.3 Second Formulation

In the second formulation, the minimization of the number of activated channels for node connections (variable ξ) is considered, unlike in the first formulation where the number of commodities served via the arc (i, j) (variable x) was considered.

Objective Function:

$$\textbf{minimize} \sum_{k \in K} \sum_{i,j \in A} c_{ij} \xi_{ij}^k \tag{24}$$

Subject to:

$$\sum_{j \in \Gamma_i} x_{ij}^k - \sum_{j \in \Gamma_i^{-1}} x_{ji}^k = b_k^i \qquad \forall i \in N \setminus \{s_k\}, k \in K \tag{25}$$

$$\sum_{k \in K} a^k \xi_{ij}^k \leq u_{ij} \qquad \forall (i,j) \in A \tag{26}$$

$$\xi_{ij}^k \leq x_{ij}^k \leq \xi_{ij}^k |T_k| \qquad \forall (i,j) \in A, k \in K \tag{27}$$

$$x_{ij}^k \geq 0 \qquad \forall (i,j) \in A, k \in K \tag{28}$$

$$\xi_{ij}^k \in \{0,1\} \qquad \forall (i,j) \in A, k \in K \tag{29}$$

The main goal of this formulation of the problem is to minimize the overall cost of the utilized connections, as expressed in the objective function (24,). The flow conservation constraints (25) guarantee that the amount of commodity entering a broker node is equal to the amount exiting it, ensuring

the continuity of the flow. The capacity constraints (26) impose a limit on the amount of commodities that can be transmitted through a connection without exceeding its maximum capacity. Additionally, the linking constraints (27) relate the two sets of variables and ensure that there is a nonzero flow of commodity k only if the connection is "open" (i.e., $\xi_{ij}^k = 1$).

2.3.1 Lagrangian Relaxation

We can relax problem (24) -(29) using Lagrangian approach, dualizing the constraints (25) in the objective function by adding Lagrangian penalties λ_i^k. These penalties correspond to the cost of violating the constraints, and they allow the objective function to be optimized subject to the remaining constraints. By dualizing the constraints, we obtain a new objective function that is easier to solve, as it involves fewer constraints. However, this new objective function may not necessarily have the same optimal solution as the original problem, but it guarantees that it is a valid lower bound.
Lagrangian Relaxation:

$$(LR) \quad z_{LR}(\lambda) = min \sum_{k\in K} \sum_{i,j\in A} c_{ij}^k \xi_{ij}^k + (\lambda_i^k - \lambda_j^k) x_{ij}^k - \sum_{k\in K} \sum_{i\in N} \lambda_i^k b_i^k \tag{30}$$

s.t.

$$\sum_{k\in K} a_k \xi_{ij}^k \leq u_{ij} \qquad (i,j) \in A \tag{31}$$

$$\xi_{ij}^k \leq x_{ij}^k \leq |T_k| \xi_{ij}^k \qquad (i,j) \in A, k \in K \tag{32}$$

$$x_{ij}^k \geq 0 \qquad (i,j) \in A, k \in K \tag{33}$$

$$\xi_{ij}^k \in [0,1] \qquad (i,j) \in A, k \in K \tag{34}$$

The problem LR (30) - (34) can be decomposed into $|A|$ sub-problems, denoted as LR_{ij}, with one sub-problem corresponding to each arc in the graph. Each sub-problem is defined as follows:

$$(LR_{ij}) \quad z_{LR}(\lambda) = min \sum_{k\in K} c_{ij}^k \xi_{ij}^k + (\lambda_i^k - \lambda_j^k) x_{ij}^k \tag{35}$$

s.t.

$$\sum_{k\in K} a_k \xi_{ij}^k \leq u_{ij} \qquad (i,j) \in A \tag{36}$$

$$\xi_{ij}^k \leq x_{ij}^k \leq |T_k|\xi_{ij}^k \qquad (i,j) \in A \tag{37}$$

$$x_{ij}^k \geq 0 \qquad (i,j) \in A \tag{38}$$

$$\xi_{ij}^k \in \{0,1\} \qquad (i,j) \in A \tag{39}$$

So, the Lagrangian Relaxation at (30) can be written as:

$$z_{LR}(\lambda) = \sum_{i,j \in A} z_{LR_{ij}}^k(\lambda) - \sum_{k \in K} \sum_{i \in N} \lambda_i^k b_i^k \tag{40}$$

To obtain the solution of each sub-problem $z_{LRij}(\lambda)$ (35), the following knapsack problem needs to be solved:

$$(LR_{ij}) \quad z_{LRij}(\lambda) = min \sum_{k \in \bar{K}} \bar{c}_{ij}^k \xi_{ij}^k \tag{41}$$

s.t.

$$\sum_{k \in \bar{K}} a_k \xi_{ij}^k \leq u_{ij} \tag{42}$$

$$\xi_{ij}^k \in \{0,1\} \qquad (i,j) \in A \tag{43}$$

where $\bar{K}$ and $\bar{c}_{ij}^k$ and the solution are defined as follows:

- if $\bar{c}_{ij}^k + \lambda_i^k - \lambda_j^k \geq 0$, then k is not included in $\bar{K}$ and $\xi_{ij}^k = x_{ij}^k = 0$
- if $\bar{c}_{ij}^k + \lambda_i^k - \lambda_j^k < 0$ but $\lambda_i^k - \lambda_j^k > 0$, then k is included in $\bar{K}$, $\bar{c}_{ij}^k = c_{ij}^k + \lambda_i^k - \lambda_j^k$ and the solution is $x_{ij}^k = \xi_{ij}^k = 1$
- if $\bar{c}_{ij}^k + \lambda_i^k - \lambda_j^k < 0$ but $\lambda_i^k - \lambda_j^k < 0$, then k is included in $\bar{K}$, $\bar{c}_{ij}^k = c_{ij}^k + |T_k|(\lambda_i^k - \lambda_j^k)$ and the solution is $x_{ij}^k = \xi_{ij}^k |T_k|$

By solving this knapsack problem, we are selecting the commodities to send along the arc (i,j), minimizing the cost of utilizing that arc.

A valid lower bound z_{LB} for the flow problem (24) - (29) can be obtained solving with a subgradient algorithm the following Lagrangian Dual:

$$z_{LB} = \max_{\lambda \geq 0} [z_{LR}(\lambda)] \tag{44}$$

3 Algorithm Implementation

In this chapter, we will present the different types of simulations performed, describing the reasons behind the code implemented in Python with the aid of solvers such as CPLEX.

The first approach to the problem described in this thesis was carried out using formulation 1, as described in Section 2.2, in a centralized manner. It is assumed that an abstract entity, aware of all the informations about the nodes and commodities involved, manages the sending of messages by publishers, the sending of requests by subscribers, and the routing by brokers in order to minimize the objective function (17). To do this, the CPLEX solver was used.

Subsequently, the problem was managed with formulation 2, described in Section 2.3, first with a centralized approach using the CPLEX solver. Then, the problem was relaxed in a Lagrangian way, using the subgradient approach (as described in Section 1.4.2). To do this, a function that solves knapsack problems and a heuristic that provides an upper bound for the subgradient algorithm were implemented.

Finally, the work focused on implementing the problem in a distributed manner. This means that each node makes decisions autonomously, calculating its penalties and requesting the others from neighboring nodes. Based on the availability of communicating nodes, commodities or requests are sent to the most cost-effective nodes in terms of the penalized edges.

For comparison purposes, an additional heuristic version was also developed for the distributed management of communications between nodes. This is based on an exchange of messages of requests and availability between nodes, in order to establish the most convenient communication for each node.

3.1 Centralized Version with Formulation 1

First of all, it is necessary to import the 'docplex' package in order to use the CPLEX solver in Jupiter.

```
from docplex.mp.model import Model
```

The instance data of the problem is available in .json files, so it is necessary to import them using the following command:

```
data = json.load(codecs.open(file, 'r', 'utf-8-sig'))
```

After the extraction and management of the data, the following are available (in parentheses are the names of the corresponding Python variables): the number of nodes (n) and commodities (nk) that make up the problem, the set of nodes (N), publishers (S), subscribers (T), and subscribers requesting the k-th commodity (Tk), the set of brokers (B) and the set of commodities (K), the cost matrices (Ck) and capacity matrices (Cap). The weight of the transmission of each commodity (a) and the coordinates of each node (coordinates). The lists of nodes to which each node can send (Lt) and from which it can receive (Lr), and the set of existing arcs (A).

The next step is to create an optimization model using the imported 'docplex' module and to define the decision variables x and ξ as initially empty.

```
opt_mod = Model("PubSub")
x = opt_mod.integer_var_cube(nk, n, n)
xi = opt_mod.binary_var_cube(nk, n, n, name ='xi')
```

Then, the various constraints described in Section 2.2 are defined.

Constraint (18):

```
for k in K:
    for t in Tk[k]:
        opt_mod.add_constraint(opt_mod.sum(x[k,j,t]
        for j in Lr[t])==1)
```

Constraint (19):

```
for k in K:
    for i in B:
        opt_mod.add_constraint(opt_mod.sum(x[k,j,i]
        for j in Lr[i]) == opt_mod.sum(x[k,i,j]
        for j in Lt[i]))
```

Constraint (20):

```
for k in K:
    for endp in A:
            opt_mod.add_constraint(len(Tk[k])*
            xi[k,endp[0],endp[1]] >= x[k,endp[0],
            endp[1]])
```

Constraint (21):

```
for i in N:
    for j in Lt[i]:
        opt_mod.add_constraint(opt_mod.sum(a[k] *
        (xi[k,i,j] + xi[k,j,i]) for k in K) <=
        Cap[i,j])
```

Constraint (22):

```
for k in K:
    for endp in A:
        opt_mod.add_constraint(x[k,endp[0],endp[1]]
        >= 0)
```

The constraint (23), which enforce the binary values of ξ_{ij}^{k}, do not need to be implemented as these variables have already been defined as binary in the previous step.

We define the objective function (17) as follows:

```
z = opt_mod.sum(opt_mod.sum(opt_mod.sum((Ck[k][j,i] *
x[k,j,i]) for j in Lr[i]) for i in N)for k in K)
```

It is communicated to the solver the intention of minimizing the objective function 'z' and the command is used to start its resolution:

```
opt_mod.set_objective('min',z)
opt_mod.print_information()
Sol = opt_mod.solve()
```

By specifying the objective function 'z' and the corresponding constraints, the solver to search for the optimal solution. Upon completion of the optimization process, the solver will return the value of the decision variables that correspond to the solution, satisfying the problem requirements and constraints. To extract the obtained decision variable values and bring them into matrix form, the following functions have been implemented:

```
def binExtract(n,nk,binary_vars):
    Xi = np.zeros((nk,n,n))
    for b in binary_vars:
        pos = re.findall(r'\d+',str(b))
        l = [int(j) for j in pos]
        Xi[l[0],l[1],l[2]] = b.solution_value
    return Xi
```

```
    def intExtract(n,nk,integer_vars):
        X = np.zeros((nk,n,n))
        for v in opt_mod.iter_integer_vars():
            pos = re.findall(r'\d+',str(v))
            l = [int(s) for s in pos]
            X[l[0],l[1],l[2]] = v.solution_value
        return X
```

The implemented functions allow to obtain tensors (matrices of matrices), where the index k identifies the solution matrix for the k-th commodity, the index i indicates the starting node, and the index j indicates the arrival node of the edges.

By implementing the optimization problem in this way, the solver has all the information of the given instance available. Therefore, a centralized approach is being used.

3.2 Continuous Relaxation of Formulation 1

To obtain the continuous relaxation of the optimization problem with integer variables described in Chapter 2.2, it is possible to define the decision variables as continuous instead of integer or binary. In this way, the integrality constraints can be relaxed, and the problem can be solved using standard optimization techniques, providing a lower bound on the optimal solution of the original problem. This technique is called "continuous relaxation" and is commonly used to deal with the computational complexity associated with solving problems with discrete variables.

```
    x = opt_mod.continuous_var_cube(nk, n, n)
    xi = opt_mod.continuous_var_cube(nk, n, n,name ='xi')
```

3.3 Centralized Version with Formulation 2

Formulation 2 represents a centralized optimization approach that aims to minimize the total cost of the necessary links to satisfy the demand of all subscribers. Unlike Formulation 1, Formulation 2 considers each link only once, regardless of the number of branches required to meet the demand of different subscribers. To implement Formulation 2, the programming language Python is still used, with the help of Jupyter as the development environment. The solver used to find the optimal solution is CPLEX, which is integrated into the implementation of the optimization model. After extracting

the relevant data from the instances in .json format, the optimization model is created, which includes capacity constraints, cost constraints, and the objective function that aims to minimize the total cost of the links. Then, the solver is launched to find the optimal solution to the optimization problem. The implementation of Formulation 2 requires a specific set of steps, such as identifying nodes and links, defining commodities, and specifying capacity and cost constraints.

Once the '*docplex*' package is imported into Python, the decision variables are defined and used to create the optimization model.

```
opt_mod = Model("PubSub")
x = opt_mod.integer_var_cube(nk, n, n)
xi = opt_mod.binary_var_cube(nk, n, n, name ='xi')
```

Then, the various constraints described at Section 2.3 are defined.

Constraint (25):

```
for k in K:
    for i in N:
        if i in S:
            opt_mod.add_constraint((
            opt_mod.sum(x[k,i,j] for j in Lt[i])
            - opt_mod.sum(x[k,j,i] for j in Lr[i]))
            >= b[k,i])
        else:
            opt_mod.add_constraint((
            opt_mod.sum(x[k,i,j] for j in Lt[i])
            - opt_mod.sum(x[k,j,i] for j in Lr[i]))
            == b[k,i])
```

Constraint (26):

```
for endp in A:
    opt_mod.add_constraint(opt_mod.sum(a[k]*
    xi[k,endp[0],endp[1]] for k in K) <=
    Cap[endp[0],endp[1]])
```

Constraint (27):

```
for k in K:
    for endp in A:
        opt_mod.add_constraint(xi[k,endp[0],endp[1]]
        <=  x[k,endp[0], endp[1]])
        opt_mod.add_constraint(x[k,endp[0], endp[1]]
        <= len(Tk[k])*xi[k,endp[0],endp[1]])
```

The constraints (28) and (29) ensure that x_{ij}^k are integers and ξ_{ij}^k are binaries. As the variables ξ_{ij}^k and x_{ij}^k have already been defined as binary and integer in the previous step, there is no need to implement these constraints.

The objective function (24) is defined as follows:

```
z = opt_mod.sum(opt_mod.sum((Ck[k][endp[0],endp[1]]*
xi[k,endp[0],endp[1]]) for endp in A) for k in K)
```

It is communicated to the solver the intention of minimizing the objective function 'z' and the command is used to start its resolution.

```
opt_mod.set_objective('min',z)
opt_mod.print_information()
Sol = opt_mod.solve()
opt_mod.print_solution()
```

The decision variable values are extracted and reformatted in a matrix form using the 'binExtract' and 'intExtract' functions defined at Section 3.1.

3.4 Continuous Relaxation of Formulation 2

The continuous relaxation of Formulation 2 is obtained by relaxing the integrality constraints on the decision variables x_{ij}^k and ξ_{ij}^k. As a result, the solver is now able to consider continuous values for these variables instead of being constrained to integer or binary values as in the original formulation. To achieve this relaxation, we modified the constraints in Formulation 2 by replacing the binary constraints on 'xi' with linear constraints that permit non-integer values of 'xi'. Similarly, we replaced the integer constraints on 'x' with linear constraints that permit non-integer values of 'x'. The process of continuous relaxation applied to Formulation 2 can offer several advantages. For instance, it enables us to obtain a lower bound on the optimal solution value of the original problem, which can be useful in validating the quality of any integer solutions obtained.

```
opt_mod = Model("PubSub")
x = opt_mod.continuous_var_cube(nk, n, n)
xi = opt_mod.continuous_var_cube(nk, n, n,name ='xi')
```

The implementation of the model, the definition of the constraints, and the definition of the objective function remain the same as in Section (3.3).

3.5 Heuristic Algorithm for Network Design Problems

A heuristic algorithm is a search and optimization technique used to solve complex computational problems based on empirical rules, experience, and learning ability. Unlike exact methods that guarantee optimal solutions but require high execution time, heuristic algorithms can provide an *acceptable* solution in relatively short time, although not necessarily optimal. A heuristic algorithm is based on a set of heuristic strategies, which are empirical or heuristic rules that allow efficient exploration of solution space. The effectiveness of a heuristic algorithm depends on the choice of heuristic strategies, the way these strategies are integrated into the algorithm, and their ability to efficiently explore solution space. Therefore, a heuristic algorithm has been developed to find acceptable solutions to the network design problems at hand. This allows us to find an Upper Bound for the optimization problem described by Formulation 2.

The implemented heuristic is based on finding the least cost path from publisher to subscriber for each commodity in the problem. This is implemented using Dijkstra's algorithm for defining the best path. To take into account the bandwidth occupation of the channels, an occupation matrix (O) is defined where the rows indicate the starting nodes and the columns the arrival node (therefore an arc) and each element defines the bandwidth occupied for via transmissions in that arc.

3.5.1 Overview of the Implemented Heuristic Algorithm

The points of reasoning underlying the heuristic are as follows:

1. Randomly Shuffling the List of Publishers.
   This guarantees that Dijkstra's algorithm finds paths starting from publishers in a different order each iteration, thus giving different priorities to the commodities in the bandwidth occupation of the edges

2. For each Pub (in the order given) and related commodity, Dijkstra's algorithm is performed.
   In doing so, a matrix is provided in which the path with the minimum cost from the publisher to the subscriber of the commodity concerned is obtained.

3. For each subscriber that requests the commodities generated by a give publisher, a starting and finishing point is defined.

   Initially the starting point is the broker who serves the subscriber requesting of the commodity and the arrival point is the subscriber himself. The bandwidth occupation for sending data in this arc is counted and if its capacity allows it, the arc is put into solution (ξ_{ij}^k=1) and the occupation matrix (O_{ij}) is updated with the weight of the commodity (a^k).

   After that, the end point is replaced with the previous starting point, and the new starting point will be the node serving the current end node according to the Dijkstra matrix. If the occupation of arc (i, j) allows it, this arc is put into solution.

   This process is repeated until the starting point is the publisher of the commodity we are considering.

4. The cost of the solution obtained is calculated and if this is lower than the minimum cost among the solutions previously formulated (out of 50 repetitions of the algorithm) it is saved in memory.

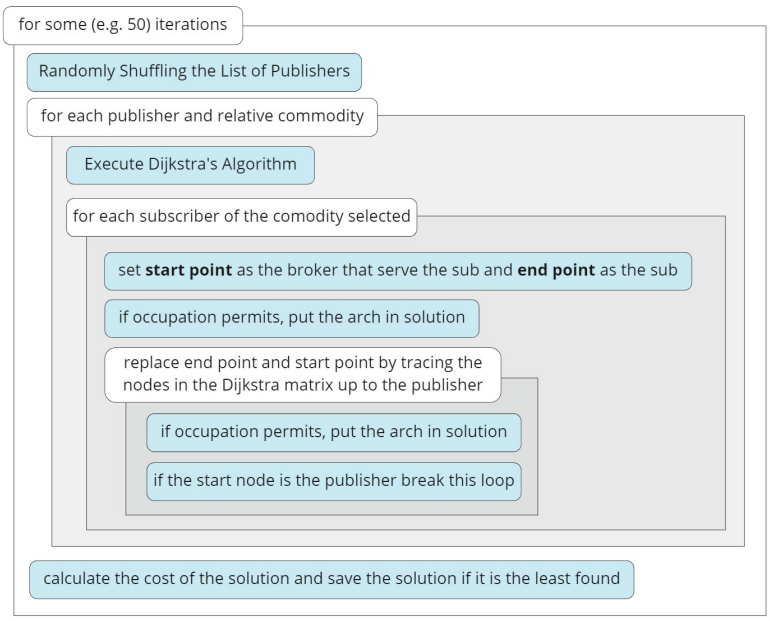

Figure 6: Heuristic Algorithm Structure

In order to understand the heuristic algorithm described in the previous paragraph, it is essential to provide an introduction to Dijkstra's algorithm and its implementation, since the heuristic algorithm is based on it.

Dijkstra's Algorithm

The objective of this algorithm is to find the shortest path between any two vertexes in a graph. As an illustration, consider the following example. Consider a problem consisting of 5 nodes, comprising a publisher (P), three brokers (B1, B2, B3), and a subscriber (S). The Figure 7 displays the corresponding network topology, where the costs of the existing arcs are indicated.

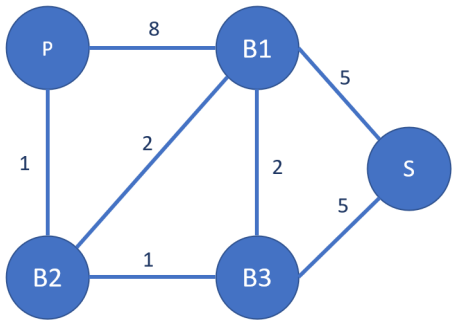

Figure 7: Network Example for Dijkstra's Algorithm - step 1

To begin with, let us assume that the distance from publisher A to itself is zero, while the distances to all other nodes are unknown. Next, we initialize the Dijkstra table by assigning high costs to reach all the other nodes, such as infinity (as shown in the Table 1).

Vertex	Shortest distance from A	Previous vertex
P	0	
B1	∞	
B2	∞	
B3	∞	
S	∞	

Table 1: Example of Realization of Dijkstra's Table - step 1

After initializing the table with vertex P and setting all other vertices as unvisited with infinite distance from P, we proceed to visit the unvisited nodes with the lowest distance from P. For each visited node, we update the

shortest distance in the table if it is lower than the current distance stored in the table. We also update the previous vertex field in the table to the current vertex if the new distance is shorter. After processing all the adjacent vertices of the current vertex, we remove it from the list of unvisited nodes and insert it into the list of visited nodes.

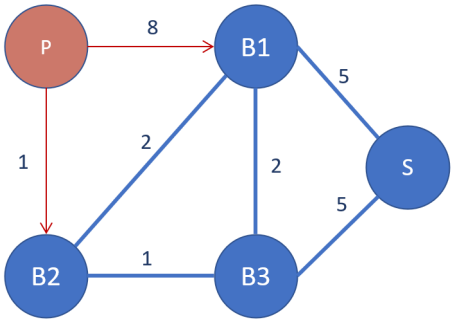

Figure 8: Network Example for Dijkstra's Algorithm - step 2

Vertex	Shortest distance from A	Previous vertex
P	0	
B1	**8**	P
B2	**1**	P
B3	∞	
S	∞	

Table 2: Example of Realization of Dijkstra's Table - step 2

Subsequently, the algorithm repeats starting from the node with the lowest distance among those not yet visited, in this case B2. Next, we visit the unvisited neighbors of B2, calculate their distance from the initial vertex (S), and save the data in the table if the new distances are lower than the previously written distances in the table.

Vertex	Shortest distance from A	Previous vertex
P	0	
B1	**3**	B2
B2	**1**	P
B3	**2**	B2
S	∞	

Table 3: Example of Realization of Dijkstra's Table - step 3

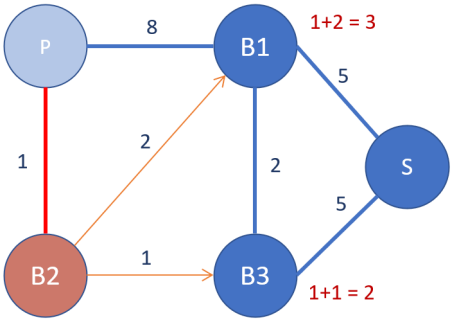

Figure 9: Network Example for Dijkstra's Algorithm - step 3

Proceeding in the same way, we obtain a table that indicates the smallest known distance from the starting vertex (S) for each node, along with the previous node from which one passes to reach it.

Vertex	Shortest distance from A	Previous vertex
P	0	
B1	3	B2
B2	7	B3
B3	1	S
S	2	B2

Table 4: Example of Realization of Dijkstra's Table - step 4

At this point, we have a table available that allows us to reconstruct the path up to the "S" vertex by tracing back through the "previous vertex" column.

3.5.2 Technical details of the code implementation

Dijkstra's Algorithm Implementation

The following Python function implements the previously described method for obtaining the Dijkstra matrix. In parentheses are the names of the corresponding Python variables. Its inputs include the number of nodes (n), the distances between nodes (distances), the starting node (start_node), the weight of each commodity (a), the occupation matrix (O), and the edge capacity (cap).

```
    def dijkstra(n,distances,start_node,a,O,cap):
        # dijkstra table initialization
        mi = 1e10
        Dijkstra = mi* np.ones((n,3),dtype=int)
        Dijkstra[:,0] = [i for i in range(n)]

        # unvisited nodes list initialization
        unvisited = [i for i in range(n) if i !=
        start_node]

        # define the neighbors of the node using the
        # 'find_neighbors' function
        unv_neig =
          find_neighbors(n,distances,unvisited,start_node)

        # update Dijkstra table with info of start_node
        # unvisited neighbors
        Dijkstra[start_node,1:] = 0
        for i in unv_neig:
            Dijkstra[i,1] = distances[start_node][i]
            Dijkstra[i,2] = start_node

        # recursively execute these instructions until
        # all nodes are visited
        while len(unvisited) >0:
            # least distant unvisted node
            min_d = np.min(Dijkstra[unvisited,1])
            # select the index of this node
            temp_D = Dijkstra[unvisited,:]
            pos = list(temp_D[:,1]).index(min_d)
            sel_node = int(temp_D[pos,0])
            # find the neighbors of the selected node
            unv_neig =
                find_neighbors(n,distances,unvisited,
                sel_node)

            # for each of these, calculate the distance
```

```
            # from the starting node and save it in the
            # Dijkstra table if it is less than those
            # found
                for i in unv_neig:
                d = distances[sel_node][i] +
                        Dijkstra[sel_node,1]
                if d < Dijkstra[i,1]
                and a + O[sel_node,i] <= Cap[sel_node,i]:
                    Dijkstra[i,1] = d
                    Dijkstra[i,2] = sel_node

            # remove the selected node from the list of
            # unseen nodes
            unvisited.remove(sel_node)

        return Dijkstra
```

The function 'find_neighbors', used within the 'dijkstra' function, takes the number of nodes, the distances, the list of unvisited nodes, and the starting node as inputs. Its output is the list of neighbors of the starting node given as input.

```
    def find_neighbors(n,distances,unvisited,s_node):
        mi = 1e10
        unv_neig = [i for i in range(n)
                    if distances[s_node][i] < mi
                    if i in unvisited if i != s_node]
        return unv_neig
```

Heuristic Algorithm Implementation

The heuristic algorithm was developed following the steps described in the following:
It initializes a matrix with commodity indices in the first row and corresponding values in the second row.

```
    s_c = np.zeros((2,len(S)),dtype = int)
    s_c[0,:] = S
    s_c[1,:] = K
    ls = len(S)
    # vectors that go from 0 to numer of sensors
    rl = list(range(ls))
```

Then, it initializes the minimum cost and the number of iteration of external cycle.

```
    min_cost = 1e10
    num_iter = 50
```

For each of the '*num_iter*' iterations, the algorithm shuffles the order of sensors, reinitialize the solution variables and the occupancy matrix, and resets the indicator for unfeasible solutions and the commodity index to 0.

```
    for i in range(num_iter):
        # shuffle the order of the sensors
        random.shuffle(rl)
        s_c = s_c[:,rl]
        sensor = s_c[0,:]
        com = s_c[1,:]
        # initializzation of solution variables and
        # occupation matrix
        Xi = np.zeros((nk,n,n))
        X = np.zeros((nk,n,n))
        O = np.zeros((n,n))
        # indicator that signals the possible
        # unfeasibility of the proposed solution
        unfeas = 0
        # commodity index
        c = 0
```

The algorithm executes Dijkstra's algorithm for each sensor in the order established in the previous line.

```
    for s in sensor:
        k = com[c]
        D = dijkstra(n,Ck[k],s,a[k],O,Cap)
```

For each commodity associated with the considered sensor, it defines a starting point and an ending point by retracing the Dijkstra's table as described in Section 3.5.1, until the start point is the sensor corresponding to the commodity being considered. If the arc occupancy permits it, it includes the arc in the solution.

```
    for t in Tk[k]:
        # define start node and end node
        strt = int(D[t,2])
        end = t

        while True:
            if O[strt,end] == 0:
                O[strt,end] += a[k]
            if O[strt,end] <= Cap[strt,end]:
                Xi[k,strt,end] = 1
                X[k,strt,end] += 1
            if strt == s:
                break
            else:
                end = strt
                strt = int(D[end,2])
```

It checks the unfeasibility of the solutions.

```
    if  sum(Xi[k,:,t])==0 :
    unfeas = 1
    break
```

Then, the algorithm computes the cost of the obtained solution and save it if it is the minimum cost found so far.

```
    if unfeas == 0:
        costo = cost2(K,A,Ck,Xi)
        if costo <= min_cost:
            min_cost = costo
            X_best = X
            Xi_best = Xi
```

3.6 Lagrangian Relaxation of Formulation 2

This chapter aims to analyze the Lagrangian relaxation approach adopted to solve the complex optimization problem described in Formulation 2. In particular, to obtain a valid lower bound, we use a Lagrangian relaxation technique that simplified the original problem by imposing a flow constraint on commodities, as described in Section 2.3.1. To implement this technique, we use the subgradient method to optimize the Lagrangian function and update penalties. This section provides a detailed description of the developed Lagrangian relaxation algorithm, giving a complete explanation of how each component works.

The following text describes the implementation of an algorithm that solves an optimization problem by combining heuristic linear programming techniques and integer linear programming. In the first step, the problem variables are initialized, and an upper bound is calculated using a heuristic algorithm. Subsequently, the integer linear programming problem is solved using the Lagrangian technique, and the obtained result is used to update the lower bound. The subgradient is then calculated, and the penalties are updated, to repeat the problem-solving process until an optimal solution is reached. During the algorithm, a knapsack problem is also solved to determine which commodities to transport along each network arc.

Step 1 - Variable initialization

We initialize the following values (in parentheses are the names of the corresponding Python variables): penalties λ_i^k to zero, the maximum number of iterations for the algorithm's execution (it_max), a counter required for updating the alpha parameter (it_count), a lower bound set to an extremely low value, the α parameter (used to update the penalties values as shown in equation 46), and solution variables x (X) and ξ (Xi). Additionally, we define the value of the UpperBound using the heuristic algorithm described in Section (3.5).

```
pnlty = np.zeros((nk,n))
it_max = 100
it_count = 0
LB = -1e20
# heuristic algorithm to define an upper-bound
UB ...
alfa = 0.1
X = np.zeros((nk,n,n))
Xi = np.zeros((nk,n,n))
```

Step 2 - Solve the Lagrangian problem and update the lower bound
The value $z_{LR}(\lambda)$ of the Lagrangian problem $LR(\lambda)$ is given by equation (45), described in Section 2.3.1, solving the problems $z^k_{LR_{ij}}(\lambda)$. If the resulting value exceeds the lower bound, it is updated.

$$z_{LR}(\lambda) = \sum_{(i,j)\in A} z^k_{LR_{ij}}(\lambda) - \sum_{k\in K}\sum_{i\in N} \lambda^k_i b^k_i \tag{45}$$

For each iteration of the algorithm, a new variable is initialized to store the solution obtained in that specific iteration, as well as the first (*zLRij_sum*) and second terms (*term2*) of equation (45) and the subgradient.

```
    for it in range(it_max):
        x = np.zeros((nk,n,n))
        xi = np.zeros((nk,n,n))
        subgrad = np.zeros((nk,n))
        zLRij_sum = 0
        term2 = 0
```

The objective function is calculated as described in Section (2.3.1). For each arc of the graph, the cost is updated (as illustrated on the last page of Section 2.3.1).

```
    for endp in A:
        c = []
        K_ = []
        a_ = []
        zLRij = 0
        nk_ = 0
        updt = 0
        for k in K:
            ci = Ck[k][endp[0],endp[1]]
            dif_pnlty = pnlty[k,endp[0]]-pnlty[k,endp[1]]
            if (ci + dif_pnlty) < 0:
                K_.append(k)
                a_.append(a[k])
                nk_ += 1
                if dif_pnlty > 0:
                    c_new = ci + dif_pnlty
                    c.append(c_new)
                    updt = 1
                else:
                    c_new = ci + len(Tk[k])*dif_pnlty
                    c.append(c_new)
                    updt = 2
```

At this stage, a knapsack problem needs to be solved for each arc to determine which commodities to transport along it. To accomplish this,

the knapsack objective function is implemented. It is important to note that, for the knapsack algorithm, costs are initially negated so that they are interpreted as benefits. This is necessary because the algorithm is designed to maximize the total value of benefits, i.e. the sum of benefits of the selected items, while taking into account the maximum capacity constraint of the knapsack. The knapsack function is described later in Section (3.6.1).

```
W = int(Cap[endp[0],endp[1]])
c_meno = [ -x for x in c]
out = KnapSack(nk_,W,c_meno,a_)
```

The values of Xi and X are assigned according to the solution proposed by the knapsack problem.

```
for i in range(nk_):
    if out[i] == 1:
        xi[K_[i],endp[0],endp[1]] = 1
        if updt == 1:
            x[K_[i],endp[0],endp[1]] = 1
        elif updt ==2:
            x[K_[i],endp[0],endp[1]] = len(Tk[K_[i]])
```

Now that we have the necessary values, we can compute the first term by summing the ZLRij terms (refer to formula 45). The costs, which were initially negated to be interpreted as benefits by the knapsack algorithm, are now negated again to return to being considered as costs.

```
j = 0
for k in K_:
    zLRij -= c_meno[j]*xi[k,endp[0],endp[1]]
    j +=1
zLRij_sum += zLRij
```

Next, we evaluate the second term and the objective function using the following code:

```
for k in K:
    for i in N:
        term2 += pnlty[k,i]*b[k,i]
zLR = zLRij_sum - term2
```

Once we have computed the objective function, we update the lower bound as follows:

```
if zLR > LB:
    LB = zLR
else:
    it_count += 1
```

Step 3 - Subgradient calculation and penalty update

The subradient is calculated as follows:

```
sum1 = 0
sum2 = 0
for k in K:
    for i in N:
        for j in Lt[i]:
            sum1 += x[k,i,j]
        for j in Lr[i]:
            sum2 += x[k,j,i]
        subgrad[k,i] = sum1 - sum2 - b[k,i]
```

Once the norm of the subgradient is computed, we can update the penalty using a step defined by the parameter α, the objective function value obtained, and the norm of the subgradient (see equation 46).

$$step = \alpha \frac{UB - Z_{LR}}{||s||^2} \tag{46}$$

$$\lambda_k^i = \lambda_k^i + step \cdot s_k^i \tag{47}$$

```
for k in K:
    norm_s = np.linalg.norm(subgrad[k,:])
    step = alfa * (UB - zLR)/ norm_s**2
    for i in N:
        pnlty[k,i] = pnlty[k,i] + step*subgrad[k,i]
```

If no better solutions are found for a certain number of iterations (e.g., 20), the algorithm reduces the value of the α parameter by halving it.

```
check_alfa = 0
if it_count == 20:
    alfa = alfa/2
    it_count = 0
    check_alfa = 1
```

3.6.1 Knapsack Problem and Function Implementation

The knapsack problem is a combinatorial optimization problem that involves finding the combination of objects with the highest total value that can fit into a knapsack with a predefined maximum capacity. The problem is defined by a set of objects, each with an associated value and weight and the knapsack's capacity limit. The objective is to select the objects in a way that maximizes the total value of the selected objects without exceeding the knapsack's capacity limit. As a result of its complexity, the knapsack problem is classified as NP-hard, meaning that no polynomial algorithm can solve it efficiently for all instances of the problem.

To solve to optimality the kanpsack problem we can use the dynamic programming. Dynamic programming is a problem-solving technique that involves breaking a problem down into smaller subproblems, solving each subproblem only once, and then using the solutions of the subproblems to solve the original problem. In the case of the knapsack problem, dynamic programming is used to construct a table, called the knapsack table, that represents the optimal solution for each subproblem. The knapsack table has rows representing the objects that can be selected and columns representing the weight capacity of the knapsack. The values in the table represent the maximum value that can be obtained using the objects up to a certain weight capacity. By solving each subproblem only once, dynamic programming avoids redundant computations and greatly reduces the time complexity of the problem.

Suppose we have $n = 4$ objects and a bag capacity of $W = 8$. Each object has a weight (e.g., $w = [2, 3, 4, 5]$) and a priority (e.g. $p = [1, 2, 5, 6]$). Our goal is to fill the bag while ensuring that the total maximum profit is maximized, as shown in equation (48).

$$\max \sum_{i=1}^{n} p_i x_i \tag{48}$$

We also need to ensure that we do not exceed the bag's capacity:

$$\sum_{i=1}^{n} w_i x_i \leq W \tag{49}$$

To solve this problem, we can use a dynamic programming approach. We create a table with rows from 0 to the number of objects (e.g., 4) and columns from 0 to the bag's capacity (e.g., 8). In the first row, which represents the

case where no object is included in the solution, the profit in each column is zero. The same is true for the first column, which represents the case where the bag's capacity is zero. Moving on to the second row, which represents the case where only one object is included in the solution, we fill in the profit value for that object (e.g., $p_1 = 1$) when the capacity is equal to or greater than its weight (e.g., 2). The profit for all other columns in this row remains zero, as only one object is included. Next, we consider the third row, which represents the case where two objects are included in the solution. We fill in the profit value for the second object (e.g., $p_2 = 2$) when the capacity is equal to or grater then its weight (e.g. 3). For columns before this one, the profit values are the same as those in the row above. When the bag's capacity is 5, we can include both the first and second objects, resulting in a total profit of 3. The profit values for all other columns in this row remain 3, as no additional objects can be included. We continue this process to fill in the entire table, using the following formulas:

$$if\ w \geq w_i : \qquad V_{i,w} = \max[V_{i-1,w}, V_{i-1,w-w_i} + p_i] \tag{50}$$

$$else\ if\ w < w_i : \qquad V_{i,w} = V_{i-1,w} \tag{51}$$

Here, $V_{i,w}$ represents the element in the table at row i and column w, w_i represents the weight of the ith object, and p_i represents its priority. We use this formula to determine the maximum profit we can achieve by either excluding the ith object (which is represented by $V_{i-1,w}$) or including it (which is represented by $V_{i-1,w-w_i} + p_i$).

By filling in the entire table using this approach, we can determine the maximum profit we can achieve while ensuring that we do not exceed the bag's capacity.

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3
3	0	0	1	2	5	5	6	7	8
4	0	0	1	2	5	6	6	7	8

Table 5: Example of Knapsack table

Using the knapsack problem table, we can identify the maximum possible profit, which is located in the last cell of the table and is equal to 8. This

value can only be generated in the fifth row of the table (i.e., by including 4 objects in the bag). Therefore, in the solution vector, we set $x_4 = 1$ to indicate that the last object is included in the knapsack. Then, we subtract the value of the fourth object (i.e., 6) from the total profit and obtain the new profit value of 2. To identify the objects that contribute to the profit of 2, we need to go back in the table to the row where this value appears. The row in question is the third one, which depends on the insertion of object 2. Therefore, we can set $x_2 = 1$ in the solution vector. In conclusion, the optimal solution is represented by the vector x = [0 1 0 1], which indicates that the second and fourth objects are included in the knapsack, while the other objects are excluded.

The Knapsack problem can be solved using a function called 'KnapSack', which is implemented as follows: The function takes in several inputs, including the number of commodities, the bag capacity (W), the profit of each commodity (c), and the weight of each commodity (a).

```
    def KnapSack(nk_,W,c,a_):

        # build knapscak table
        c = [0,*c]
        a_ = [0,*a_]
        M = np.zeros((nk_+1,W+1))
        for n_com in range(nk_+1):
            for cap in range(W+1):
                if n_com == 0:
                    M[n_com,cap] = 0
                elif a_[n_com] <= cap:
                    M[n_com,cap] = max(c[n_com]+M[n_com - 1,
                                       cap - a_[n_com]],
                                       M[n_com - 1,cap])
                else:
                    M[n_com,cap] = M[n_com - 1,cap]

        # build solution
        cout = np.zeros(nk_+1,dtype=int)
        s = M[nk_,-1]
        for i in range(nk_,-1,-1):
            if s not in M[i-1,:]:
                cout[i] = 1
                s = s - c[i]
        cout = cout[1:]

        return cout
```

3.7 Distributed Lagrangian Heuristic

The distributed version of the algorithm is based on the Lagrangian relaxation with the subgradient method, as described in the Section 3.6. In this approach, each node only has information about itself, namely its penalties and the commodities it possesses or needs, and exchanges this information with neighboring nodes. The principle behind the method is as follows:

- Each node calculates its own **penalties** and keeps track of the penalties of other nodes.
- **Subscribers** send requests for commodities to brokers based on the penalized weight.
- **Publishers** that own commodities send them to the most convenient brokers based on the penalized weight of the various connections.
- Regarding **brokers**, two alternative options have been proposed and implemented:
    1. In the first case, if brokers receive requests from subscribers or other brokers and do not have the requested commodity, they act as requesters until they can make a request to nodes with availability, creating a link between requesting subscribers and emitting publishers. Otherwise, if they have the requested commodity, they send it.
    2. In the second case, brokers that receive requests act as requesters themselves, sending requests to the most convenient nodes. If they have availability, they still try to push the commodity into the network to the most convenient nodes, hoping to find a link between the publisher and subscriber of the commodity.
- The nodes exchange the penalty calculated between neighbors.

Before fully explaining the functioning of the algorithms created to manage requests and the transmission of commodities, it is important to define the communication methods used by nodes and the functions developed to facilitate them. To ensure efficient communication among nodes, several methods of information exchange have been implemented, including transmission of request and response messages, signaling the occupancy status of connections, and updating the availability of commodities. Additionally, to simplify communication and information management, several functions such as "r_connect", "s_connect", "b_connect", and "b_push" have been created

to manage requests and transmission of commodities and to identify the best connection options among nodes.

Lagrangian Penalty Update

For the penalty update by the nodes, the function 'pnlty_update' is used. This function requires several inputs, including the node to consider (node), the penalties already known by the node (pnlty_node), the alpha parameter for the penalty update, the Lagrangian solutions x and xi, the set of edges (A), the set of commodities (K), the set of nodes (N), the set of requesters (Tk), the set of nodes to which the considered node can send (Lt) and the set of nodes from which it can receive (Lr), the original cost matrix (Ck), the weight of the commodities (a), the parameter b, and the edge capacities (Cap). Then, the function uses the solution formula by relaxing the sub-problem related to the considered node, as indicated in the Section 2.3.

```
    def pnlty_update(node, pnlty_node, alfa, x, xi, A, K, N,
                     Tk, Lt, Lr, Ck, a, b, Cap):
    zLRij_sum = 0
    term2 = 0
    for endp in A:
        if endp[0] == node:
            c = []
            K_ = []
            a_ = []
            zLRij = 0
            nk_ = 0
            updt = 0
            for k in K:
                ci = Ck[k][endp[0],endp[1]]
                dif_pnlty = pnlty_node[k][endp[0]] -
                            pnlty_node[k][endp[1]]
                if (ci + dif_pnlty) < 0:
                    K_.append(k)
                    a_.append(a[k])
                    nk_ += 1
                    if dif_pnlty > 0:
                        c_new = ci + dif_pnlty
                        c.append(c_new)
                        updt = 1
                    else:
                        c_new = ci + len(Tk[k])*dif_pnlty
                        c.append(c_new)
                        updt = 2

            # knapsack for edge ij
            W = int(Cap[endp[0],endp[1]])
```

```
            c_meno = [ -x for x in c]
            out = KnapSack(nk_,W,c_meno,a_)

            # build xi with knapsack
            for i in range(nk_):
                if out[i] == 1:
                    xi[K_[i],endp[0],endp[1]] = 1
                    if updt == 1:
                        x[K_[i],endp[0],endp[1]] = 1
                    elif updt == 2:
                        x[K_[i],endp[0],endp[1]] =
                        len(Tk[K_[i]])

            j = 0
            for k in K_:
                zLRij -= c_meno[j]*xi[k,endp[0],endp[1]]
                j +=1
            zLRij_sum += zLRij

    for k in K:
        for i in N:
            term2 += pnlty_node[k][i]*b[k,i]

    # objective function
    zLR = zLRij_sum - term2

    # subgradient
    sum1 = 0
    sum2 = 0
    for k in K:
        for j in Lt[node]:
            sum1 += x[k,node,j]
        for j in Lr[node]:
            sum2 += x[k,j,node]
        subgrad = sum1 - sum2 - b[k,node]
        step = alfa*subgrad

        # penalty update
        pnlty_node[k][node] = pnlty_node[k][node] + step

    return pnlty_node, x, xi, zLR
```

Each node iteratively calculates its own penalties and generates solutions for the Lagrangian sub-problem of the connected edges through knapsack problem solving. As it is a distributed algorithm, subgradient norm cannot be calculated, and thus, penalties are updated using a constant step approach, determined by multiplying the alpha parameter with the component of the

subgradient relative to node i.

$$step = \alpha \cdot s_i^k \tag{52}$$

The penalty update can be improved by using the "quasi-constant" step rule. The principle is simple, if the solution improves α increases, if it doesn't improve for a certain number of iterations α decreases.

Request of Commodities by Subscribers

Thanks to the available penalties, each node can determine the most convenient connection to make a request or send commodities. Subscribers can request commodities using the 'r_connect' function, which requires as input the interested node (node), the commodity to request (k), the cost matrix (Ck), the set of nodes from which the considered node can receive (Lr), the weight of the commodities (a), the connection capacities (Cap), the set of sub-requesters Tk, the penalties known by the node (pnlty), the requests (req), and the availabilities (disp) of the nodes, as well as the solution Xi.

Firstly, the function collects information about the neighbors, such as the original cost of the connection, their availabilities, and their requests. Then, the cost is penalized using the penalties known by the node.

```
    neighborhood = {}
    cost = Ck[k][:,node]
    for neig in Lr[node]:
        diff_pnlty = pnlty[k][neig]- pnlty[k][node]
        info = {neig: [(cost[neig]+ diff_pnlty), req[k,neig],
                disp[k,neig]]}
        neighborhood.update(info)
```

Next, the function considers all connections of the node for the commodity of interest, initially setting them to zero and taking into account the occupation of nearby channels, which considers any other commodities sent over the same channels.

```
    precedent = []
    for i in N:
        if Xi[k][i,node] == 1:
            precedent.append(i)
    if precedent != []:
        for pre in precedent:
            Xi[k][precedent,node] = 0
```

```
            req[k,precedent] = 0

    # occupation of edges
    O = np.zeros((n,n))
    for key in neighborhood.keys():
        for com in K:
            O[key,node] += Xi[com][key,node]*a[com]
                           + Xi[com][node,key]*a[com]
            O[node,key] = O[key,node]
```

The function prioritizes nodes with negative penalized weight for the connection since they are advantageous and which have availability of the requested commodity. Next in priority are nodes with negative penalized weight but no availability, followed by nodes with positive penalized weight and commodity availability. Lastly, nodes with positive penalized weight and no availability are considered. Once the function identifies the most suitable node for the request, it updates the solutions by activating the relevant connection.

```
    n_check = 0; nd_check = 0; pd_check = 0; p_check = 0;
    n_id = {'id':[],'w':[]}; nd_id = {'id':[],'w':[]}
    pd_id = {'id':[],'w':[]}; p_id = {'id':[], 'w':[]}
    for key,val in neighborhood.items():
        # positive weight with disponibility
        if  val[0] <= 0 and val[2] == 1
        and O[key,node] + a[k] <= Cap[key,node]:
            nd_check = 1
            nd_id['id'].append(key)
            nd_id['w'].append(val[0])
        # negative weight without disponibility
        elif val[0] <= 0
        and O[key,node] + a[k] <= Cap[key,node]:
            n_check = 1
            n_id['id'].append(key)
            n_id['w'].append(val[0])
        # positive weight with disponibility
        elif val[2] == 1
        and O[key,node] + a[k] <= Cap[key,node]:
            pd_check = 1
            pd_id['id'].append(key)
            pd_id['w'].append(val[0])
        # positive weight without disponibility
        elif O[key,node] + a[k] <= Cap[key,node]:
            p_check = 1
            p_id['id'].append(key)
            p_id['w'].append(val[0])
```

```
    if nd_check == 1:
        connect_to = mindict(nd_id)
        connect_to = random.choice(connect_to)
    elif n_check == 1:
        connect_to = mindict(n_id)
        connect_to = random.choice(connect_to)
    elif pd_check == 1:
        connect_to = mindict(pd_id)
        connect_to = random.choice(connect_to)
    else:
        if p_id['id'] != []:
            connect_to = mindict(p_id)
            connect_to = random.choice(connect_to)
        else:
            no_choice = 1
            connect_to = None
    # update solution
    if no_choice == 0:
        Xi[k][connect_to,node] = 1
        req[k,connect_to] = 1
```

The outputs of the r_connect function include the updated Xi solution, the updated requests and availabilities of the nodes, the index of the node to which the request was made, and the dictionary of neighbors with their respective availabilities, requests and weights.

Commodity Sending by Publisher

The function 's_connect' is used to handle the sending of commodities by publishers. It takes in several parameters (name of the variables indicated in brackets), such as the interested node (node), the requested commodity (k), the cost matrix (Ck), the set of nodes to which the node can send (Lt), the weight of the commodity (a), the capacity of the connections (Cap), the set of sub-requesters (Tk), the penalties known by the node (pnlty), the requests (req) and availabilities (disp) of the nodes and the solution Xi. After generating a dictionary containing information about the neighbors as it follows:

```
    neighborhood = {}
    cost = Ck[k][node,:]
    for neig in Lt[node]:
        diff_pnlty = pnlty[k][node]- pnlty[k][neig]
        info = {neig: [(cost[neig]+ diff_pnlty), req[k,neig],
                disp[k,neig]]}
        neighborhood.update(info)
```

To find new connections, the algorithm cancels previous connections and searches for new ones, while taking into account the current occupancy of the arcs. To choose which nodes to connect, the algorithm maintains a priority hierarchy. First, nodes with negative penalized weight and commodity request are selected. Next, nodes with negative penalized weight but no request are chosen. After that, nodes with positive penalized weight and commodity request are considered. Finally, nodes with positive penalized weight but no requests are chosen. Since the 's_connect' function has a similar structure to the 'r_connect' function, we won't include the specific code text for 's_connect'.

Request of Commodities by Brokers

To handle requests from brokers, the algorithm uses the "b_connect" function, which is similar to "r_connect" and "s_connect". This function requires several inputs, including the node of interest (node), the requested commodity (k), the cost matrix (Ck), the set of nodes from which the considered node can receive the commodity (Lr), the commodity weights (a), the connection capacities (Cap), the set of sub-requesters (Tk), the node penalties (pnlty), the requests (req), the availabilities (disp) of nodes, as well as the solution Xi.

To fulfill the requests, the broker first evaluates the neighborhood information, then the penalized weight of the connections and the availabilities and requests of adjacent nodes. Next, it evaluates the best connection option based on the penalized weight and availability, following a priority hierarchy: it starts with nodes that have negative penalized weight and availability of the requested commodity. If there are none, it moves to those with negative penalized weight but no availability, then to those with positive penalized weight and availability, and finally to those with positive penalized weight but no availability.

Commodity Sending by Brokers

As mentioned earlier, we also considered the case where brokers with available commodities but no requests try to push them into the network by sending them to the node with the most convenient penalized weight. The functioning principle is similar to that of the other described functions: after an initial phase of collecting information from neighboring nodes, the most convenient node to send the commodity to is selected, prioritizing nodes with negative penalized weights and requests for the commodity in possession of the sending broker. This has been implemented through function 'b_push'.

Penalty Exchange between Nodes

Each node calculates its own Lagrangian penalties and updates the penalized costs based on its own penalties and those of neighboring nodes. This information is obtained periodically by requesting neighboring nodes to send their penalties through the 'pnlty_exchange' function. It should be noted that each node maintains a record of penalties for adjacent nodes, calculating its own penalties and requesting penalty information from neighboring nodes to update its records.

```
    def pnlty_exchange(N,Lt,pnlty_register):

        old_register = deepcopy(pnlty_register)
        for node in N:
            for key,val in pnlty_register[node].items():
                for v in Lt[node]:
                    pnlty_register[node][key][v] =
                    old_register[v][key][v]
        return pnlty_register
```

Other useful Function

The function 'satisub' is used to verify that every subscriber has received the requested commodities. It takes as input a set of subscribers 'K', a dictionary of requested commodities for each subscriber 'Tk' and a matrix 'disp' that indicates which commodities have been dispatched to each subscriber. The function iterates over all subscribers and their requested commodities, checking if each requested commodity has been dispatched to the corresponding subscriber. If any requested commodity is missing, the function returns 0 indicating that at least one subscriber is unsatisfied. Otherwise, it returns 1 indicating that all subscribers are satisfied.

```
    def satisub(K,Tk,disp):
    sub_satisfaction = 1
    for k in K:
        for t in Tk[k]:
            if disp[k,t] == 0:
                sub_satisfaction = 0
    return sub_satisfaction
```

The function 'mindict' is used to find the node with the minimum penalized cost for connection in a dictionary that has the format {'id node': [], 'weigh': []}. It takes as input a dictionary and returns a list of node IDs with the minimum weight. The function first extracts the list of weights and finds the minimum weight. It then finds the indices of all weights equal to the minimum weight and returns the corresponding node IDs.

```
    def mindict(dictionary):
        id_min = []
        w = list(dictionary['w'])
        min_w = min(w)
        indice_minimo = [i for i, x in enumerate(w)
                         if x == min_w]
        list_id = list(dictionary['id'])
        for i in indice_minimo:
            id_min.append(list_id[i])
        return id_min
```

3.7.1 Distributed Lagrangian Heuristic - First Approach

In this version of the distributed Lagrangian heuristic algorithm, brokers only act as requesters of commodities. Subscribers request commodities from brokers, who send them if available, otherwise they become requesters themselves and send requests to other brokers. Publishers send their commodities to nodes with the most advantageous penalized weight of links. Brokers that have received commodities wait for them to be requested and do nothing until then. A loop (while True) is therefore imposed that iteratively searches for solutions until interruption.

The process of node analysis occurs iteratively:
For a certain number of iterations, each node updates its Lagrangian penalties using the "pnlty_update" function.

```
for it in range(it_max_pnlty):
pnlty_node,x,xi,zLR = pnlty_update(node, pnlty_node,
alfa, x, xi, A, K, N, Tk, Lt, Lr, Ck, a, b, Cap)
```

In particular:

- Subscribers send their requests trying to find the best possible connection based on known penalized costs thanks to Lagrangian penalties, through the "r_connect" function.
- The publisher pushes commodities into the network towards the most convenient node from the Lagrangian penalties point of view, through the "s_connect" function.
- If the node is an intermediary with commodities requests, it becomes a requester and sends requests to other intermediaries through the "b_connect" function until it contacts a node with availability.

The brokers' activity is put in a loop of some iterations to prioritize their work. This allows the network to find connections without being continuously interrupted by the choices of publishers or subscribers.

```
# first the subscribers send request
for node in T:
    ...calculate penalty with "pnlty_update"

    for k in K:
        if node in Tk[k]:
            Xi, req, disp, connect_to, neighborhood =
            r_connect(node, k, Ck, Lr, a, Cap, Tk,
            pnlty_node, req, disp, Xi)

# publishers send commodities
k = 0
for node in S:
    ...calculate penalty with "pnlty_update"

    Xi, req, disp, connect_to, neighborhood =
    s_connect(node, k, Ck, Lt, a, Cap, Tk,
    pnlty_node, req, disp, Xi)
    k+=1

# brokers send request
for it_b in range(5):
    for node in B:
        ...calculate penalty with "pnlty_update"
        for k in K:
            if req[k,node] == 1 and disp[k,node] == 0
            and (all(xu==0 for xu in Xi[k][:,node])
            or it_updt%n==0):
                Xi, req, disp, connect_to, neighborhood =
                b_connect(node, k, Ck, Lr, a, Cap, Tk,
                pnlty_node, req, disp, Xi)
```

Subsequently, penalties are exchanged between nodes and the memories of the nodes that keep track of penalty for the calculation of weights are updated.

```
    pnlty_register = pnlty_exchange(N,Lt,pnlty_register)
```

It is checked if subscribers have availability of the requested commodities.

```
    sub_satisfaction = satisub(K,Tk,disp)
```

The image below summarizes some of the main steps of the reasoning behind the algorithm.

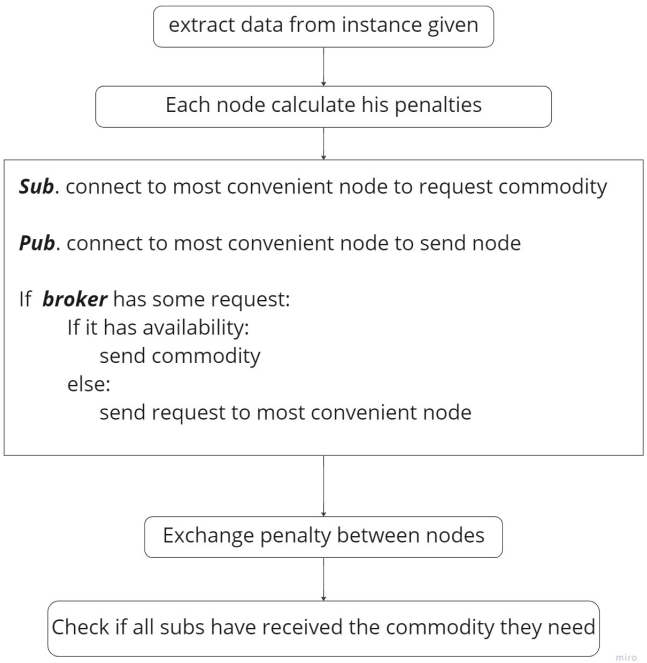

Figure 10: Description of First Approach of the Distributed Lagrangian Heuristic Algorithm

3.7.2 Distributed Lagrangian Heuristic - Second Method

The second approach was implemented as an improvement over the first to address issues that arise in large-scale instances, as illustrated in Section 4.4. The main difference from the first approach is that brokers, even if they possess a commodity for which there is no request, still insert it into the network using the "b_push" function. This function selects the most convenient node based on Lagrangian penalties, allowing for efficient and optimal distribution of commodities within the network. Another difference in this approach is that not all nodes are sequentially processed at each iteration, but only those with requests or availability of commodities are considered. Only in case of difficulty in the network's ability to adapt to the problem at hand, the network is reset. The threshold of iterations that determines this reset is set based on the capacity of the fundamental arcs of the network, which can reach a saturation condition, making it difficult to obtain acceptable solutions.

The "pnlty_update" function is used by each node to update its Lagrangian penalties for a certain number of iterations.

```
    for it in range(it_max_pnlty):
        pnlty_node,x,xi,zLR = pnlty_update(node, pnlty_node,
        alfa, x, xi, A, K, N, Tk, Lt, Lr, Ck, a, b, Cap)
```

Subsequently, nodes are iteratively analyzed:

- The subscribers attempt to establish the best possible connection using known penalized costs and Lagrangian penalties through the "r_connect" function.
- The publishers utilize the "s_connect" function to push commodities into the network towards the most favorable node based on Lagrangian penalties.
- If the node is a broker with commodity requests, it sends available commodities, and if not, it becomes a requester and sends requests to other brokers using the "b_connect" function.
- If a node is a broker with available commodities and no request, it utilizes the "b_push" function to send commodities to neighboring nodes.

The priority of brokers in creating connections within a network is a crucial aspect for the proper functioning of the system. In order for the network to find the most relevant and useful connections, it is necessary for brokers

to be able to perform their activity without being constantly interrupted by the choices of publishers or subscribers. To reduce the execution time for large instances, an approach is adopted to select the nodes involved in the transmission of commodities. In this way, not all nodes are considered at each iteration, simplifying the execution process.

```
    # for every node and every commodities
    for node in N:
        for k in K:
            # if it has requests, no availability
            # and there are no connection already made
            if req[k,node] == 1 and disp[k,node]== 0
            and all(xu==0 for xu in Xi[k][:,node]):
            # if it has availability, no request
            # and there are no connection already made
                node_list.append(node)
            elif req[k,node] == 0 and disp[k,node]== 1
            and all(xu==0 for xu in Xi[k][node,:]):
                node_list.append(node)
```

In the process of managing commodities, nodes with availability or requests for such resources are considered. Initially, the process starts from the subscribers, who send their requests. Subsequently, the publishers push the commodities to brokers considered most convenient in terms of the penalized cost of connections. Then, if brokers have received requests, they proceed with requesting commodities from other nearby nodes. However, if they have commodities available but have not yet received requests, they still try to distribute them in the network, hoping that requesting brokers and sending brokers will intersect and conclude the exchange of commodities.

```
for node in node_list:
    for k in K:
        # if the node is a subsribers, it sends requests
        if node in Tk[k]:
            Xi, req, disp, connect_to, neighborhood =
            r_connect(node, k, Ck, Lr, a, Cap, Tk,
            pnlty_node, req, disp, Xi)

        # if the node considered is a subscriber,
        # it sends his commodities
        if node == S[k]:
            Xi, req, disp, connect_to, neighborhood =
            s_connect(node, k, Ck, Lt, a, Cap, Tk, pnlty_node,
            req, disp, Xi)
```

```
        # if the node considered is a broker
        if node in B:
            # if it has request and no availability send
            # requests
            if req[k,node] == 1 and disp[k,node] == 0:
                Xi, req, disp, connect_to, neighborhood =
                b_connect(node, k, Ck, Lr, a, Cap, Tk,
                pnlty_node, req, disp, Xi)

            # if it has availability but no requests,
            # it pushes commodities
            if req[k,node] == 0 and disp[k,node] == 1:
                Xi, req, disp, connect_to, neighborhood =
                b_push(node, k, Ck, Lt, a, Cap, Tk,
                pnlty_node, req, disp, Xi)
```

Afterward, penalties are exchanged between nodes, and the nodes' memory that tracks the penalties for weight calculation is updated.

```
    pnlty_register = pnlty_exchange(pnlty_register,Lt)
```

In case the number of iterations of the outermost loop exceeds a certain threshold defined based on the instance, unsatisfied subscribers will signal the need for a reset, and a new search for solutions will be initiated.
The image below summarizes some of the main steps of the reasoning behind the algorithm.

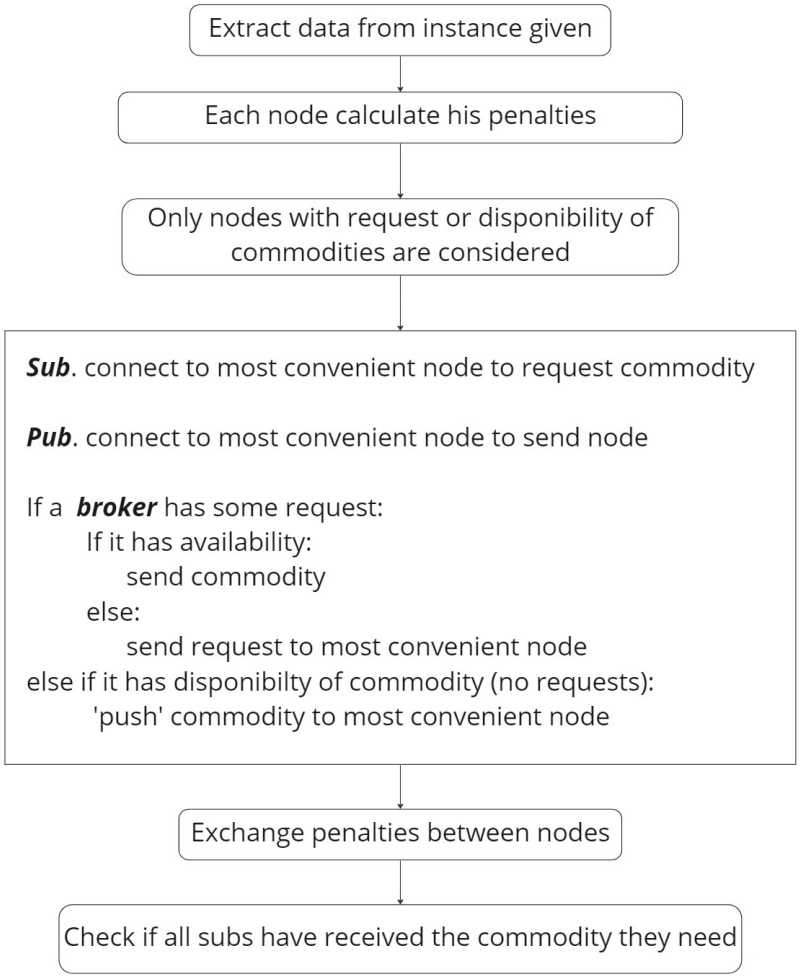

Figure 11: Description of Second Approach of the Distributed Lagrangian Heuristic Algorithm

3.8 Alternative Distributed Heuristic

This version of the distributed algorithm is based on the diffusion of information through a message exchange between nodes, rather than on Lagrangian penalties. The idea is to make information on the availability of commodities in the network, so that each node can select the best partner for data exchange.
Initially, nodes communicate their requests and availability to adjacent units, keeping track of the information in a local memory. Subscribers signal their requests to neighbors with less costly channels, which save the IDs of requesting subs in the "request" memory. Publishers instead signal their availability to nearby nodes with less expensive connections, which save the IDs of nodes with availability in the "availability" memory.

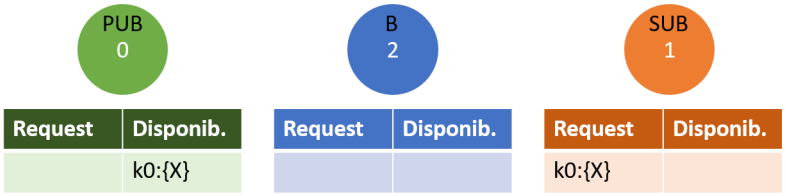

Figure 12: Example for Heuristic Distributed Algorithm - step 1

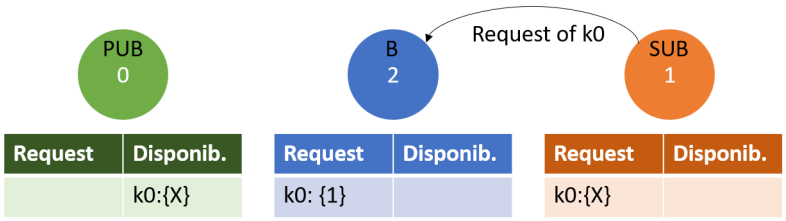

Figure 13: Example for Heuristic Distributed Algorithm - step 2

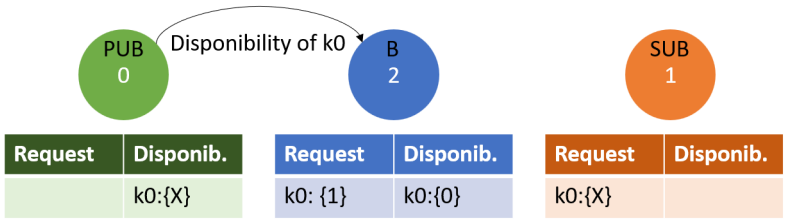

Figure 14: Example for Heuristic Distributed Algorithm - step 3

Subsequently, brokers follow a simple algorithm to satisfy requests. If they have availability of the requested commodities, they signal their availability to the ID saved in the "request" memory; otherwise, they request from more advantageous communicating nodes.

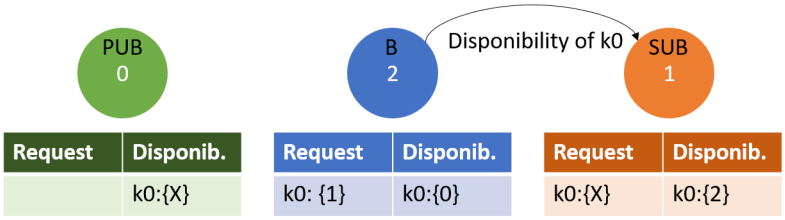

Figure 15: Example for Heuristic Distributed Algorithm - step 4

At this point, each node has a list of nodes with availability for the requested commodity. Subscribers connect to the first node in the "availability" list for that commodity. The node to which the subs connect will also connect to the first node whose ID appears in the "availability" list (under the relevant commodity). To avoid congestion, the node checks the bandwidth of the channel in which it intends to exchange data. If the channel is saturated, it moves on to the next node in the "availability" memory list.

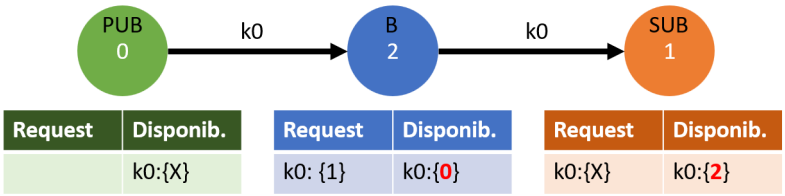

Figure 16: Example for Heuristic Distributed Algorithm - step 5

In summary, this distributed algorithm uses message exchange and information diffusion to select the best partners for data exchange, avoiding congestion and ensuring efficient management of commodities in the network.

The code implemented for managing requests for commodities by subscribers.

```
    for node in T:
    for k in K:
        if node in Tk[k]:
            info = scanNeighbors(node,k,N,K,Lr[node],
                    Ck[k][:,node])
            id_node = req_send(info,Cap,a[k],node)

            for i in id_node:
                status['request'][i][k].append(node)
```

The code implemented for managing the signaling of availability of commodities by publishers.

```
    k = 0
    for node in S:
        info = scanNeighbors(node,k,N,K,Lt[node],
                Ck[k][node,:])
        id_node = req_send(info,Cap,a[k],node)
        for i in id_node:
            status['disponibility'][i][k].append(node)
        k += 1
```

If a broker has a request for a commodity and has it in stock, they will signal its availability. However, if they don't have the commodity in stock, they will send a request to the most cost-effective node for fulfillment. This process is repeated in a loop until the subscribers receive the information on the availability of the commodity they require.

```
for node in B:
    for k in K:
        # if there are requests
        if len(status['request'][node][k]) >0:
            # but there is not disponibility
            if len(status['disponibility'][node][k]) == 0:
                # send request to most convenient node
                lr = Lr_register[k][node]
                info = scanNeighbors(node,k,N,K,lr,
                        Ck[k][:,node])
                if info == {}:
                    break
                else:
                    id_node = req_send(info,Cap,a[k],node)
                    for i in id_node:
                        if node not in
                        status['request'][i][k]
```

```
                    and i not in S:
                    status['request'][i][k].append(node)
                        # eliminate the already
                        # visited node
                        Lr_register[k][node].remove(i)

            # else if there is disponibility
            if len(status['disponibility'][node][k]) > 0:
                # the commodity is sent to node in request
                for r in status['request'][node][k]:
                    if node not in
                    status['disponibility'][r][k]:
                    status['disponibility'][r][k].append(node)
```

The 'scanNeighbors' function has been implemented to store information on neighboring nodes, including their node IDs and associated costs.

```
    def scanNeighbors(node,k,N,K,L,cost):
        info_neighbors = {}
        for neig in L:
            info = {neig: cost[neig]}
            info_neighbors.update(info)
        return info_neighbors
```

The 'req_send' function has been implemented to manage the selection of the most cost-effective node for sending 'request' or 'availability' messages. This function returns the ID of the node identified as the most convenient to communicate with, in terms of connection costs. The rationale behind this function is that if all connections have a positive weight, the function contacts the node with the lowest-weight connection. However, if there are negative weights, the function contacts all nodes with negative-weight connections, as this is advantageous.

```
    def req_send(info,Cap,a,node):
    filt_pesi = list(info.values())
    if all(x > 0 for x in filt_pesi):
        min_p = min(filt_pesi)
        send_req = [key for key, v in info.items()
                    if v == min_p]
    else:
        send_req = [key for key, v in info.items() if v <= 0]

    return send_req
```

After the preliminary phase in which nodes exchange information on requests and availability of commodities, a solution to the problem can be reconstructed using the 'disponibility' memory in the following way:

```
for node in N:
    for k in K:
        if node in Tk[k]:
            # select an ending point (the sub)
            endp = node
            disp = status['disponibility'][node][k]
            i = 0

            while True:
                if i < len(disp):
                   # select a starting point
                   # (the first id in 'disponibility')
                    startp = disp[i]

                    # if the capacity of the link allows it
                    # built the solution
                    if a[k] + O[startp,endp] <=
                            Cap[startp,endp]:
                        Xi[k][startp,endp] = 1
                        X[k][startp,endp] += 1
                        O[startp,endp] = a[k]

                    # else, select the next id in
                    # 'disponibility'
                    else:
                        i +=1
                else:
                    print('ERROR')
                    break
```

Within the loop started with the above code, another loop is executed to complete the solutions using the information available to the brokers. An arc's start point and end point are defined based on the IDs in the 'disponibility' memory, thus retracing the entire path of the commodity back to the source publisher.

```
# for brokers
while True:
    # if the start point is the pub. break
    if startp == S[k]:
        break
    # else retrace the disponibility memory of each broker
    else:
        endp = startp
        disp = status['disponibility'][endp][k]
        i = 0
        while True:
            if i < len(disp):
```

```
                startp = disp[i]
                if Xi[k][startp,endp]==1:
                    X[k][startp,endp] += 1
                    break
                elif a[k] + O[startp,endp] <=
                          Cap[startp,endp]:
                    Xi[k][startp,endp] = 1
                    X[k][startp,endp] = 1
                    O[startp,endp] = a[k]
                else:
                    i +=1
            else:
                print('ERROR')
                break
```

3.9 Auxiliary Functions

Function to graph the network

The function 'myGraph' creates a graphical representation of the nodes in a graph. It takes as input several parameters, such as K (the set of commodities), N (the set of nodes), S (the set of source nodes), B (the set of broker nodes), coordinates (the coordinates of each node), and Xi (the decision variables). The nodes are assigned colors based on whether they belong to the set of source nodes, broker nodes, or destination nodes. The function generates a series of unique colors for the arcs and assigns each commodity k a different color. The resulting graph is displayed using the matplotlib library, with a legend to identify the different commodities.

```
def myGraph2(K, N, S, B, T, coordinates, Xi):
    G = nx.DiGraph()

    # colors
    node_colors = []
    for i in N:
        if i in B:
            if any(Xi[k][i,j]==1 for k in K for j in N) or
            any(Xi[k][j,i]==1 for k in K for j in N):
                clr = "yellow"
            else:
                clr = "white"
        if i in S:
            clr = "green"
        elif i in T:
```

```
            clr = "red"
        node_colors.append(clr)

        # coordinate
        cord = [coordinates[0,i],coordinates[1,i]]
        G.add_node(i, pos= cord , node_color = clr)

    # unique colors for edges
    edge_colors = sns.color_palette("husl", len(K))
    edge_color_map = {k: edge_colors[idx] for idx, k
                      in enumerate(K)}

    for k in K:
        for i in N:
            for j in N:
                if Xi[k][i,j] == 1:
                    G.add_edge(i, j, color=edge_color_map[k])


    nx.draw_networkx_nodes(G, pos=nx.get_node_attributes(G,
    'pos'), node_color= node_colors)
    nx.draw_networkx_labels(G, pos=nx.get_node_attributes(G,
    'pos'))
    nx.draw_networkx_edges(G, pos=nx.get_node_attributes(G,
    'pos'), edgelist = G.edges, edge_color=[e[2]['color']
    for e in G.edges(data=True)],
    connectionstyle="arc3,rad=0.1")


    # Legend
    handles = [Patch(color= edge_color_map[k],
    label=f'k = {k}') for k in K]
    plt.legend(handles=handles, loc='lower left',
    bbox_to_anchor=(0.95, 0.76))

    plt.draw()
```

Function to calculate the cost of the solution with Formulation 1

This function takes as input the sets of commodities, nodes, links, and the costs of links for each commodity. It also takes the decision variables X that indicate whether a link is used or not for each commodity. The function then calculates the total cost of the solution using Formulation 1. It does this by iterating through all commodities, nodes, and their neighbors, checking whether each link is used, and multiplying the cost of that link by the decision variable for that commodity and link. The total cost is returned

as the output of the function.

```
    def cost(K,N,Lr,Ck,X):
        Zv1 = 0
        for k in K:
            for i in N:
                for j in Lr[i]:
                    Zv1 += Ck[k][j,i]*X[k][j,i]
        return Zv1
```

Function to calculate the cost of the solution with Formulation 2

This function takes as input the sets of commodities, arcs, and the costs of arcs for each commodity. It also takes the decision variables Xi that indicate whether an arc is used or not for each commodity. The function then calculates the total cost of the solution using Formulation 2. It does this by iterating through all commodities and their paths (represented as arcs), checking whether each arc is used, and multiplying the cost of that arc by the decision variable for that commodity and arc. The total cost is returned as the output of the function.

```
    def cost2(K,A,Ck,Xi):
        Zv2 = 0
        for k in K:
            for endp in A:
                Zv2 += Ck[k][endp[0],endp[1]]*
                       Xi[k][endp[0],endp[1]]
        return Zv2
```

4 Results Analysis

In this chapter, we delve into the practical application of the algorithms discussed in the previous chapter. The primary objective is to assess the effectiveness of these algorithms under different simulation scenarios. The algorithms were specifically designed to facilitate message exchange between nodes within a publish-subscriber model, where central nodes act as brokers that route information.

To comprehensively evaluate the performance of these algorithms, we utilized eight different case studies. Each case study was carefully crafted to include problems of different sizes, including varying numbers of nodes, edges, capacity, and the cost of node utilization. Through simulating these instances, we were able to gather valuable data for the evaluation and modification of the algorithms.

The primary goal of conducting these simulations was to identify the strengths and weaknesses of each algorithm under different circumstances. By analyzing the results obtained, we were able to identify the specific scenarios in which each algorithm performs better or worse. This chapter presents the details of the simulations and performance analyses of the algorithms used. We also provide a critical evaluation of the results obtained from each simulation, discussing the implications of these results and providing suggestions for future algorithm improvements.

4.1 Simulation Scenarios

In the subsequent section, we describe the different simulation scenarios considered in detail. Each scenario was carefully crafted to provide a comprehensive evaluation of the algorithms under different circumstances. These scenarios were designed to vary in size and complexity, allowing us to obtain a comprehensive understanding of the performance of the algorithms under different conditions.

Tiny case

The instance named "*Tiny*" represents the smallest dimension solution among the available alternatives. It involves the use of six nodes, of which two act as publishers and one as a subscriber, each requiring both commodities in question. The configuration of this instance is clearly represented in Figure 17, where nodes corresponding to brokers, publishers, and the subscriber are distinguished by the colors blue, green, and red, respectively. Additionally, the edges present between nodes are highlighted in gray.

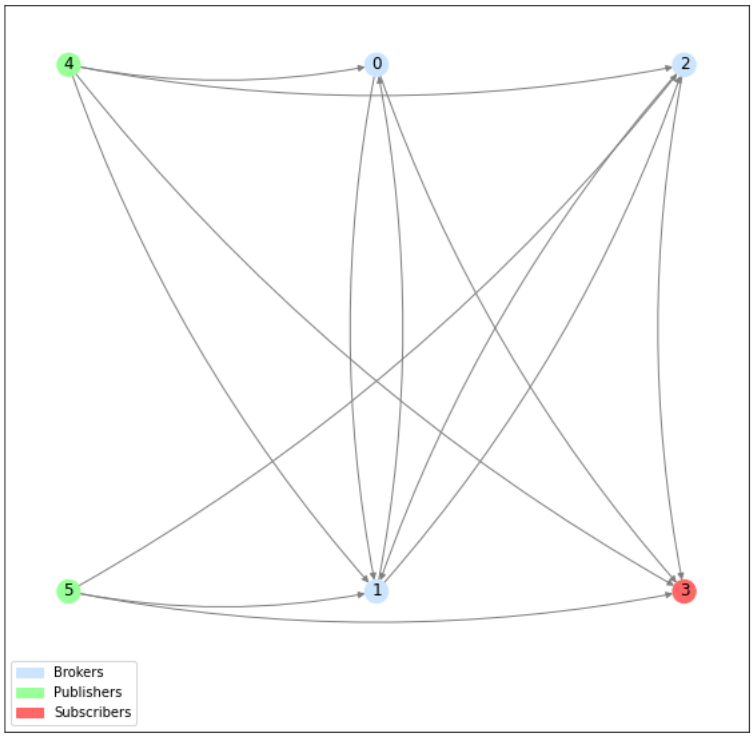

Figure 17: Tiny Instance

Case2

This instance has 2 subscribers, 3 publishers, and 6 brokers, for a total of 11 nodes. Figure 18 shows the topology of this scenario. Subscriber '6' requests commodities 0 and 1 generated by publishers '8' and '9', while subscriber '7' requests commodities 0 and 2 generated by publishers '8' and '10'. The subcriber 6 requests the commodities: [0, 1]. The subscriber 7 requests the commodities: [0, 2].

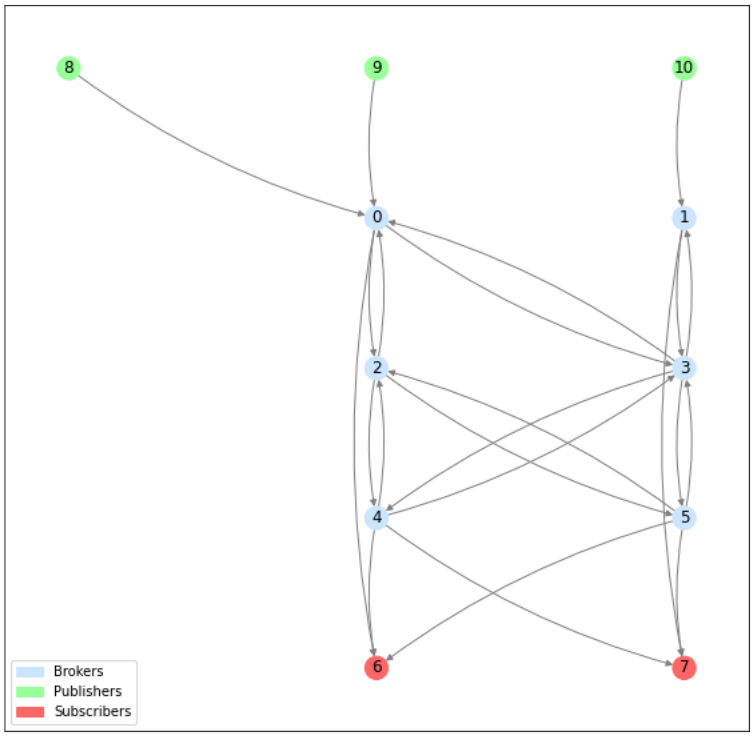

Figure 18: case2 Instance

Case3

This instance has 20 nodes, including 6 publishers, 3 subscribers, and 11 brokers. There are 57 links that allow communication between the nodes in this problem. Figure 19 shows the topology of this scenario. The subscriber 11 requests the commodities: [0, 2, 5]. The subscriber 12 requests the commodities: [1, 2, 3]. The subscriber 13 requests the commodities: [3, 4, 5].

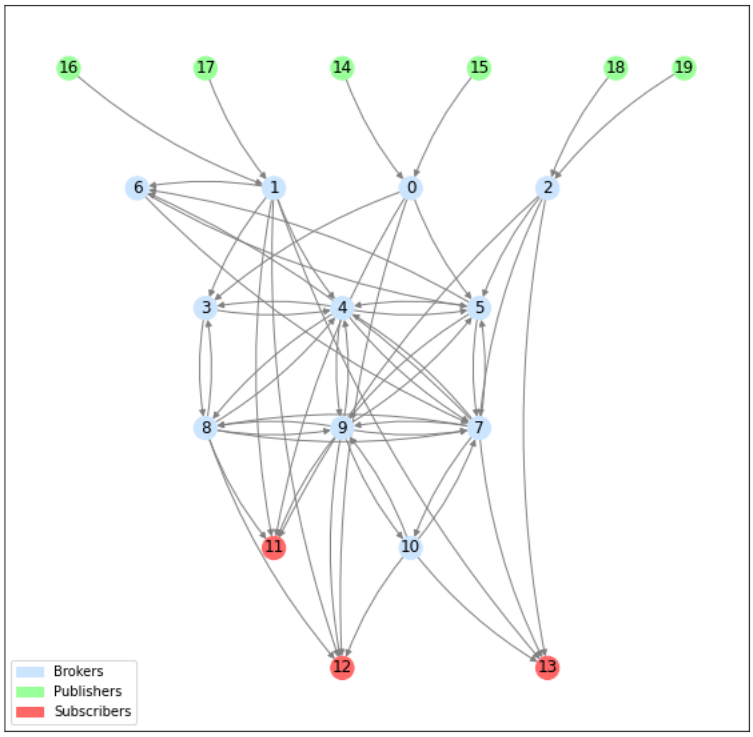

Figure 19: case3 Instance

Case4

This instance has 29 nodes, including 6 publishers, 3 subscribers, and 20 brokers. There are 69 links that allow communication between the nodes in this problem. Figure 20 shows the topology of this scenario. The subscriber 26 requests the commodities: [0, 2]. The subscriber 27 requests the commodities: [1, 4]. The subscriber 28 requests the commodities: [3, 5].

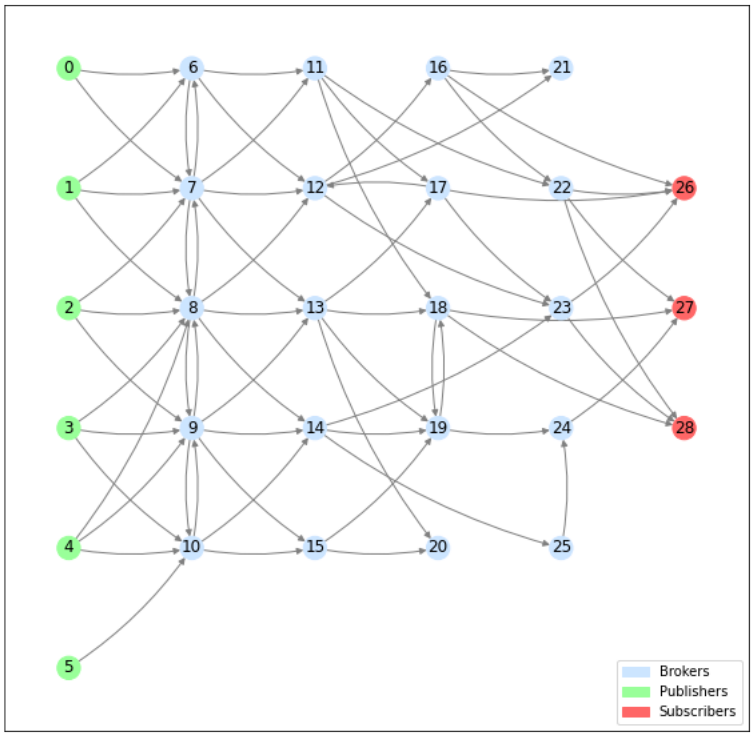

Figure 20: case4 Instance

Case5

This instance has 14 nodes, including 3 publishers and 3 subscribers. There are 21 links that allow communication between the nodes in this problem. Figure 21 shows the topology of this scenario. This instance was created to recreate a more realistic situation by generating random positions of the nodes. The subscriber 2 requests the commodities: [1]. The subscriber 3 requests the commodities: [0, 2]. The subscriber 11 requests the commodities: [1, 2].

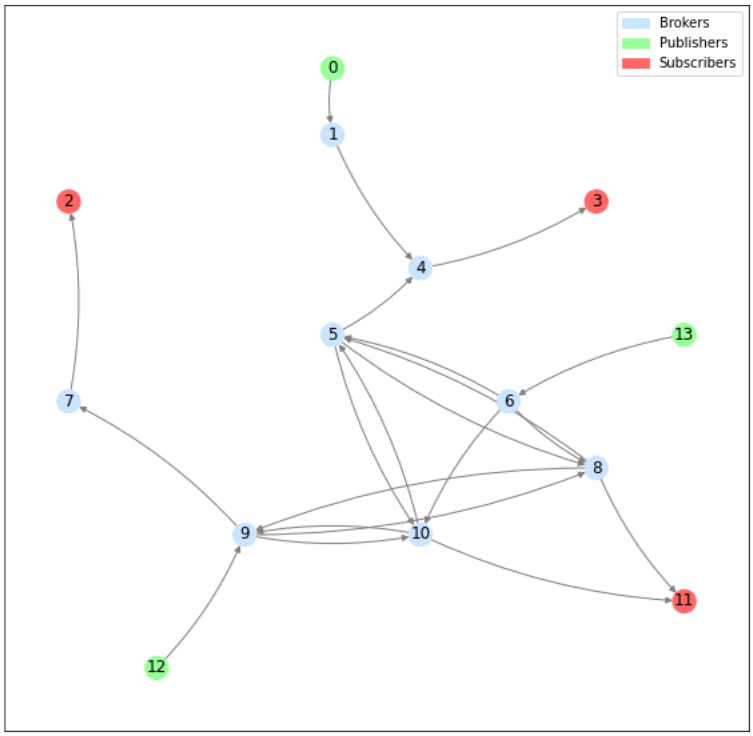

Figure 21: case5 Instance

Case6

This instance has 17 nodes and 30 links (Figure 22), aiming to represent a realistic situation similar to Case 5, but with more nodes. The subscriber 0 requests the commodities: [1, 2]. The subscriber 1 requests the commodities: [0, 2, 3]. The subscriber 14 requests the commodities: [0, 3].

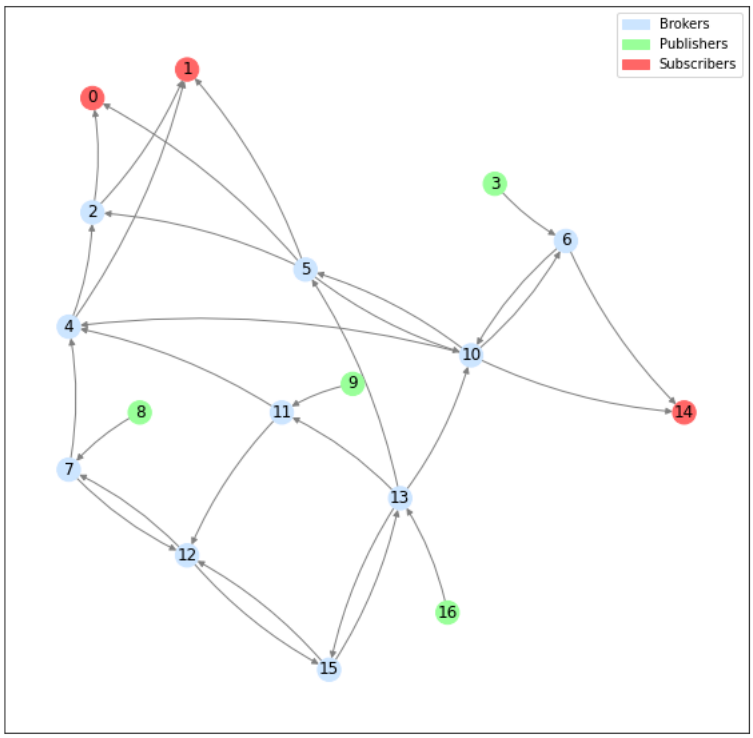

Figure 22: case6 Instance

Case7

This instance involves a high number of nodes and links, specifically 76 nodes and 126 links (Figure 23). The subscriber 72 requests the commodities: [2, 5]. The subscriber 73 requests the commodities: [0]. The subscriber 74 requests the commodities: [1, 4]. The subscriber 75 requests the commodities: [3].

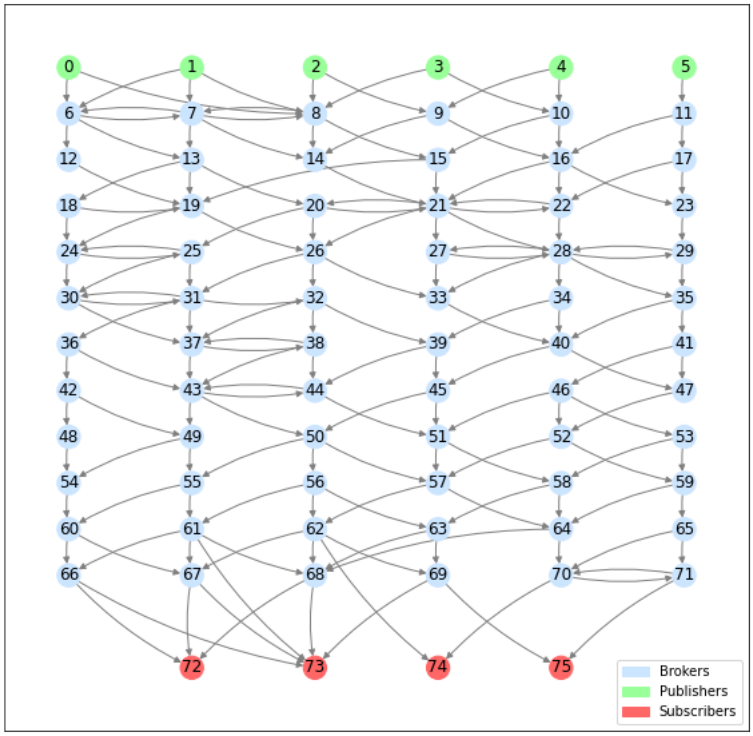

Figure 23: case7 Instance

Case8

This instance is a reproduction of the 'case7' instance that has been modified by updating the weights and capacities of the connections, in order to confer greater complexity to the problem resolution.

4.2 Simulations with Formulation 1

In this section, we present the results obtained with Formulation 1 described in Section 2.2. Formulation 1 was implemented in Python using the solver CPLEX, as described in Section 3.1. The main objective of Formulation 1 is to find the minimum cost of the problem, which is the sum of the products of the cost of each link by the variable x_{ij}^k, indicating the number of sub-networks served through the arc (i, j) (equation 53). This formulation assumes a centralized approach where the solver has knowledge of all information about the nodes, links, and commodities. The results of the problem solved with Formulation 1 are presented together with the values obtained from the continuous relaxation. Table 6 shows the results obtained for each instance of the problem, along with the objective function and corresponding continuous relaxation values.

It is important to note that the objective function values found using Formulation 1 correspond or closely approximate the values of continuous relaxation, providing a clear indication of the optimality of the solutions found. This demonstrates the effectiveness of Formulation 1 and the accuracy of its implementation in Python using CPLEX.

$$\textbf{minimize} \sum_{k \in K} \sum_{i \in N} \sum_{j \in \Gamma_i^- 1} c_{ji} x_{ji}^k \tag{53}$$

The results obtained are shown in the Table 6:

Instance	**Integer Solution**	**Continous Relaxation**
Tiny	16	16
case2	88	88
case3	198	196
case4	114	112.33
case5	38	38
case6	120	120
case7	345	345
case8	365	353

Table 6: Results Obtained with Formulation 1 with integrality constraints, and Continous Relaxation

Integer Optimal Solution with Formulation 1

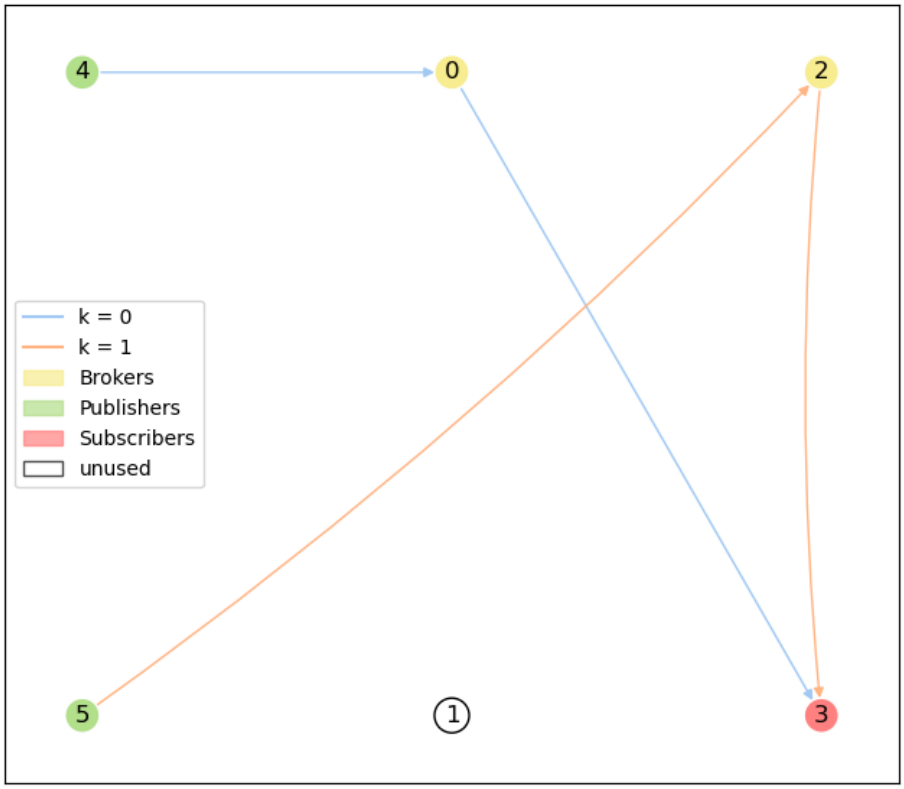

Figure 24: case Tiny solved with Formulation 1 using CPLEX

Case2 Instance Integer Optimal Solution with Formulation 1

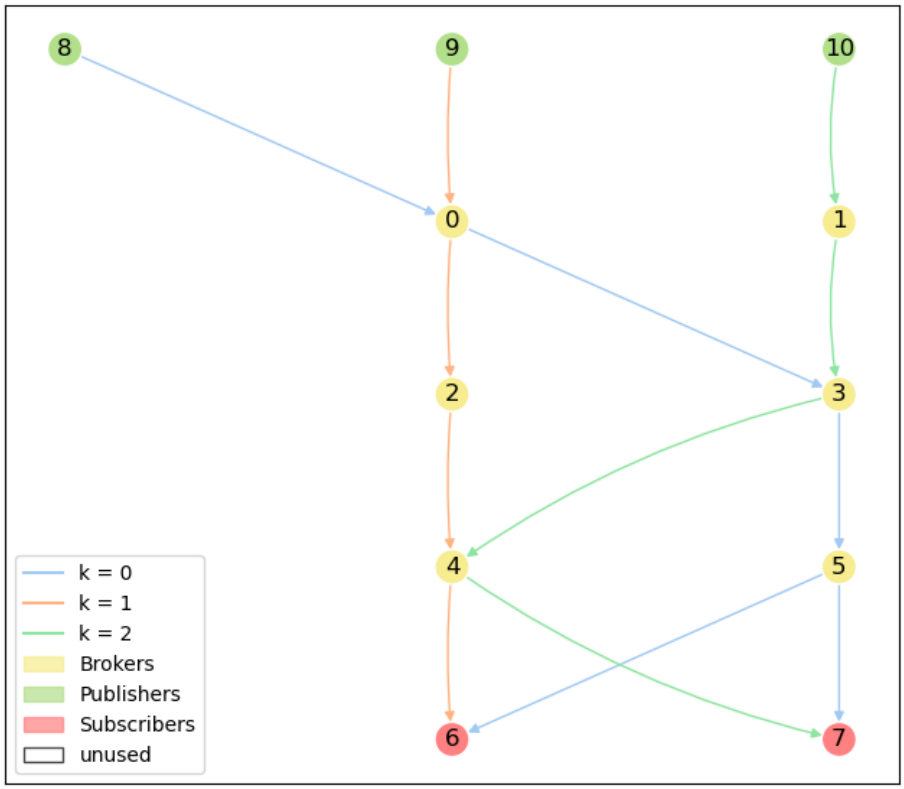

Figure 25: case2 solved with Formulation 1 using CPLEX

Case3 Instance Integer Optimal Solution with Formulation 1

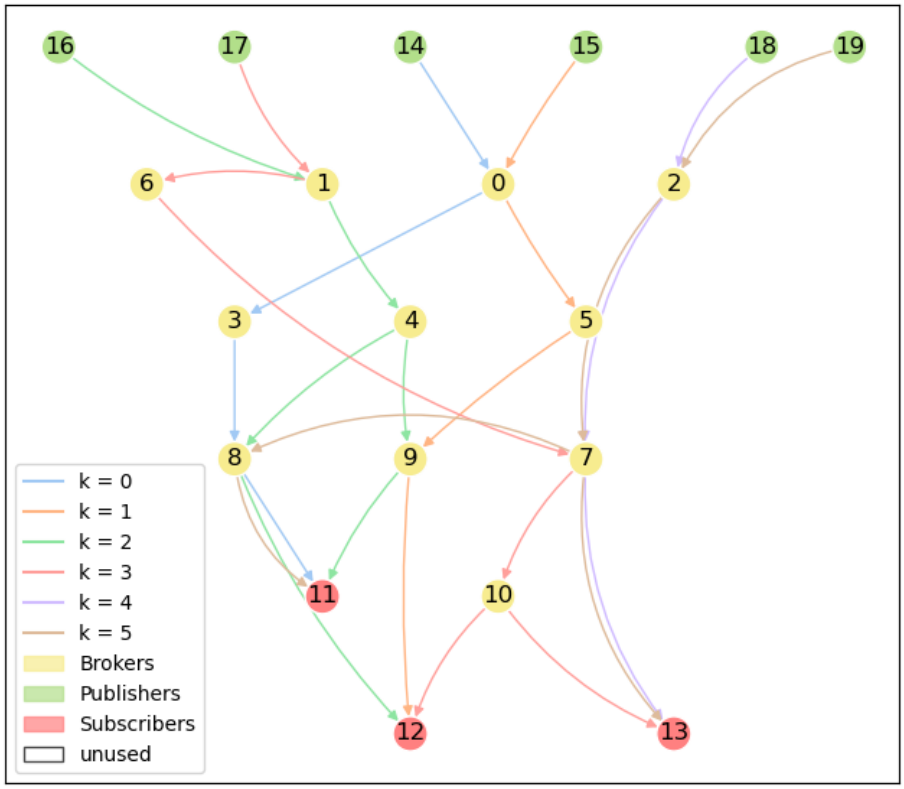

Figure 26: case3 solved with Formulation 1 using CPLEX

Case4 Instance Integer Optimal Solution with Formulation 1

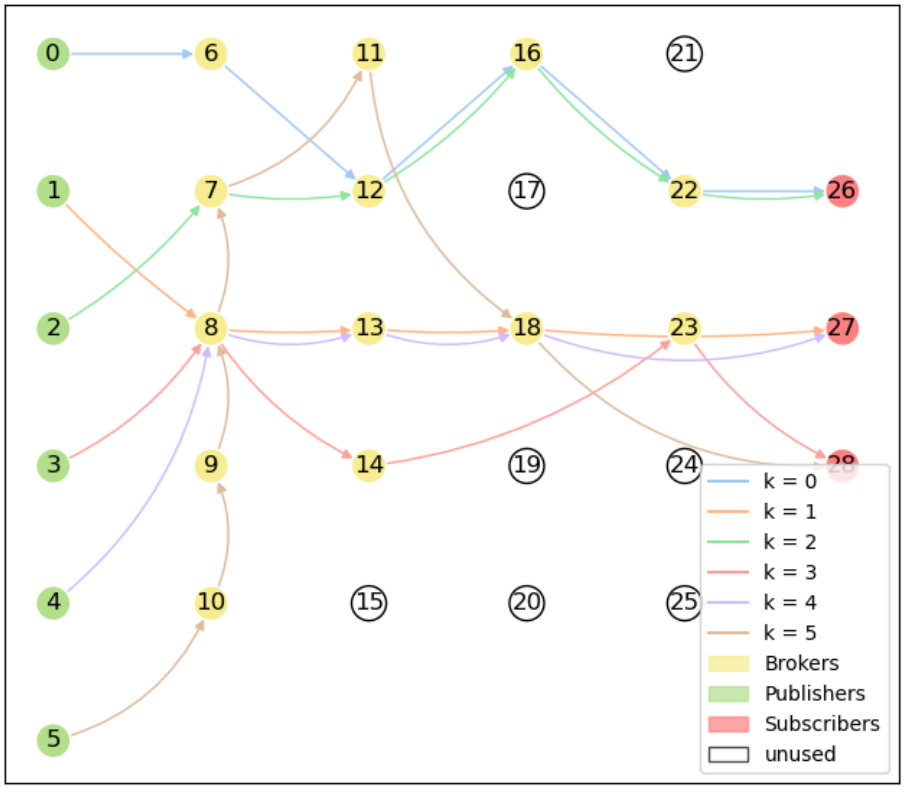

Figure 27: case4 solved with Formulation 1 using CPLEX

Case5 Instance Integer Optimal Solution with Formulation 1

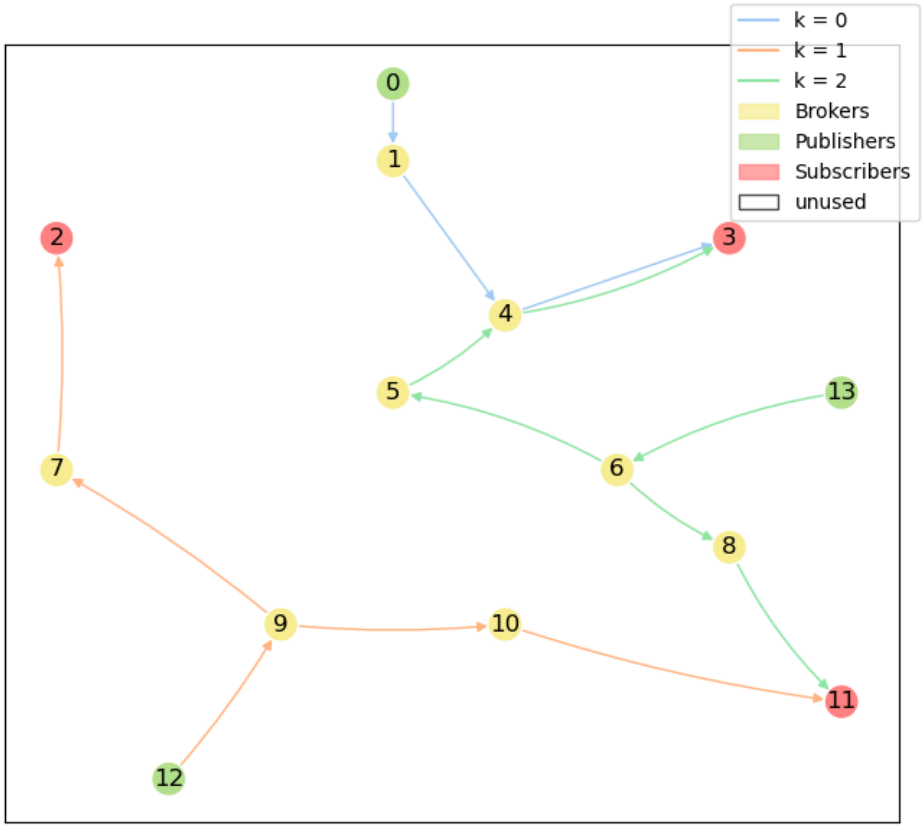

Figure 28: case5 solved with Formulation 1 using CPLEX

Case6 Instance Integer Optimal Solution with Formulation 1

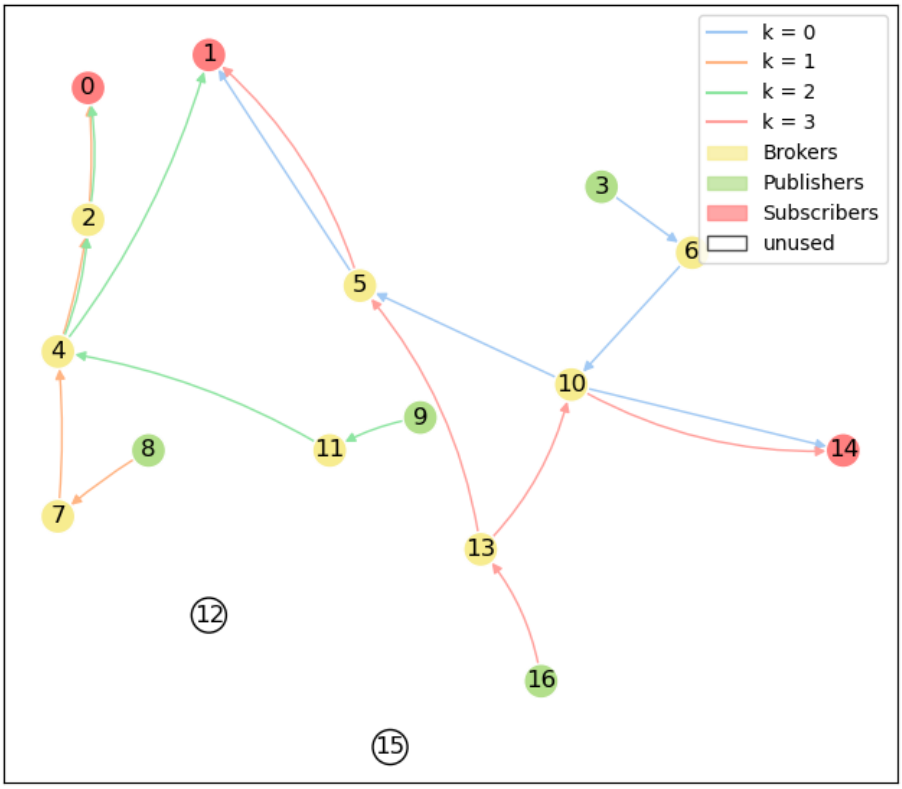

Figure 29: case6 solved with Formulation 1 using CPLEX

Case7 Instance Integer Optimal Solution with Formulation 1

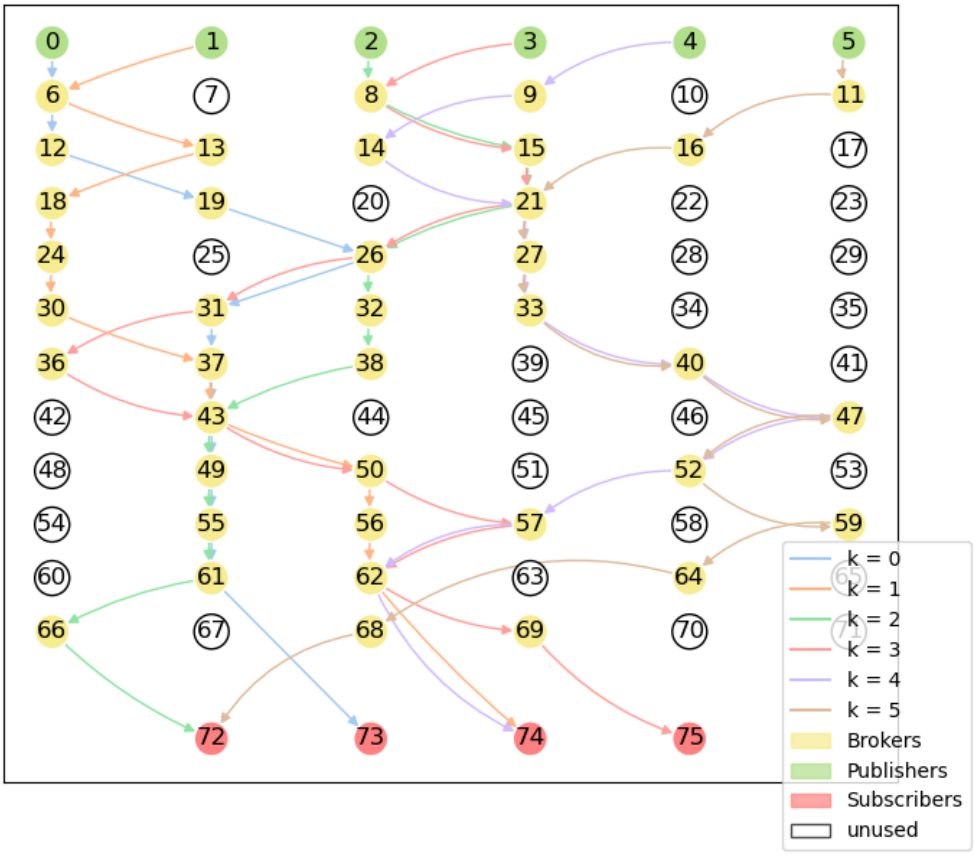

Figure 30: case7 solved with Formulation 1 using CPLEX

Case8 Instance Integer Optimal Solution with Formulation 1

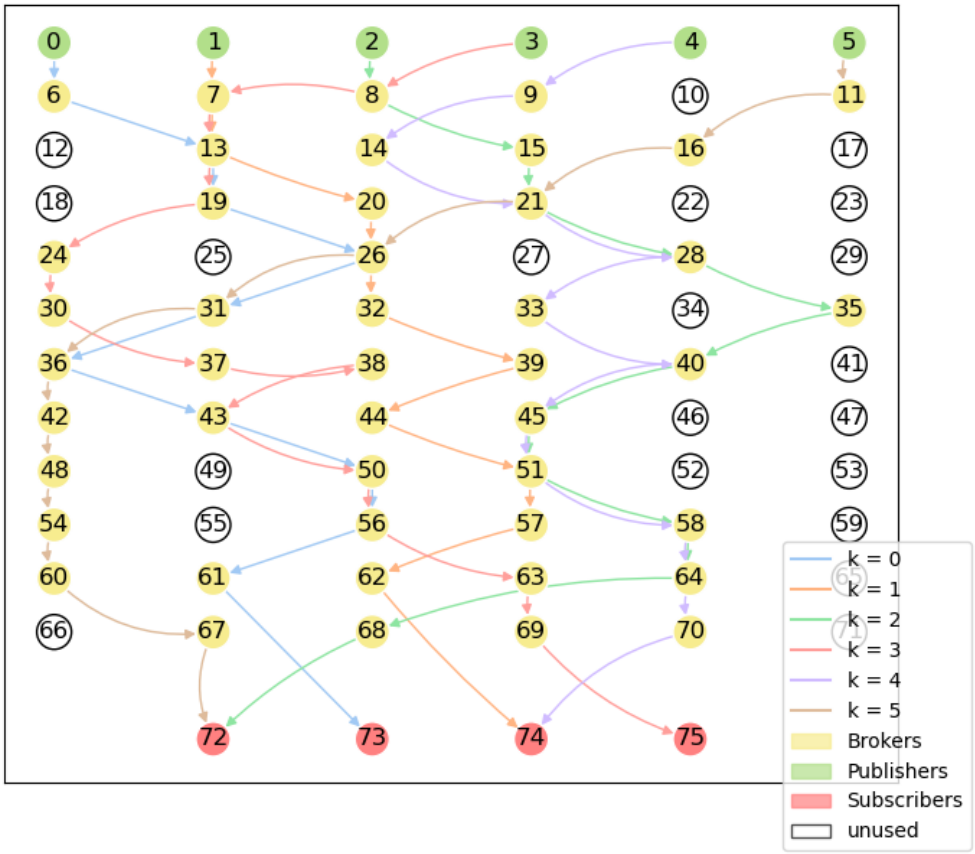

Figure 31: case8 solved with Formulation 1 using CPLEX

4.3 Simulations with Formulation 2

This section describes the results obtained from the application of formulation 2, which was previously described in Section 2.3, implemented using the Python programming language with CPLEX as the solver, as described in Section 3.3. It is important to note that, like Formulation 1, this formulation adopts a centralized approach, assuming that the solver has access to all information about the network, including nodes, edges, and commodities. The optimization model developed in this formulation aims to minimize the sum of the costs of each link for the variable ξ_{ij}^k, which is a binary variable indicating whether information exchange occurs along arc (ij) or not. Unlike Formulation 1, this model does not consider the subsequent branching that may occur to serve different subs.

The objective function of this model is expressed as follows:

$$\textbf{minimize} \sum_{k \in K} \sum_{i,j \in A} c_{ji} \xi_{ji}^k \tag{54}$$

In the following pages, the results obtained from the heuristic algorithm developed to find an upper bound will be analyzed and compared, and analyses of the formulation with Lagrangian relaxation will be carried out. Finally, the graphical results obtained from the simulation of the model for the various defined instances will be reported at the end of Section 4.3.

4.3.1 Analysis of Heuristic Algorithm Results

The centralized heuristic algorithm has been designed to provide upper bound values for the original problem by generating a set of feasible solutions and selecting the best one. As evidenced by the analysis of the results reported in Table 7, the number of algorithm iterations is a crucial factor that affects the algorithm's ability to find the optimal solution. In particular, increasing the size of the proposed solution set allows for exploration of a greater number of solutions, thereby increasing the probability of finding a better solution. In fact, in the search for the best solutions, the algorithm is executed five times for each set of solutions (1, 2, 5, and 10), ultimately selecting the best solution found. The results clearly indicate that the probability of finding optimal solutions increases with the number of solutions examined. However, it is important to note that the use of heuristic algorithms for optimization problems always entails some degree of uncertainty in the solution. Therefore, it is necessary to carefully evaluate the results obtained from the centralized heuristic algorithm, even though the produced upper bounds may provide an approximate estimate of the optimal solution.

Instance	1 solution	2 solutions	5 solutions	10 solutions
Tiny	16	16	16	16
case2	68.2	67.4	67	67
case3	174	140.2	137	135.2
case4	112.8	112.4	112	112
case5	34	34	34	34
case6	95	95	95	95
case7	345	345	345	345
case8	374.2	373.4	371.8	368.6

Table 7: Analysis of centralized heuristic algorithm

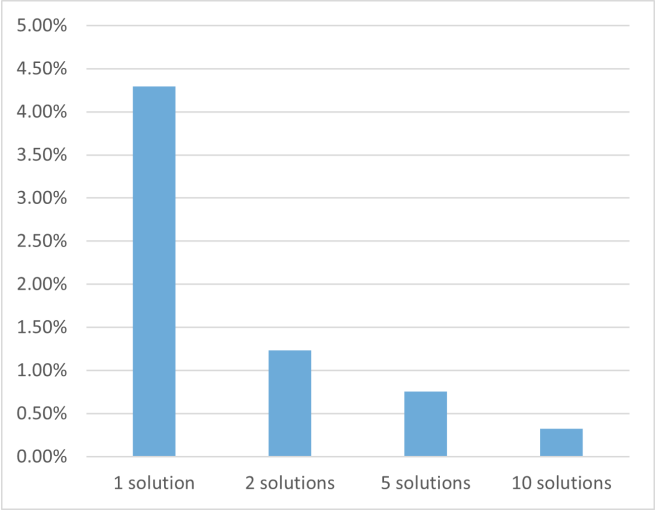

Figure 32: Percentage deviation from the optimal value of heuristic algorithm

From the bar chart shown in Figure 32, it is possible to observe the percentage deviation of the solutions produced by the heuristic algorithm compared to the optimal solutions provided by the optimizer. This deviation depends on the number of solutions considered for selecting the best solution.

4.3.2 Analysis of the Lagrangian Relaxation for Formulation 2

Below are presented the results of the Lagrangian relaxation proposed in Section 3.6, where the flow constraints of commodities are relaxed. The algorithm uses the subgradient method to update the Lagrangian penalties and return the best lower bound obtained in all iterations. As can be seen from the trend in Figure 33, the maximum value of the lower bound tends to increase, albeit with some oscillations, until it reaches values close to the

optimal solution of the problem.

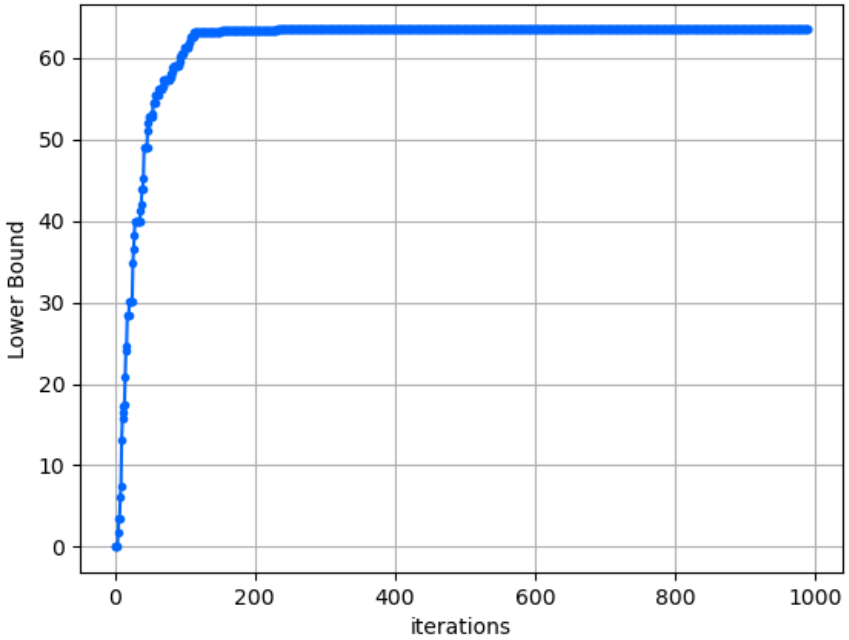

Figure 33: Lower Bound trend for instance 'case2' with α=0.1 on 1000 iteratinos

For analysis purposes, simulations were carried out to evaluate the effect of parameters such as the value of the upper bound and the alpha parameter on the value of the Lower Bound obtained by the algorithm.

Analysis of the Influence of Upper Bound

To evaluate the effect of the upper bound value on the results of the algorithm with Lagrangian relaxation and, therefore, on the achieved lower bound, an analysis was conducted. Various upper bound values were used, starting from the optimal solution obtained with the heuristic algorithm (UB) and deviating by 7%, 15%, 22%, 30%, 37%, 60%. The obtained values were then extracted and graphically represented as shown in the example in Figure 34.

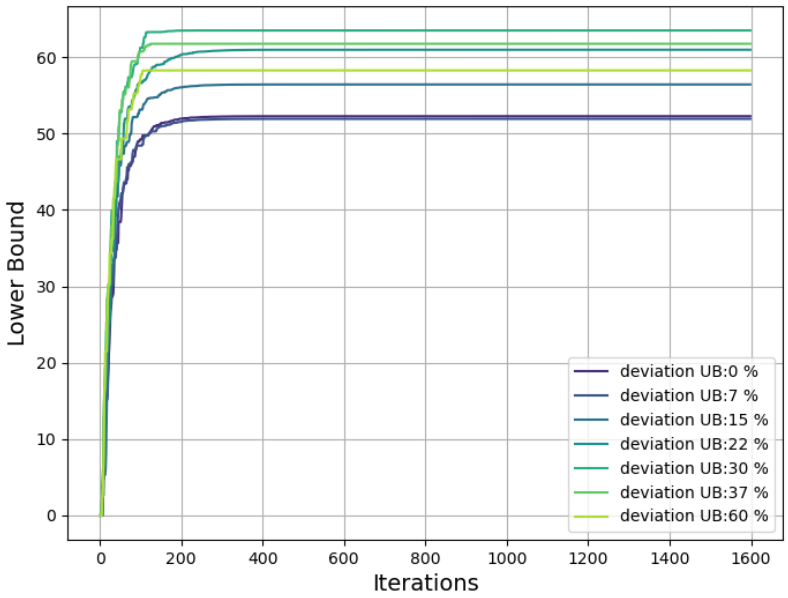

Figure 34: Upper Bound influnece on Lower Bound values, simulated with instance 'case2', α=0.1 for 2000 iterations

The results obtained show how the value of the upper bound affects the trend of the obtained lower bound. In particular, it can be observed that for moderate increments of the upper bound value, the lower bound increases, while for higher increments, the lower bound tends to decrease. The reason for this can be traced back to the fact that the upper bound is a fundamental parameter for updating the Lagrangian penalties in the algorithm. The update of the penalties is explicitly shown in formulas 55 and 56 for clarity.

$$step = \alpha \frac{UB - Z_{LR}}{||s||^2} \tag{55}$$

$$\lambda_k^i = \lambda_k^i + step \cdot s_k^i \tag{56}$$

Analysis of the Influence of parameter alpha

The choice of the value of the parameter alpha can have a strong impact on the results obtained from the Lagrangian relaxation. To explore this effect, a set of alpha values were selected, including 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, and 0.5. For each alpha value, the algorithm was executed, and the results were collected for further analysis. As shown in Figure 35, the obtained results vary considerably depending on the selected alpha value.

It should be noted that the alpha parameter is dynamically updated within the algorithm and is halved if no better lower bound is found after a certain number of iterations. The selected set of alpha values only defines the initial values of the alpha parameter used during the algorithm's execution.

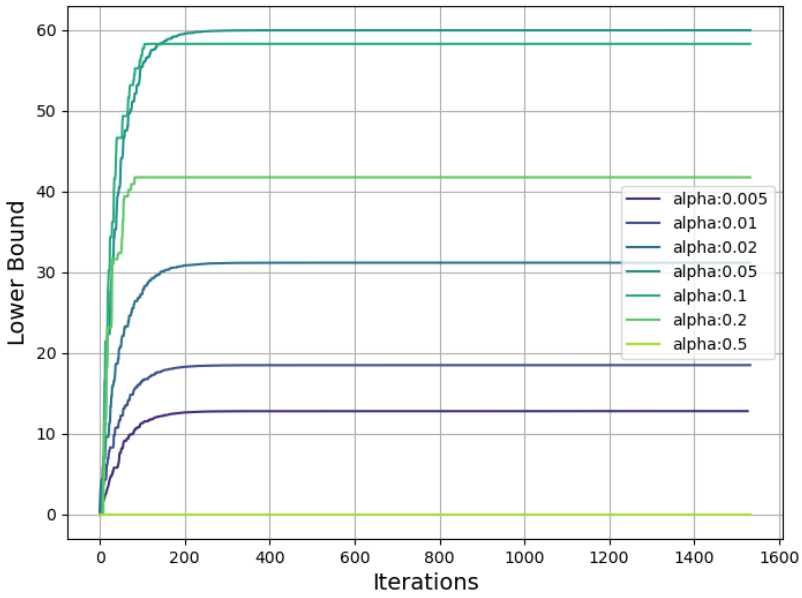

Figure 35: Parameter α influnece on Lower Bound values, simulated with instance 'case2', for 2000 iterations

From the results obtained, it can be observed that the optimal value for the parameter α in this case is between 0.05 and 0.1. It is important to note that this parameter affects the step size of the Lagrangian penalty update

(as indicated in equation 55). If the value of α is too small, the number of iterations required to reach the desired lower bound values becomes excessive. Conversely, if the value of α is too high, the search for the maximum in the Lagrangian function becomes less precise, as the step size becomes too large. In practice, this will result in overshooting the maximum and compromising the solution found.

Comparison of Results

The results obtained using the model implemented with Formulation 2 and the aid of CPLEX, together with the results obtained using the continuous relaxation and the Lagrangian relaxation formulations, are reported in Table 8. The relaxation was solved with a parameter α equal to 0.1 for a total of 2000 iterations. It is noteworthy that the results presented in the table represent a comparison between different methodologies employed to solve the problem under consideration. In particular, the continuous relaxation and the Lagrangian relaxation are techniques used to simplify the problem resolution process. It is emphasized that careful consideration of the results obtained is important, since the use of different resolution techniques can significantly affect the model performance and the final results.

Instance	**CPLEX**	**Continous Rel.**	**Lagrangian Rel.**
Tiny	16	16	16
case2	67	66	64.49
case3	135	130.5	125.52
case4	112	112	98.87
case5	34	22	18.84
case6	95	70	65.79
case7	345	345	236.68
case8	365	353	262.62

Table 8: Results obtained from different algorithms

The results obtained from the Lagrangian relaxation provide a lower bound for the original problem. However, they could be improved as they could theoretically dominate continuous relaxation. Future developments could include the implementation of the "quasi-constant" step rule mentioned in Section 4.3.2 for the penalties update.

Results obtained from Formulation 2 implemented with CPLEX
Tiny Instance Optimal Solution with Formulation 2

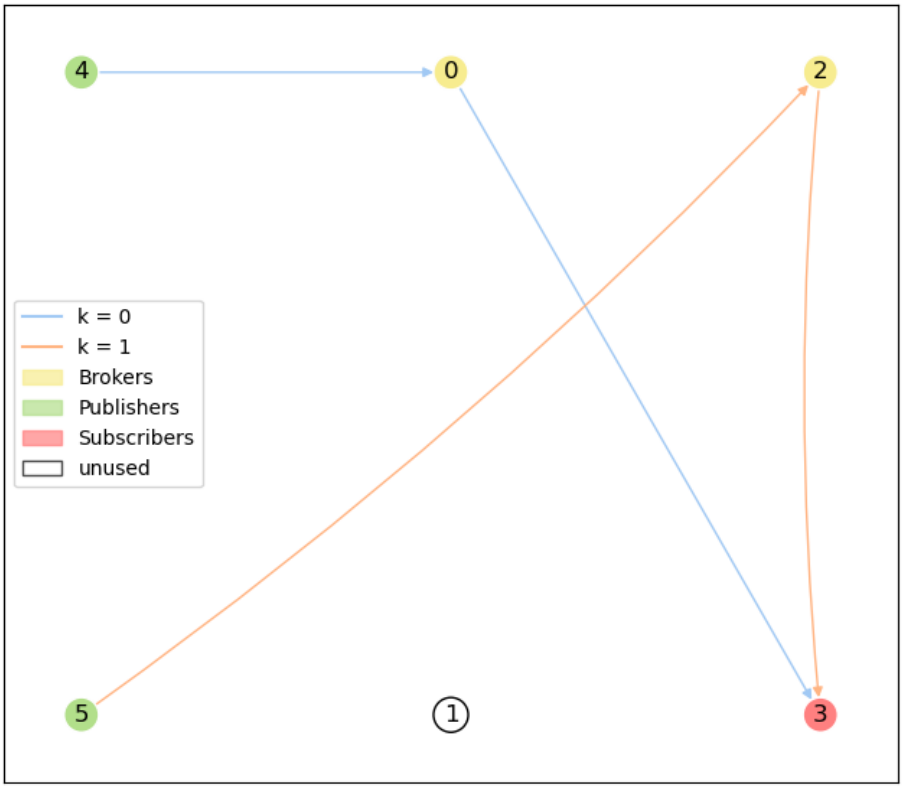

Figure 36: case Timy solved with Formulation 2 using CPLEX

Case2 Instance Optimal Solution with Formulation 2

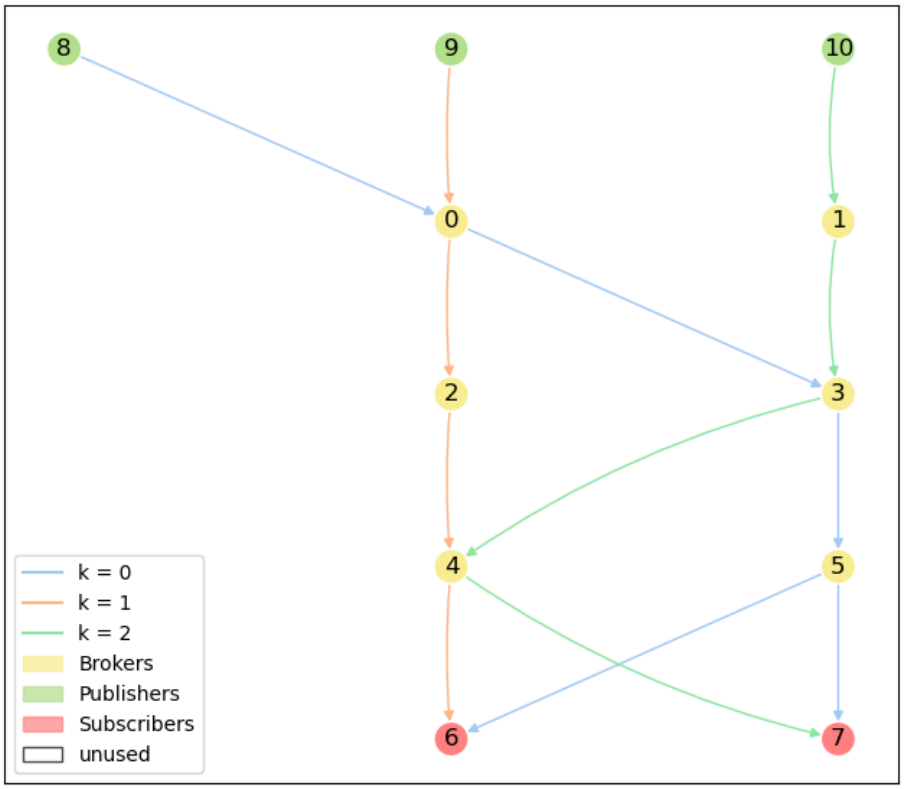

Figure 37: case2 solved with Formulation 2 using CPLEX

Case3 Instance Optimal Solution with Formulation 2

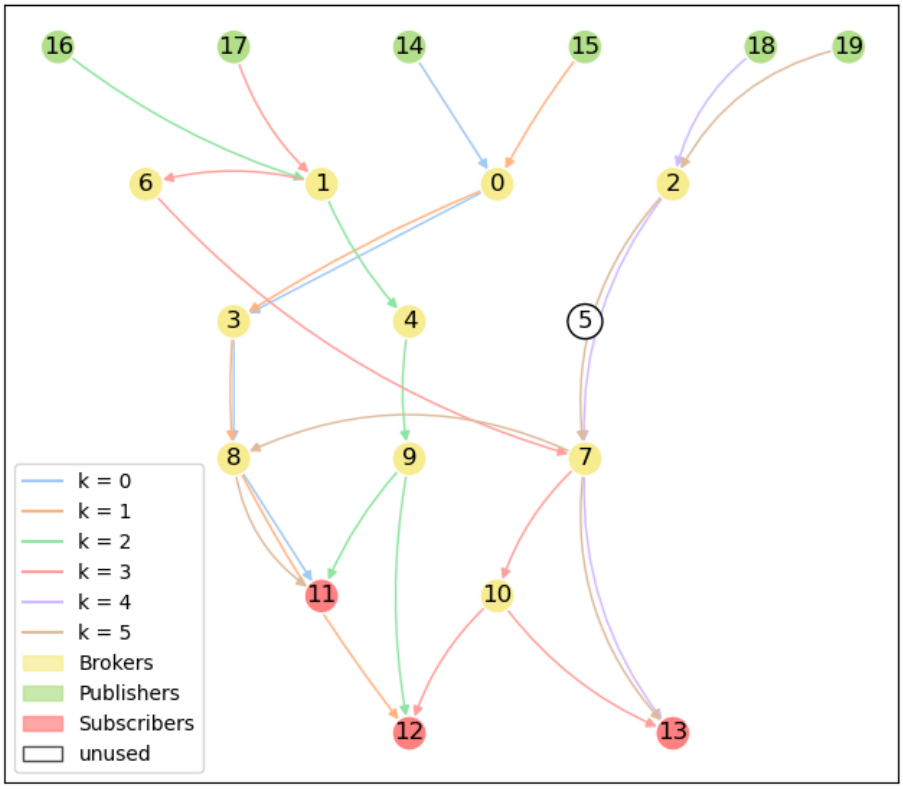

Figure 38: case3 solved with Formulation 2 using CPLEX

Case4 Instance Optimal Solution with Formulation 2

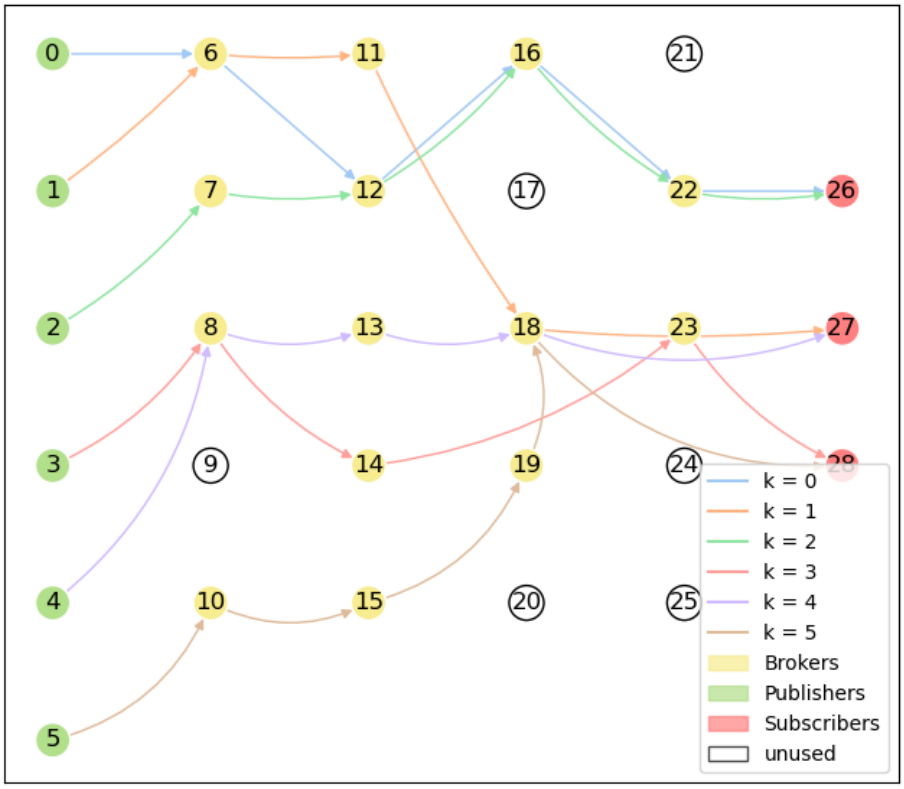

Figure 39: case4 solved with Formulation 2 using CPLEX

Case5 Instance Optimal Solution with Formulation 2

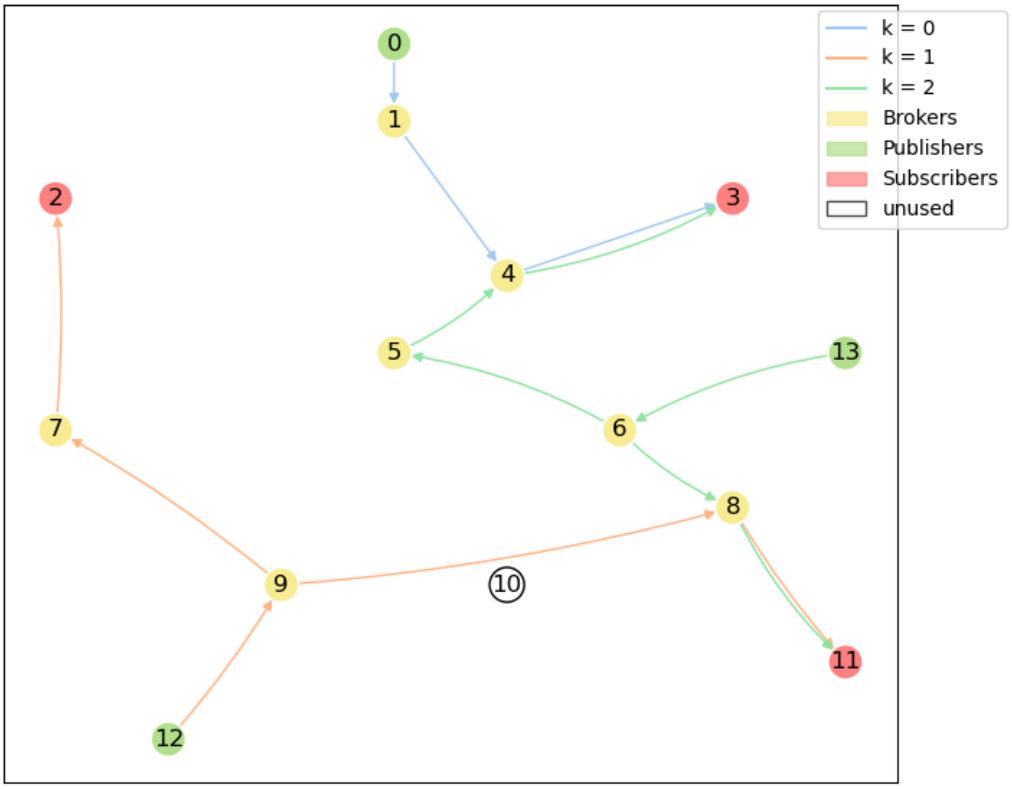

Figure 40: case5 solved with Formulation 2 using CPLEX

Case6 Optimal Solution with Formulation 2

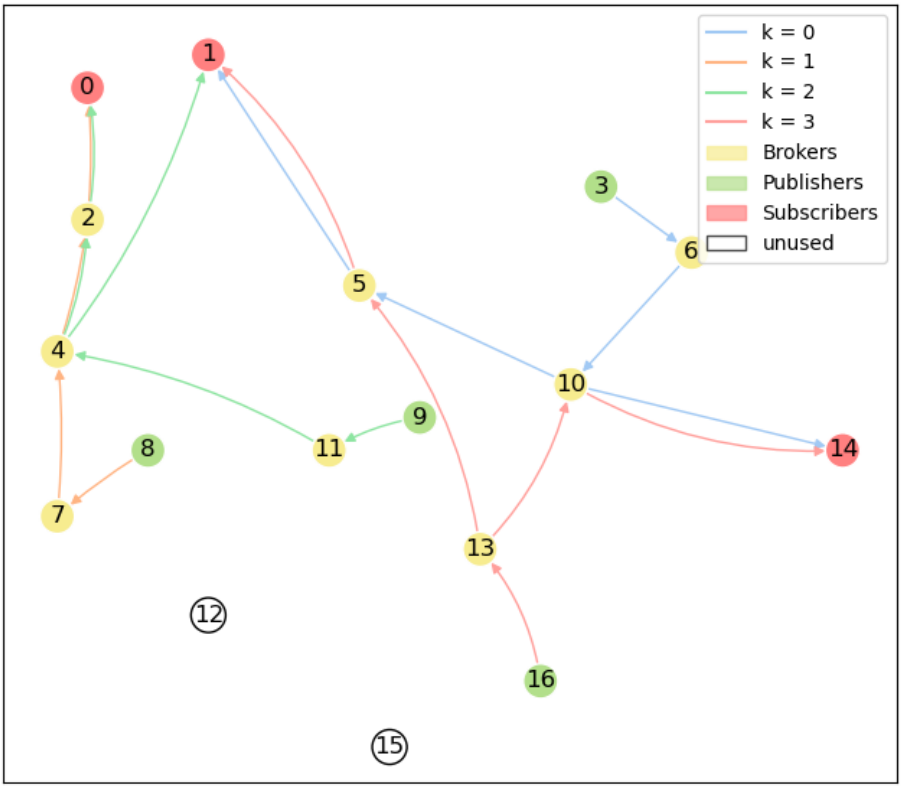

Figure 41: case6 solved with Formulation 2 using CPLEX

Case7 Optimal Solution with Formulation 2

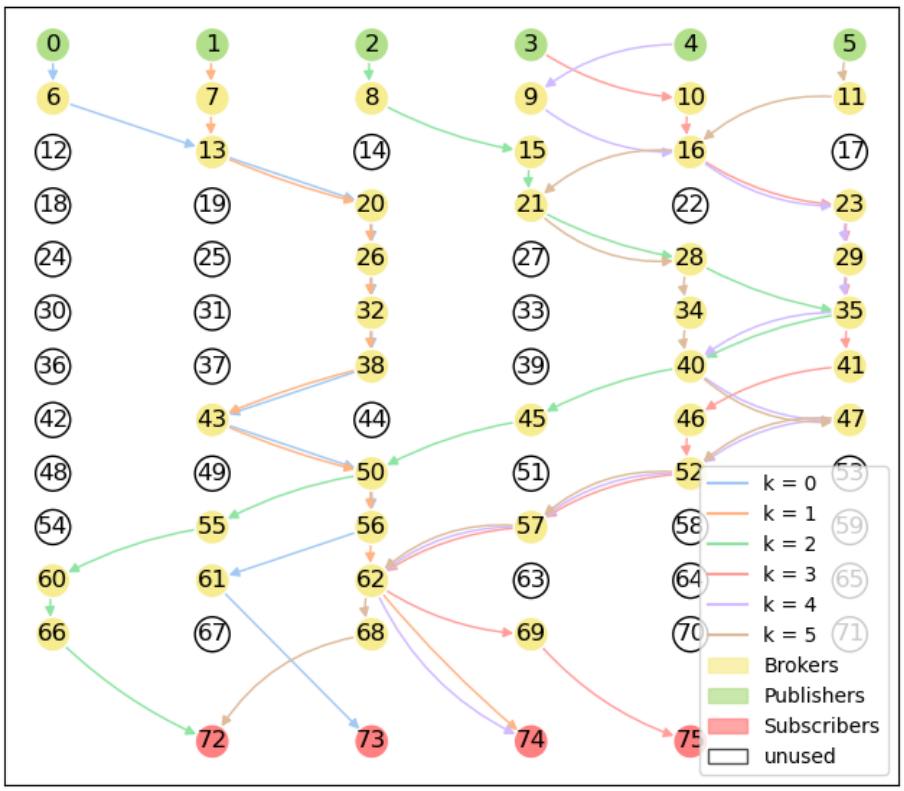

Figure 42: case7 solved with Formulation 2 using CPLEX

Case8 Optimal Solution with Formulation 2

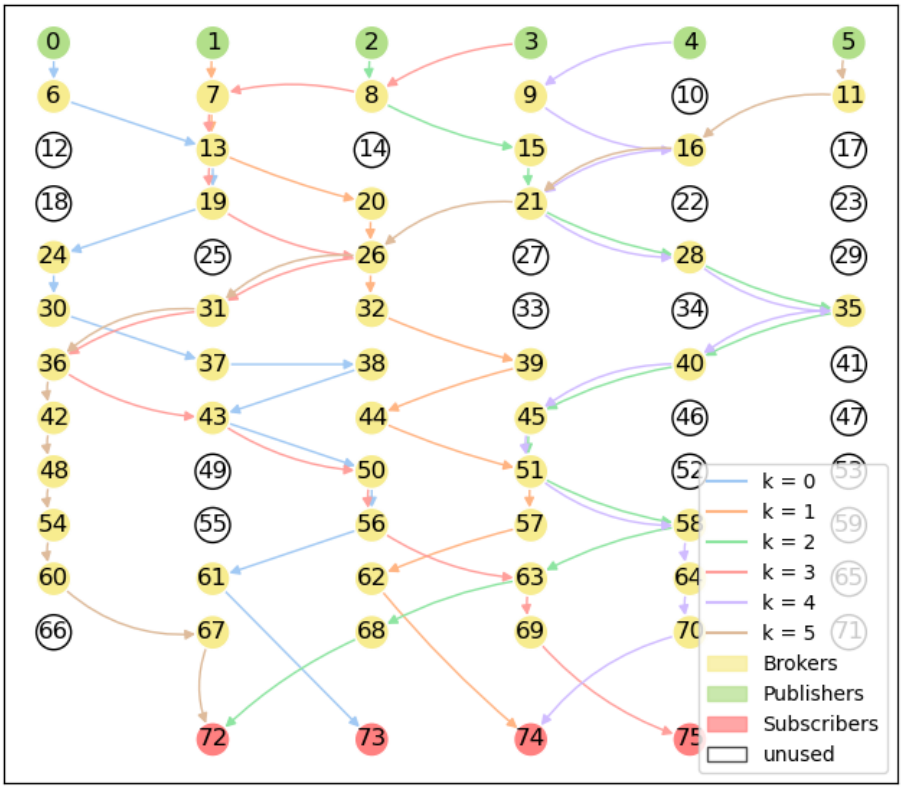

Figure 43: case8 solved with Formulation 2 using CPLEX

4.4 Simulation with Distributed Algorithm

In this section, we describe the results obtained through the use of the distributed Lagrangian heuristic algorithm presented in Section 3.7. The performance of the system is analyzed and compared with that of the previously described algorithms, along with the rationale behind the development choices made. In Section 3.7, two different approaches were described for addressing the problem. The first approach is primarily based on the request of commodities by brokers, while the second approach focuses on a combination of requests and the sending of commodities by brokers. In this study, these approaches are compared to determine the most promising one.

It should be noted that if subscribers are not satisfied after a certain number of iterations (which can be translated into time limits in realistic applications), for example, due to channel saturation caused by non-optimal choices, they will signal the need for a network reset to formulate new solutions.

The possible admissible solutions for the considered instances are numerous, especially with the increase of nodes and connections between them. In order to make a fair comparison, average parameters were extracted from the algorithm's execution five times and the obtained average values were evaluated. The considered parameters include:

- Average cost of the found solution
- Average number of resets performed to obtain solutions
- Average time required to find a solution

The cost of a solution can be understood as the latency for streaming data between publisher and subscriber. This means that high solution costs result in a delay between the generation of data by subscribers and the reception by subscribers. However, once the connection is established, the continuous flow of data between source and destination is guaranteed. It should also be noted that the simulations performed require nodes to perform their functions serially. In a real-world setting, nodes would work concurrently, thus reducing the time required to obtain solutions. It would therefore be interesting for future developments to evaluate multithreaded simulations, in which node work is parallelized.

4.4.1 Results of Distributed Heuristic Algorithm - First Apporach

In summary, the algorithm involves the initial sending of commodities by publishers and requests by subscribers to the network brokers that are considered most convenient based on the penalized weight of the connections. Brokers that receive requests from subscribers (or other brokers) work to establish a connection with brokers that have received from publishers, by sending requests to neighboring brokers that are considered most convenient in the network (for more information, see Section 3.7.1).

The algorithm finds feasible solutions for the given cases. It should be noted that case 2 presents high-cost connections (1000) compared to the average, between subscriber 6 and broker 0 and subscriber 7 and broker 1 (see Figure 18). These connections are used when the available bandwidth of the most convenient channels is saturated. In such situations, the cost of the solutions is particularly disadvantageous compared to the optimal case.

To estimate the computing time, we compute the average time required to execute the algorithm on 50 tests. Then, the number of times all nodes need to be active at least once to find a feasible solution was extracted. Finally, these two values were multiplied to obtain an approximate estimate of the time required to reach a feasible solution.

Table 9 shows the costs obtained from the simulations performed for the different instances and their average value, using the first approach. Table 10 shows the number of network resets required to obtain feasible solutions for the different instances and the average value calculated using the first approach. Finally, Table 11 provides an estimate of the average computing time for obtaining solutions using the first approach.

Instance	*cost*	*cost*	*cost*	*cost*	*cost*	**Mean**
Tiny	16	20	16	16	16	16.8
case2	68	1045	68	67	68	263.2
case3	219	336	218	317	240	266
case4	121	126	127	143	132	129.8
case5	34	34	34	34	34	34
case6	130	115	100	110	100	111
case7	385	380	370	395	400	386
case8	400	525	485	510	505	485

Table 9: Results of costs and mean cost of solutions obtained with first approach

Instance	*n.reset*	*n.reset*	*n.reset*	*n.reset*	*n.reset*	**Mean**
Tiny	0	0	0	0	0	0
case2	0	0	0	0	0	0
case3	0	0	0	0	0	0
case4	0	0	0	0	0	0
case5	0	0	0	0	0	0
case6	0	0	0	0	0	0
case7	1	1	3	2	1	1.6
case8	2	1	2	3	3	2.2

Table 10: Results of number of reset and mean number of reset with first approach

Instance	*time [ms]*	*time [ms]*	*time*	*time [ms]*	*time [ms]*	**Mean**
Tiny	2.22	3.33	2.22	2.22	2.22	2.44
case2	32.12	32.12	12.85	12.85	25.69	23.12
case3	57.49	114.98	34.49	57.49	34.49	59.79
case4	44.39	73.98	73.98	73.98	44.39	62.14
case5	5.64	5.64	5.64	5.64	5.64	5.64
case6	14.97	14.97	49.90	14.97	29.94	24.95
case7	6896	7688	14358	5540	7236	8343
case8	9142	5877	8209	13993	14366	10317

Table 11: Time estimation and mean time for obtaining solutions with first approach

As evident from the results, while this approach provides mostly feasible and high-quality solutions for the instances provided, it struggles to find solutions for large-scale problems with a high number of nodes and connections. This results in suboptimal response times for subscribers' requests. It is important to note, however, that these are simulations where nodes perform their tasks in a serial and not parallelized manner.

It is intuitive to understand that for large-scale problems, brokers face significant difficulties in finding the right path to reach nodes with the required commodity availability. Moreover, brokers may make suboptimal choices, saturating channels and making it impossible to satisfy all requesting nodes. Consider the example shown in Figure 44, where on the left, there is a situation where it is not possible to send all commodities to the requesting subscriber due to channel saturation, while the optimal situation is shown on the right.

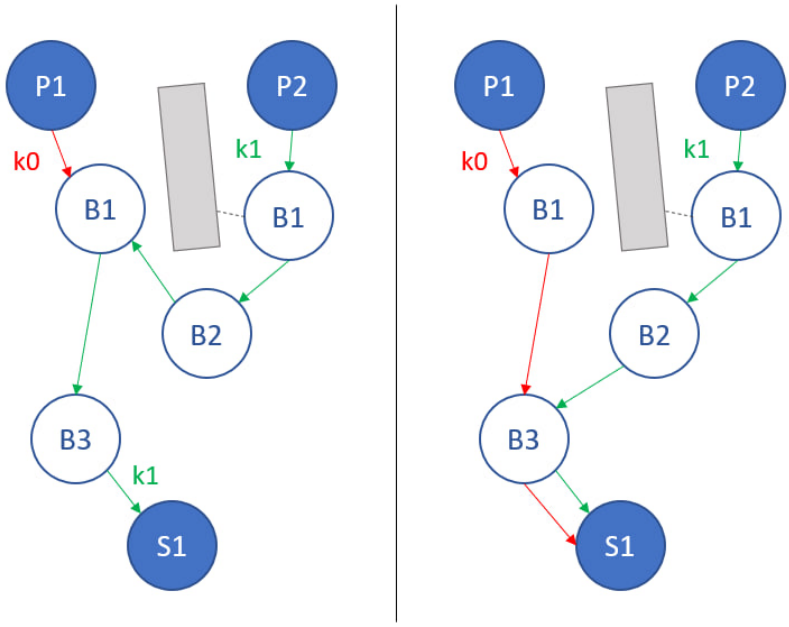

Figure 44: Example of bad decisions and good decisions made by brokers of the network

One of the reasons that led to the development of a new approach is related to the difficulty encountered in the previous approach, where brokers had to rely solely on subscriber requests to send commodities into the network. In the subsequent approach, however, efforts were made to improve the situation by also leveraging the resources of brokers with available commodities but without requests from subscribers. In this new approach, these brokers actively try to push commodities into the network, hoping to facilitate connections with requesters and increase the chances of satisfying all requests.

4.4.2 Results of Distributed Heuristic Algorithm - Second Approach

This new approach involves brokers sending requests, but unlike the previous approach, it also involves those brokers who have commodities available sending them into the network, even if they haven't received any requests (as described in Section 3.7.2). The goal of this strategy is to simplify the task of brokers in providing a connection between subscribers and publishers, especially in medium-sized networks that require a significant number of hops.

As with the previous approach, average parameters were extracted to evaluate the effectiveness of the algorithm and its performance, such as average cost, average number of resets, and average computing time to reach solutions for different instances.

We estimate the average computing time required to execute the algorithm on 50 tests. Then, the mean number of times the algorithm was executed to find a solution to the problem was extracted from 5 tests. Finally, these two values were multiplied to obtain an approximate estimate of the time required to reach a feasible solution. Unlike the previous algorithm, this algorithm does not examine all nodes at each iteration, but only works on nodes that have commodity requests or availability, making the execution more efficient.

Table 12 shows the costs obtained from simulations performed for different instances and their mean value using the second approach. Table 13 shows the number of network resets needed to obtain feasible solutions for different instances and their mean value calculated over 5 tests, using the second approach. Finally, Table 14 provides an estimate of the average time required to obtain feasible solutions using the second approach.

Instance	*cost*	*cost*	*cost*	*cost*	*cost*	**Mean**
Tiny	16	16	20	16	16	16.8
case2	1045	68	67	67	68	263
case3	141	160	139	334	140	182.8
case4	128	126	132	126	141	130.6
case5	34	34	34	34	34	34
case6	125	110	110	95	100	108
case7	410	385	370	375	365	381
case8	505	529	434	461	435	472.8

Table 12: Results of costs and mean cost of solutions obtained with second approach

Instance	*n.reset*	*n.reset*	*n.reset*	*n.reset*	*n.reset*	**Mean**
Tiny	0	0	0	0	0	0
case2	0	0	0	0	0	0
case3	0	0	0	0	0	0
case4	0	0	0	0	0	0
case5	0	0	0	0	0	0
case6	1	0	0	0	0	0.2
case7	0	0	0	0	0	0
case8	1	0	1	0	0	0.4

Table 13: Results of number of reset and mean number of reset with second approach

Instance	*time[ms]*	*time[ms]*	*time[ms]*	*time[ms]*	*time[ms]*	**Mean**
Tiny	5.47	10.64	8.82	6.08	6.08	7.42
case2	18.76	18.76	12.10	12.10	17.55	15.85
case3	307.40	198.32	208.24	198.32	287.56	239.97
case4	117.83	63.23	112.09	74.72	175.31	108.64
case5	13.32	13.32	13.32	13.32	13.32	13.32
case6	87.52	150.90	150.90	108.65	69.41	113.48
case7	1812	2271	849	1583	1354	1574
case8	1384	8759	2402	1243	1045	2967

Table 14: Time estimation and mean time for obtaining solutions with first approach

The results obtained show that, although the solution costs are comparable (with slight average improvements for approach 2), there is a significant increase in the probability of finding feasible solutions, reducing the time required to solve the given problems. Furthermore, it is highlighted that the need for reset actions in this new approach is almost non-existent. Additionally, these results can be evaluated through the graphs represented in images 45 and 46 below, which demonstrate a significant improvement in terms of time required to obtain the solutions.

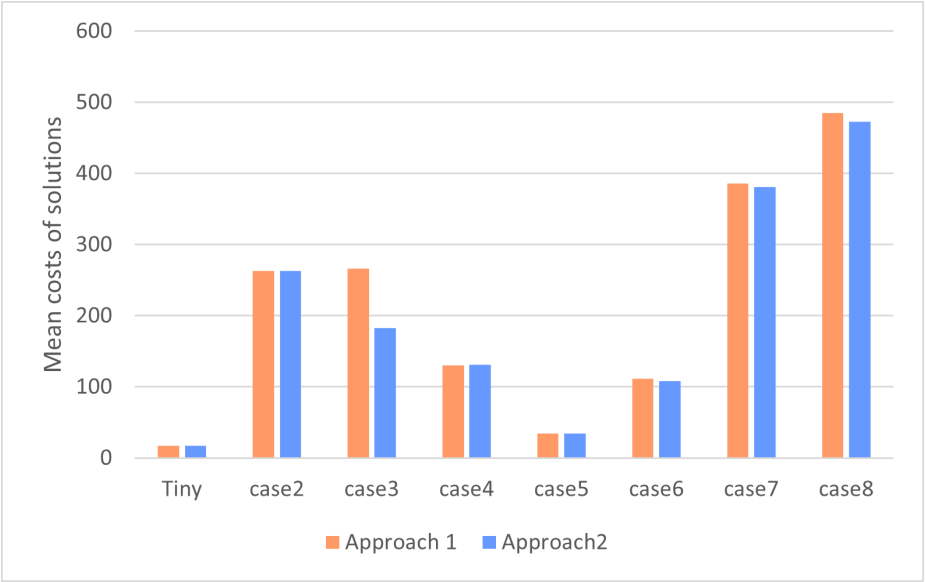

Figure 45: Comparison of approaches 1 and 2 on the average cost of solutions

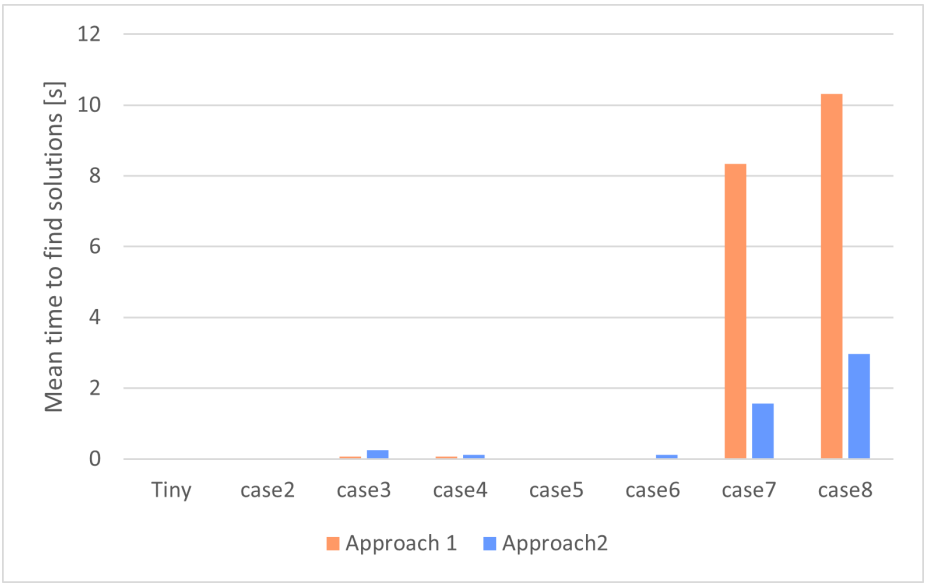

Figure 46: C Comparison of approaches 1 and 2 on the average times to obtain solutions

As a further comparison between the two different approaches described for the problem, the average number of times in which these heuristic algorithms were able to find optimal solutions in the different instances was also extracted, averaging over 5 simulations (Table 15).

Instance	**Approach 1**	**Approach 2**
Tiny	80.0%	80.0%
case2	20.0%	40.0%
case3	0.0%	0.0%
case4	0.0%	0.0%
case5	100.0%	100.0%
case6	0.0%	20.0%
case7	0.0%	0.0%
case8	0.0%	0.0%

Table 15: Percentage of optimal solutions found

4.4.3 Dynamic Adaptability of the Distributed Algorithm

This ability to adapt to changes is crucial for the efficient and reliable operation of distributed systems. In traditional centralized systems, a single point of failure can bring the entire system down, resulting in data loss and service disruption. However, in distributed systems, the workload is spread across multiple nodes, reducing the impact of any single node failure. Furthermore, the dynamic adaptation of nodes in a distributed system can improve its fault tolerance, scalability, and performance. When a new node is added to the network, it can join the system without disrupting the existing nodes. Similarly, when a node fails, the remaining nodes can quickly reconfigure their connections and redistribute the workload to maintain the system's overall functionality. In addition to its adaptability, distributed systems also offer increased privacy and security. By decentralizing the control of data and resources, distributed systems can reduce the risk of data breaches and unauthorized access to sensitive information.

As an example, suppose node 8 of instance 5 has failed, causing it and all its connections to become unavailable. Initially, this node was crucial for subscriber 11, which required commodities k1 and k2, as shown in Figure 47.

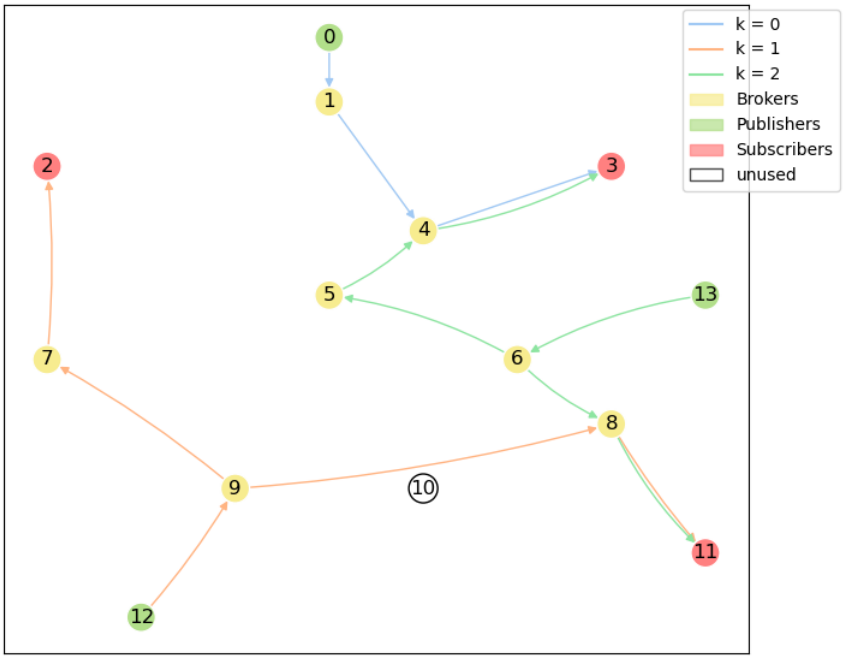

Figure 47: Optimal Solution Instance 'case5' before node 8 failure

As a result, subscriber 11 is left without the requested commodities, while nodes 6 and 9 respectively have commodities k1 and k2. However, these nodes reconsider the cost of their available connections and establish new connections that are more cost-effective. Eventually, they connect to node 10, which was initially inactive due to its less advantageous connections with nearby nodes, thus allowing subscriber 11 to receive the data.

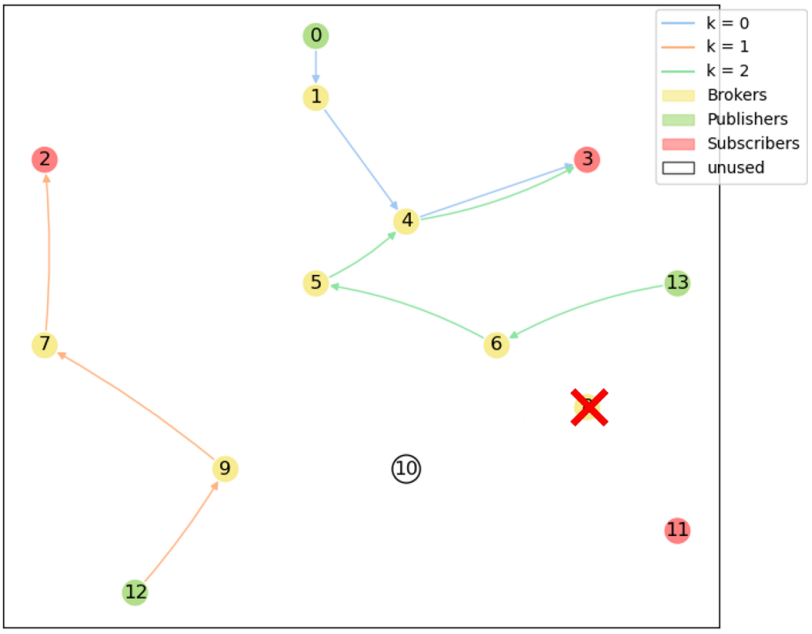

Figure 48: Situation on Instance 'case5' immediately after node 8 failure

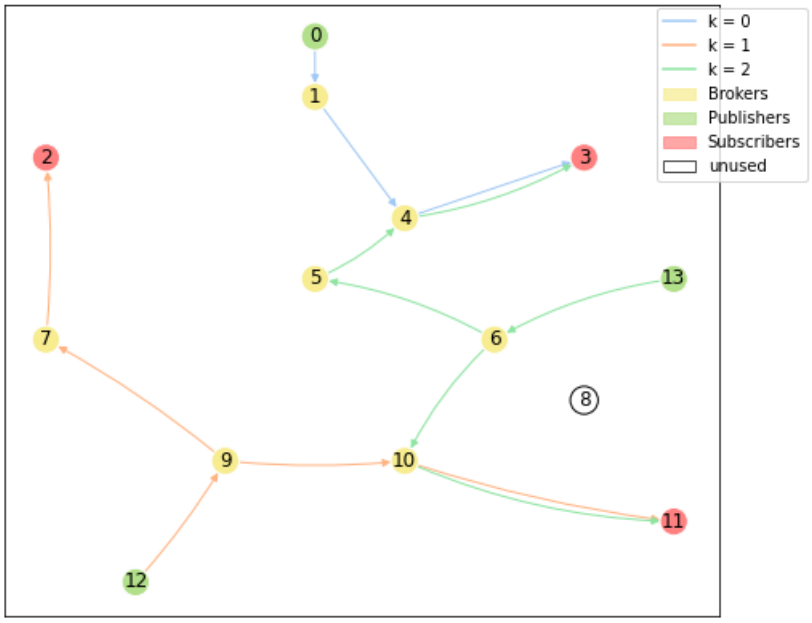

Figure 49: New Solution found for Instance 'case5' after node 8 failure

The cost of the solution increases from 34 initially to 43 after reconfiguration, which results in slightly higher latency for reestablishing data

streaming from publisher to subscriber after the failure. However, the network still ensures the ability to meet all subscriber requests. This depends on the availability of brokers for connection and/or alternative paths to connect subscribers and publishers. It is therefore important to anticipate these situations in the network design phase to ensure the ability to maintain a stable connection in any situation. Otherwise, temporary suspensions of data streaming and a slight increase in latency for data reception may occur.

4.4.4 Comparison of results obtained by all algorithms with Formulation 2

In this section, the results obtained from different techniques used to solve the optimization problem will be presented for comparison. In particular, the cost results of the centralized algorithm based on Formulation 2 and utilizing CPLEX as the solver will be analyzed and compared with the average results obtained through the use of continuous and Lagrangian relaxation techniques for the optimization problem. Additionally, the average results obtained from the two distributed approaches will be examined to evaluate the performance differences between the different methodologies adopted.

Instance	**Optimal**	**Cont. Rel.**	**Lag. Rel.**	**App. 1**	**App. 2**
Tiny	16	16	16	16.8	16.8
case2	67	66	64.49	263.2	263
case3	135	130.5	125.52	266	182.8
case4	112	112	98.87	129.8	130.6
case5	34	22	18.84	34	34
case6	95	70	65.79	111	108
case7	345	345	236.68	386	381
case8	365	353	262.62	485	472.8

Results obtained from Distributed Heuristic Algorithm implemented with approach 2

Tiny Instance Solution with approach 2

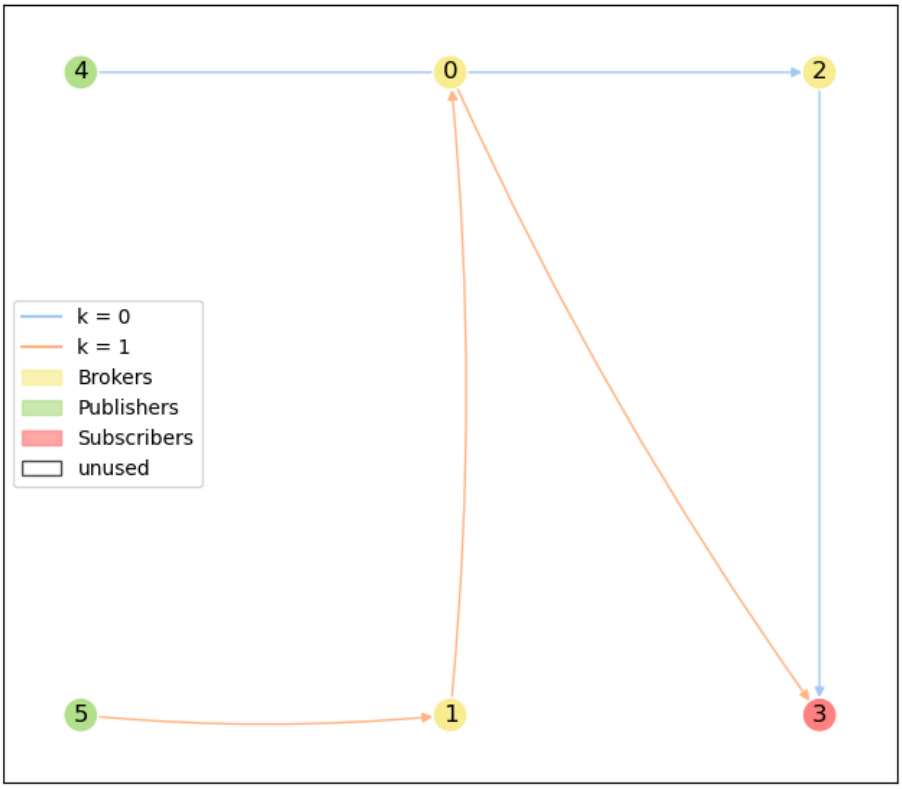

Figure 50: case Timy solved with second approach of Distribute Lagrangian Heuristic

Case2 Instance Solution with approach 2

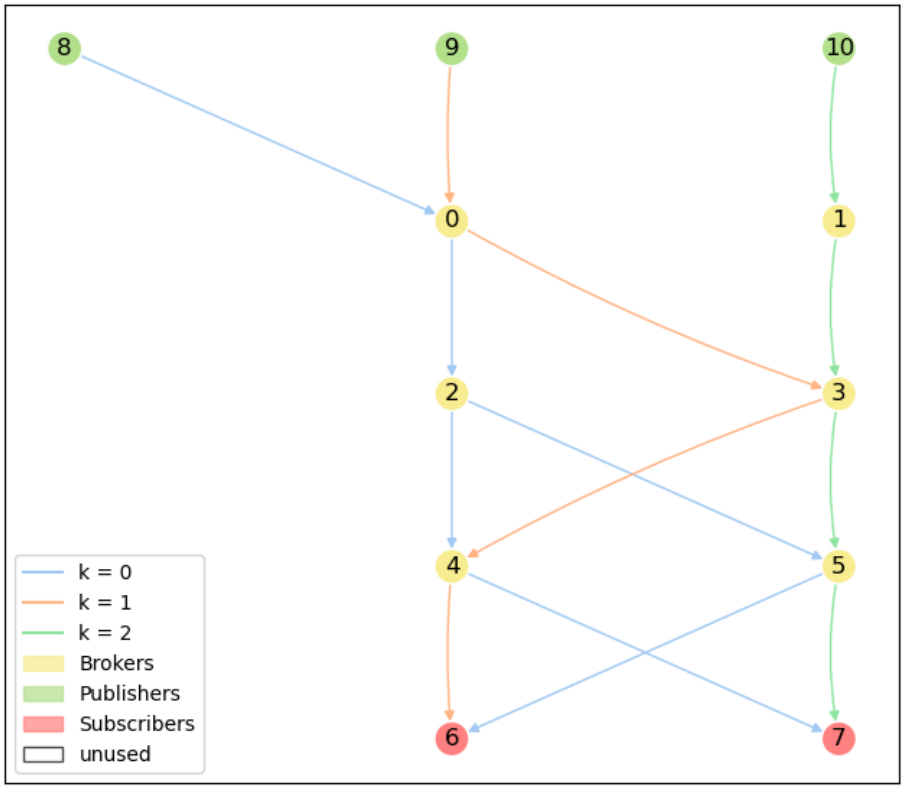

Figure 51: case2 solved with second approach of Distribute Lagrangian Heuristic

Case3 Instance Solution with approach 2

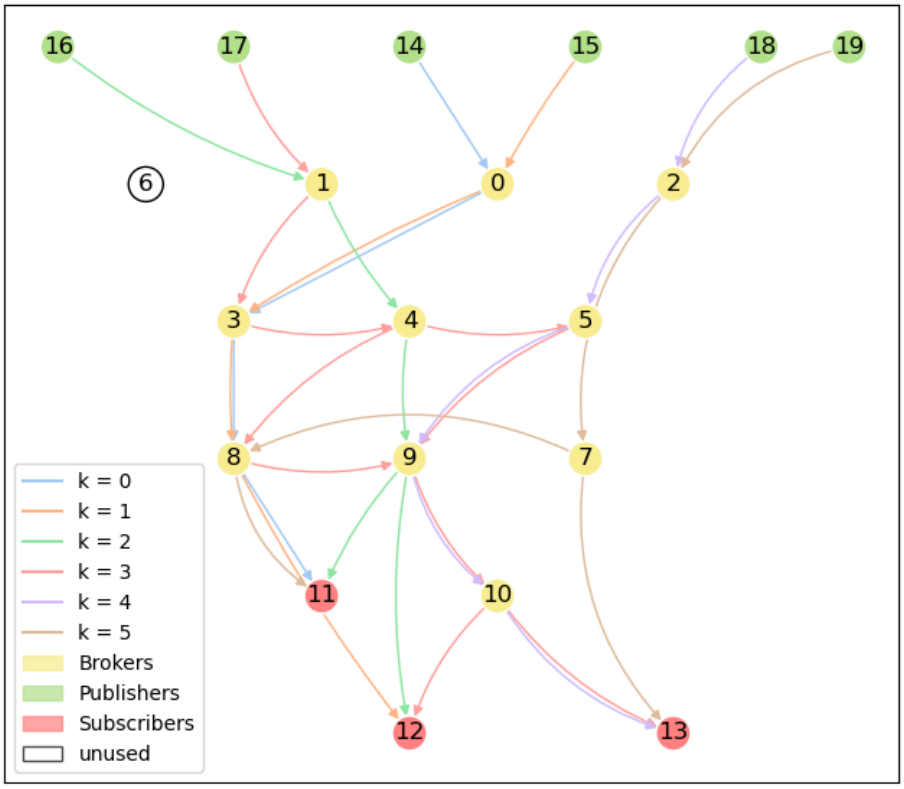

Figure 52: case3 solved with second approach of Distribute Lagrangian Heuristic

Case4 Instance Solution with approach 2

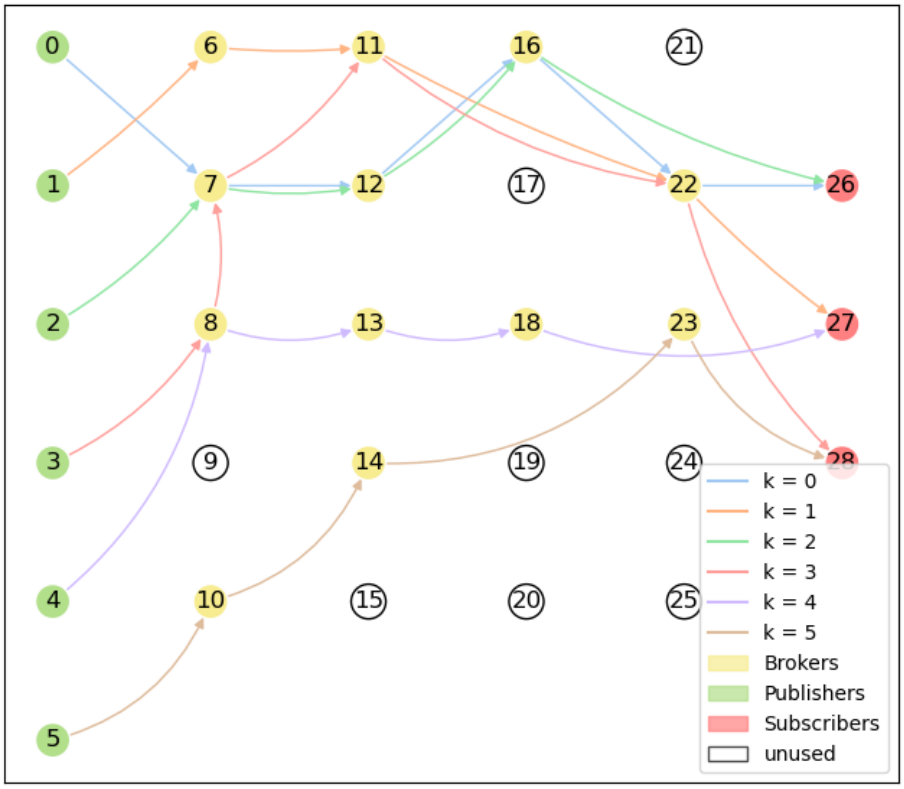

Figure 53: case4 solved with second approach of Distribute Lagrangian Heuristic

Case5 Instance Solution with approach 2

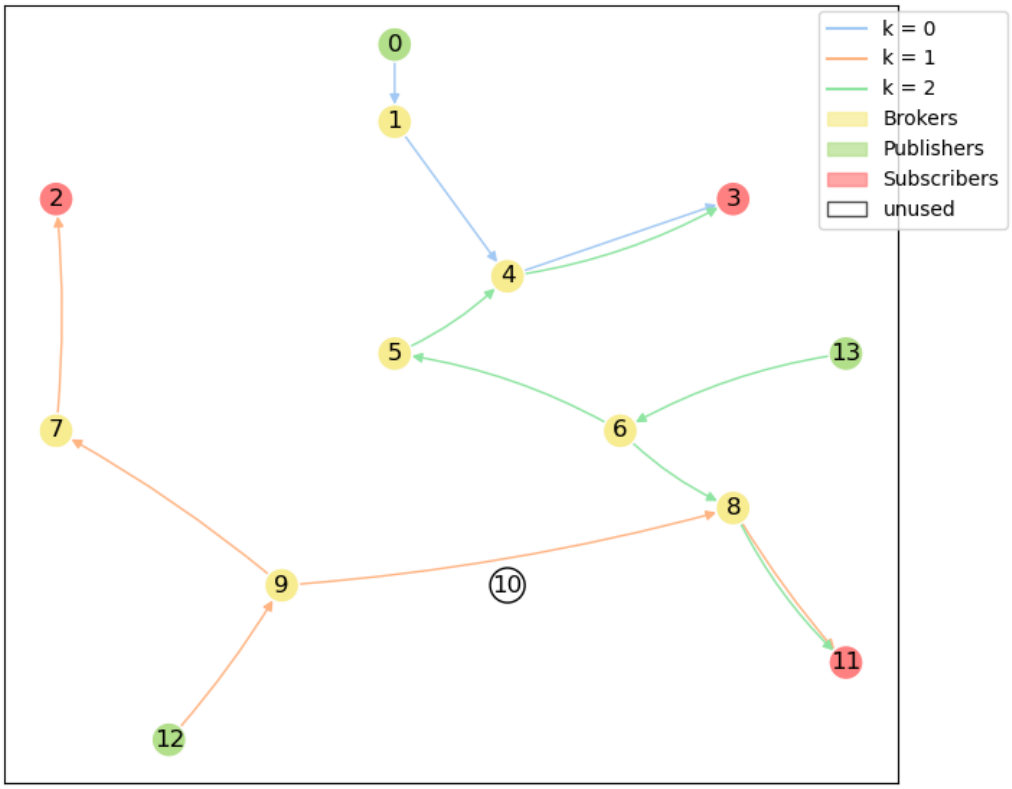

Figure 54: case5 solved with second approach of Distribute Lagrangian Heuristic

Case6 Instance Solution approach with 2

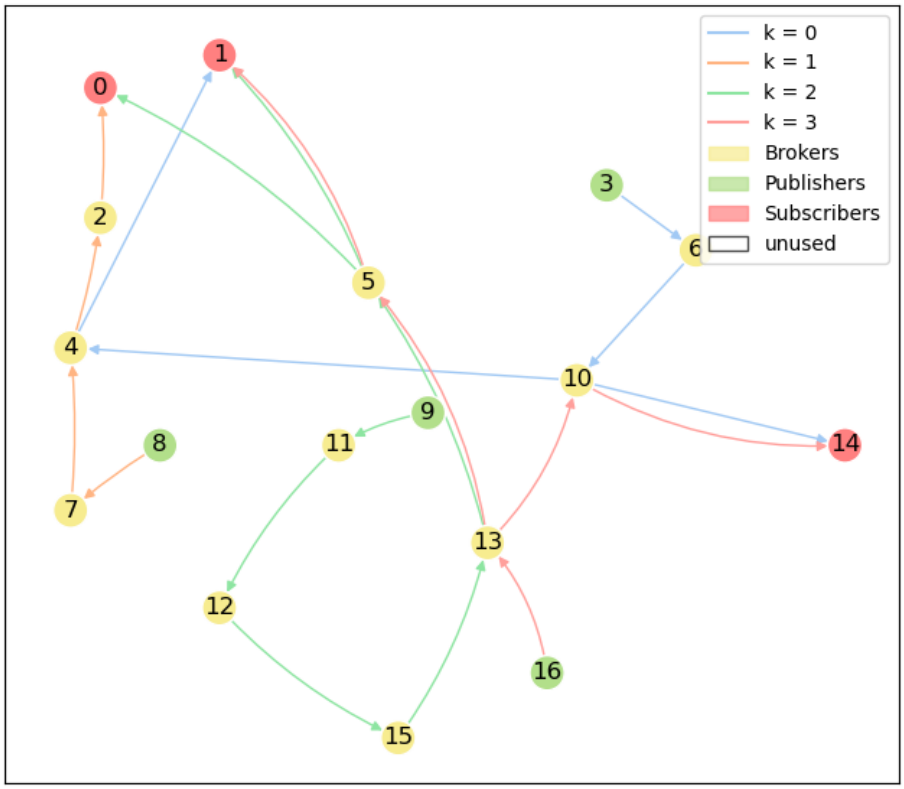

Figure 55: case6 solved with second approach of Distribute Lagrangian Heuristic

Case7 Instance Solution with approach 2

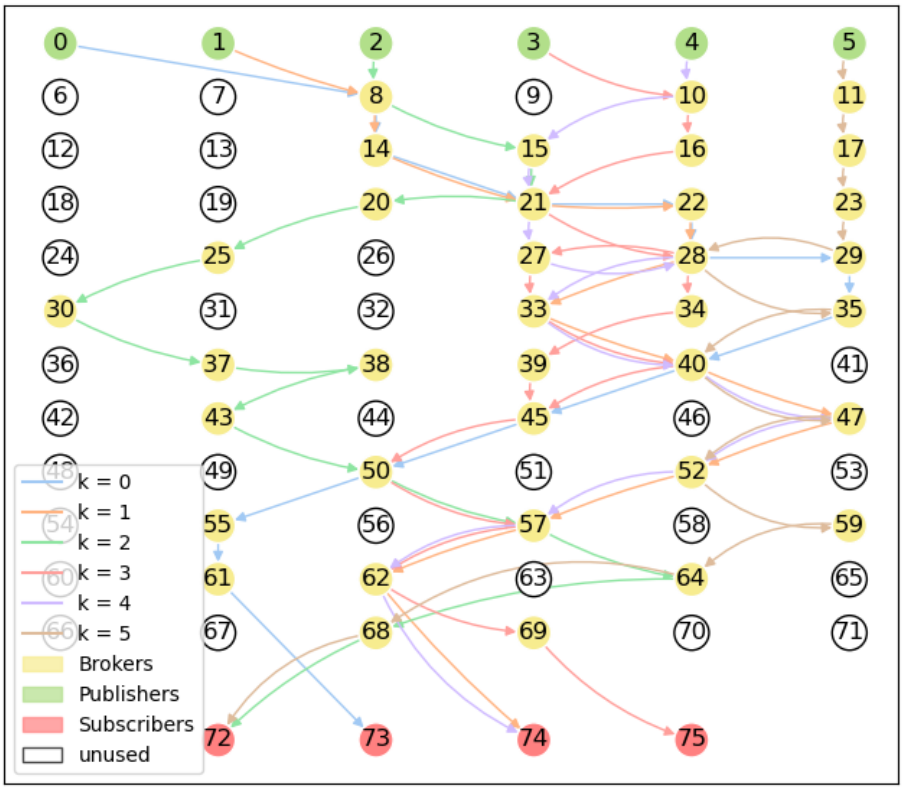

Figure 56: case7 solved with second approach of Distribute Lagrangian Heuristic

Case8 Instance Solution with approach 2

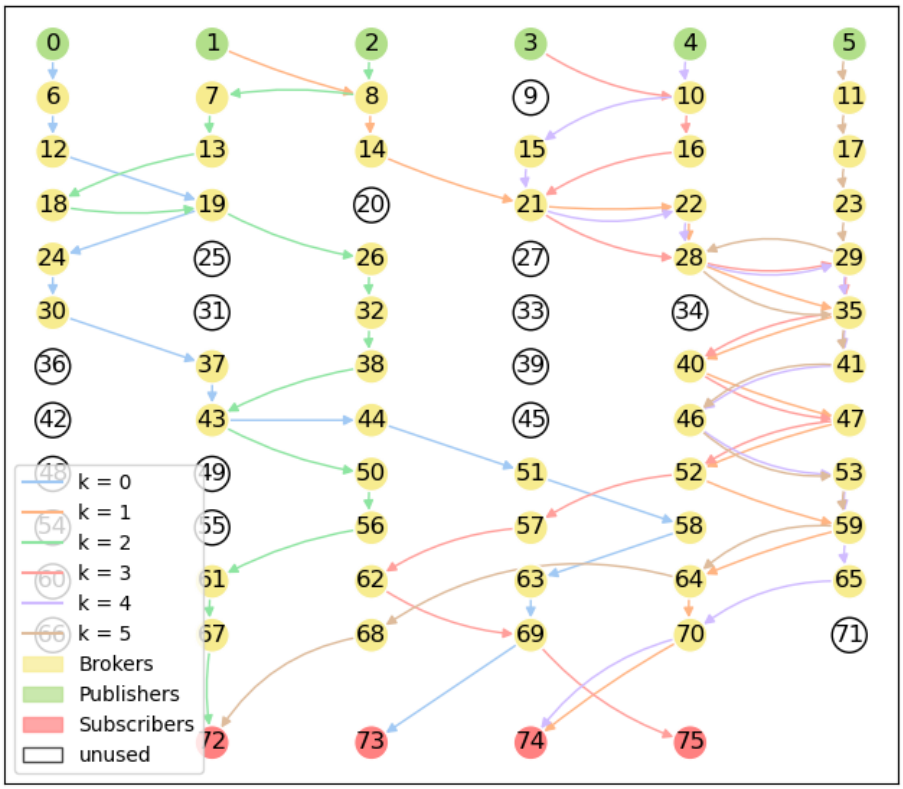

Figure 57: case8 solved with second approach of Distribute Lagrangian Heuristic

5 Conclusions

This thesis highlights how the IoT field is constantly evolving to meet the growing needs for information exchange and acquisition. In particular, the thesis explores the opportunity to develop distributed algorithms based on MQTT protocols using multiple brokers acting as a single entity, leveraging the concept of peer-to-peer.

The decentralized heuristic algorithms described in this thesis are mainly based on Lagrangian relaxations, which allow each node in the network to compute its own penalties and receive those of neighboring nodes, facilitating the selection of effective heuristic solutions for data transmission between network senders and receivers. Thus, the thesis demonstrates the feasibility of developing high-performance distributed algorithms in the IoT environment using relaxation techniques.

The thesis examines various aspects of the optimization problem and demonstrates the effectiveness of relaxation techniques in breaking down the problem into simpler subproblems, simplifying the search for solutions. The proposed implementation also guarantees dynamic adaptability of connections to network variations, allowing for connection in case of link or node failures or the addition of new nodes.

The implemented scripts were developed using the Python programming language in Jupiter Notebook (Anaconda), while the linear programming solver used is CPLEX for the centralized version that seeks optimal solutions to problems.

These distributed algorithms based on relaxation techniques have multiple areas of use, both in indoor and outdoor environments. They are particularly useful in situations with low computing power and limited network resources, as they guarantee a light code footprint and reduced memory usage, since each node only needs to know its own information and that of its neighbors. For example, they can be used for monitoring environmental conditions in rural areas, such as for forest fire prevention. Thanks to the sensors that establish a stable connection with requesting machines, it is possible to process the collected data and analyze it, providing continuous control of the monitoring site thanks to dynamic connection adaptability, allowing for timely intervention in case of need and preventively.

Possible future developments of this work may include improving the efficiency of the implemented algorithms through the use of multithreaded simulations. This would allow network nodes to perform their operations in parallel, rather than serially. Additionally, it would be interesting to create simple hardware structures (such as using physical devices like Raspberry Pi) to test the functioning of these algorithms in a real-world environment.

The research in this field is increasingly moving towards edge computing, which involves the use of machine learning in solutions called TinyML, i.e., machine learning technologies capable of performing data analysis on low-power devices. Additionally, AI-based IoT solutions, called edge or fog computing, are being considered for areas with limited connectivity and resources. However, it must be acknowledged that current IoT infrastructures are not yet ready for this decentralization, and further research is needed to integrate fog computing functionalities while maintaining the ease of use and high availability of existing infrastructures.

In summary, the design of distributed networks represents a topic of great interest for many applications, such as sensor networks for environmental monitoring or smart city management. The solution proposed in this thesis could represent a starting point for the development of similar solutions in other application contexts. Research in this field is still evolving, and there will certainly be further developments in the future.

References

[1] Marco Antonio Boschetti, Vittorio Maniezzo, *Some Notes on Pub/Sub Brokers Readjustment. A Multi-Commodity Flow Approach*

[2] Marco Antonio Boschetti, Vittorio Maniezzo, *A Fully Distributed Lagrangean Solution for a Peer-to-Peer Overlay Network Design Problem*, INFORMS Journal on Computing 2011.

[3] Pietro Manzoni, Marco Antonio Boschetti, Vittorio Maniezzo, *Modeling Distributed MQTT Systems Using Multicommodity Flow Analysis*, Electronics 2022.

[4] Hua Wei, Hong Luo, Yan Sun, *A New Cache Placement Strategy for Wireless Internet of Things*, Journal of Internet Technology Volume 20 (2019) No.3.

[5] Marco A. Boschetti, Vittorio Maniezzo, Matteo Roffilli, *A Fully Distributed Lagrangean Solution for a Peer-to-Peer Overlay Network Design Problem*, INFORMS Journal on Computing 23(1), 90-104, 2011.

[6] Khalil Chebil, Mahdi A. Khemakhem, *A dynamic programming algorithm for the Knapsack Problem with Setup*, December 2015 Computers & Operations Research.

[7] Adeel Javaid, *Understanding Dijkstra Algorithm*, January 2013 SSRN Electronic Journal.

Desidero esprimere la mia profonda gratitudine al mio relatore, il Professor Marco Antonio Boschetti, per la sua preziosa guida e il suo costante supporto durante lo svolgimento di questo lavoro. Grazie alla sua competenza e al suo sostegno, ho potuto sviluppare le mie capacità e raggiungere questo traguardo accademico. Inoltre, desidero esprimere la mia gratitudine a tutti gli amici che mi hanno accompagnato in questo percorso di studi e che mi hanno sostenuto nei momenti più difficili. In particolare, vorrei ringraziare Giacomo per aver condiviso con me ogni sfida e per il suo sostegno costante, e Federico, amico di lunga data e compagno di studi, per i suoi preziosi consigli e il suo incondizionato supporto. Un ringraziamento speciale va alla mia famiglia, Marco, Paola e Lisa, per il loro costante incoraggiamento e supporto. Grazie alla vostra generosità, ho avuto l'opportunità di coltivare la mia passione per l'apprendimento e la crescita personale. Senza il vostro sostegno, non avrei mai potuto raggiungere questo traguardo. Vi sarò sempre grato per tutto ciò che avete fatto per me. Infine, desidero ringraziare tutti coloro che in modo diretto o indiretto hanno contribuito al successo di questo percorso formativo. Grazie di cuore.