A Lightweight Heuristic for Micro-services Placement and Chaining in Low Latency Services

Hichem Magnouche*, Guillaume Doyen†, Caroline Prodhon*
*LIST3N, University of Technology of Troyes, Troyes, France, {first.last}@utt.fr
†SOTERN - IRISA (UMR CNRS 6074), IMT Atlantique, Rennes, France, guillaume.doyen@imt-atlantique.fr

***Abstract*—The rise of novel Low-Latency (LL) applications, such as cloud gaming or the metaverse, imposes rigorous end-to-end LL constraints. Decomposing Virtualized Network Functions (VNFs) into micro-services has proven its effectiveness to reduce the Service Function Chaining (SFC) latency thanks to key characteristics: lighter entities, less resource consumption, and a strong capacity to operate in parallel. However, to make such a promising technology actually deployed in real operated networks, novel dedicated placement and chaining methods are required. Current solutions either do not fit with tied LL constraints or exhibit a prohibitive computation time by relying on exact resolution methods. In this paper, we propose a heuristic method dedicated to the placement and chaining of micro-services. Its purpose is to maximize the deployment of SFCs while respecting the required LL by considering intrinsic features of micro-services and integrating suitable load balancing, which makes it highly scalable. A comprehensive evaluation campaign highlights that generated solutions achieve results that are at most a factor of 1.1 to the optimal with an execution time up to 20,000 times faster.**

***Index Terms*—Heuristic, Micro-Services, Placement and Routing, Optimization, Orchestration**

I. Introduction

The rise of novel internet applications, such as tele-surgery, cloud gaming and the Metaverse, has led to new latency requirements, which must not exceed a few milliseconds. This imposes unprecedented challenges on the global telecommunication infrastructures for both traffic forwarding and processing, and various means are leveraged to date to minimize the traffic latency. Regarding traffic forwarding, mitigating congestion points with dedicated active queue management [1] strategies coupled with adapted congestion control algorithms such as TCP-Prague [2] enable end-to-end LL. Besides, traffic processing implemented by VNFs offered a significant advancement to allow flexible and customizable deployment of network services. However, while numerous studies [3] addressed Service Function Chains (SFCs) orchestration approaches to optimize the placement and chaining of VNFs, they either do not consider LL requirements at the core of their operation. Additionally, many rely on monolithic VNFs, which aren't ideal for LL services. To overcome these limits, the micro-services approach, initially applied in the cloud domain [4], has been introduced in the context of VNF. This architectural pattern functionally decomposes monolithic VNFs into smaller components called micro-services, thereby facilitating the placement of SFCs. By mutualizing redundant functions and parallelizing certain packet processing operations, they significantly contribute to the latency reduction.

Given this set of novel features (i.e. mutualization and parallelization), state-of-the-art orchestration solutions become obsolete as they do not integrate them into their computation algorithms. In a previous work [5], we proposed a Mixed-Integer Linear Programming (MILP) model for the placement and chaining of micro-services, which leverages the mutualization of redundant processing and adaptive parallelization depending on the network infrastructure setup. Although exhibiting relevant results regarding the meeting of LL constraints and success rate of micro-service placements, the exact method considered for the model resolution exhibited an extremely long time. This is not compatible with operational constraints of a telco even in small case scenarios.

To overcome this scalability issue, this paper presents an innovative heuristic approach for micro-service placement and chaining, effectively pooling redundant services and optimizing parallelization. By leveraging lightweight algorithms, it proves to be efficient even for large-scale scenarios. Leveraging a $Waterfilling$ approach [6] and an online learning method, our method especially optimizes placement load balancing to enhance the respect of LL requirements, while averting potential bottlenecks related to server saturation leading to scaling issues. To assess our approach, we examine the gain and cost induced by its features in a step-by-step manner and compare them to the exact resolution method results. We also demonstrate that the computation time of our solution is low making it appropriate for large-scale operational scenarios while maintaining a satisfying balancing quality.

The paper is organized as follows. Section II provides a review of recent literature on SFCs orchestration approaches, VNF parallelization, and load balancing. Section III details the proposed heuristic and the various algorithms that compose it. Section IV benchmarks our heuristic method through a comprehensive series of evaluation tests performed on its different features along with their respective analysis. Finally, Section V concludes the paper by summarizing our main observations and findings.

II. Related Work

The issue of SFCs placement and chaining has been extensively studied in the literature with various optimization models leveraging the exact solution method, via solvers, or approximate methods, using heuristic algorithms, for their resolution. Besides, several improvements dedicated to LL applications have been developed: VNF decomposition into

978-3-903176-59-1 ©2023 IFIP

micro-services, parallelization, and load balancing during deployment. Thus, in the following, we review each aspects.

A. Micro-services: Architectures and Orchestration

The benefits of micro-services have been demonstrated in [7] as a solution to the limitations of monolithic VNFs, such as overlapping functionalities, light weightiness, loss of CPU cycles, and lack of flexibility in scaling. As a result, several micro-services architectures have been developed and implemented [8], [9], which primarily use lightweight container virtualization and zero-copy of packets with DPDK [10] to enhance performance. In [11], the authors examine the impact of container scaling and identify a latency increase when using multiple containers due to packet flow between them. In [12], the authors discuss the pros and cons of using micro-services and evaluate their execution latency in comparison to the monolithic approach. They subsequently propose an AI-based architecture to determine the reuse, creation, or duplication of micro-services during deployment. [13] proposes MicroNF, a framework that utilizes the benefits of micro-services to address the performance degradation caused by monolithic NFV by leveraging three elements: (1) SFCs reconstruction to re-factor micro-services when possible; (2) micro-services placement to consolidate them on the same node; and (3) scaling approach to balance the load between network nodes.

Mutualization is the process of grouping two or more identical micro-services belonging to the same SFC into a single one to reduce its length and latency while saving memory and CPU resources. This concept has been studied in [13], [14] as part of the functional decomposition of VNFs into micro-services, which show that mutualization of two identical micro-services is not always possible. It requires checking the set of micro-services from the original SFC to ensure that the processing of packet data will not be affected by leveraging a mutualization table, as proposed in [13]. Besides, [14] also exposes a micro-services framework including a mutualization algorithm. However, it does not propose any solution for their placement and chaining, which is mandatory to make this contribution operational.

B. VNF Placement and Chaining Heuristics

The VNF placement and chaining problem is mainly tackled by two strategies. Exact methods, like the Simplex algorithm [15], offer precise results but require long computation times. By contrast, approximate methods, based on heuristics or metaheuristics, offer near-optimal quicker solutions. Despite potential sub-optimality, the faster results of approximate methods make them often preferred in practice.

Existing heuristics for VNF placement and chaining often break down the problem into more manageable steps. For example, [3] tackles it by modeling the problem as a multi-tiered graph and then utilizing the Viterbi algorithm [16] to deploy VNFs. However, these methods face a significant challenge: the dynamic parallelization of VNFs, which is unknown before deployment, complicates the creation of a multi-level graph. By contrast, Askari *et al.* present in [17] a shared-use approach. Their heuristic examines each VNF placement of existing instances, chooses the closest to source and destination, or, if absent, calculates the shortest path and deploys a new one. This method may however lead to non-optimal solutions, increasing the traffic latency.

Besides, most studies propose a per SFC approach to deploy VNFs on the shortest path. Notably, [18], [19], follows this idea by using Dijkstra's algorithm to calculate the shortest path. In cases where the deployment is not possible on some nodes, they propose to recalculate another one and attempt the deployment again. Similarly, Hirewe *et al.* [18] suggest to remove the most overloaded node, while Gadre *et al.* [19] suggest to remove the highest latency arc before recalculating the new shortest path. However, the latter may not necessarily be the next shortest path, which is a limitation of this approach. Several optimal methods for k shortest path calculation exist too: the modified Dijkstra algorithm, A*, Bellman-Ford-Moore, and Yen or Eppstein algorithms [20], [21]. The latter extends the Dijkstra algorithm and stands out by its lower complexity. It identifies the k shortest paths in a non-negative graph using a priority queue to store them. It first calculates the shortest paths from each node to the destination node using Dijkstra. After that, it proposes a new representation of the original graph, which facilitates efficient traversal either on the shortest path with zero-cost or via an auxiliary node, with the cost corresponding to the forwarding latency.

C. VNF Parallelization

The challenge of SFC latency reduction has sparked many innovations, including the parallelization of VNFs. In [22], the authors proposed a VNF parallelization algorithm through a dependency graph, enabling both internal and external parallelism. However, external parallelization can impact latency due to packet copying and merging. Conversely, the internal VNF parallelization can efficiently use shared memory, as suggested by DPDK [10]. [23] introduced internal parallelism for all SFCs located on the same node, mitigating copy/merge time through specific techniques. Meanwhile, [24] tackled the restrictions of [22], [23] by permitting node-level parallelism without conditions on the complete SFC, thus enhancing agility. In [25], a dual-phase approach was proposed to manage internal and external parallelization and minimize the latency of SFCs. The findings highlight that partial parallelism can decrease latency by up to 25% as compared to a serial deployment. However, establishing parallelization through heuristic methods in either pre- [22] or post-deployment [24], limits its optimization since it may not adapt to network capabilities. This could be overcome by considering the parallelization of VNFs during deployment.

D. Load Balanced Deployment of SFCs

The growth of low-latency services has encouraged load balancing research for SFC placement and chaining, which brings improved latency, increased stability, and traffic congestion prevention. Existing studies are divided into two groups: (1) leveraging VNF replication for load balancing on nodes and arcs, and (2) achieving network-wide load equilibrium without VNF replication or flow splitting.

For instance, [26] and [27] use packet flow division for load balancing. The former introduces a hash-based method that may lead to corrupted packets due to improper reassembly. The latter introduces two heuristics adjusting flow distribution based on network node load changes. Similarly, [28] addresses load balancing by optimizing VNF replication through an ILP model to minimize costs. Nevertheless this optimization, while supporting network balancing, might affect latency due to VNF duplication, risking network congestion and resource consumption surge. Furthermore, SFC flow handling between VNF replications can increase latency and impose technical limits. In the second group, [29] proposes a MaxMin-based placement heuristic deploying many VNFs on least loaded nodes. Despite balancing efficiency, this can cause topological mismatch leading to non-optimal SFC latency since least loaded nodes may be far from each other. Finally, the authors of [30] use a $Waterfilling$ problem-based [6] heuristic for efficient VNF placement on pre-calculated network paths. It computes a non-exceedable limit of memory usage for each node, redistributing placement requests upon limit reach. This non-duplicative approach overcomes the topology mismatch effect of [29]. However, the authors consider that the nodes should not be used at 100% to allow for certain limit overstepping during balancing.

Our review identifies a gap in the scalable and efficient placement and chaining of micro-services. Existing monolithic heuristics fall short due to the inherent characteristics of micro-services, such as their large quantity, small size, asynchronous operation, and robust parallel processing capability. Furthermore, most of them sub-optimally calculate shortest paths, compromising the solution performance. Finally, load balancing is often addressed in separated works, despite its potential benefits in terms of optimising latency. Consequently, we propose a two-step heuristic using an optimal algorithm to (1) compute the k shortest paths, before (2) deploying micro-services to promote efficient micro-services management and optimized load balancing.

III. A Micro-Services Placement and Chaining Heuristic

In this section, we present the micro-services placement and chaining problem that we tackle as well as the dedicated heuristic method we propose. The latter aims to: (1) place and chain the micro-services, (2) manage adaptive parallelization, (3) maximize the number of SFCs that meet the prescribed latency, and (4) optimize load balancing. To reach these objectives, our lightweight heuristic integrates several optimization techniques, exploiting the intrinsic characteristics of micro-services, which are introduced subsequently in a step-by-step approach.

A. Problem Statement

The micro-services placement and routing problem we investigate is defined by a network graph G = (N, L), where N represents a set of nodes n, characterized by a memory capacity M_n. L represents a set of links between two nodes n_i, $n_j \in N$ characterized by a latency link $\delta_{n_i n_j}$. Q is a set of SFC requests, with each request, $q \in Q$, characterized by a source and destination represented respectively by s_q, $d_q \in N$, a required latency rl_q.Additionally, each request has a set of micro-services, represented as $m \in M$, where M is the set of all types of micro-services that an edge flow must traverse. The objective of this problem is to:

- Place the micro-services for each SFC;
- Chain micro-services together.

Subject to :

- Memory capacity constraints on the nodes;
- Micro-services forwarding and execution latency constraints;
- Micro-services execution order constraints;
- Parallelism execution constraints.

Functions	Definition
$EppRep(s_q, d_q, G)$	Gives $Eppstein$ representation for paths from S_q to d_q in G
$NeShPa(ER_q)$	Finds next shortest path using $Eppstein$ rep. ER_q
$AvaiSpace(p)$	Computes available memory on a path p
$Place(m, n)$	Assigns micro-service m to node n
$Para(m1, m2)$	Checks if micro-services $m1$, $m2$ can run in parallel
$Cont(n, m)$	Checks if micro-service m is on node n
$IsCritical(q)$	Checks if SFC q length is less than its rl_q
$Move(m, n)$	Migrates micro-service m to node n
$TwiceDecr(q)$	Checks if SFC q score decreased twice consecutively
Sets	**Definition**
$q \in Q$	Set of SFCs
$m \in M$	Set of micro-services
$m \in PM_{m_n}$	Set of micro-services operating in parallel to m_n
Parameters	**Definition**
G	Network infrastructure for SFC deployment
ER_q	$Eppstein$ representation for SFC q
$NbIt$	Number of iterations for the online learning process
Δ_s	Constant to adjust SFC q score
N	Factor to amplify SFC q score deterioration
s_q	Source node of SFC q
d_q	Destination node of SFC q
Variables	**Definition**
sp_q	Shortest path for SFC q
as	Available space
mr	Space left on p post q deployment
mpn	Space left on p post q deployment per node
rm	Residual space on p post deployment
ca_n	Memory capacity of node n
m^-	Precedent of micro-service m
sc_q	SFC q score
rl_q	Required Latency of SFC q
el_q	Effective latency of SFC q

TABLE I: Functions, sets, variables, and parameters considered in our heuristic method

B. Algorithm Overview

The heuristic ($Algorithm$ 1), accepting pre-processed SFC set Q and network infrastructure G as inputs, generates a placement solution for all SFCs within the infrastructure capacity. Pre-processing involves mutualizing and identifying parallelizable micro-services, as detailed in our previous work [5]. Composed of three key algorithms, the heuristic utilizes $EppRep()$ and $NeShPat()$ for k shortest path calculation, and $Algorithm$ 2 for a load-balanced SFC deployment, which itself employs $Algorithm$ 3 to deploy micro-services while managing parallelization.

This section elaborates on the heuristic primary steps, special attributes, shortest path calculation, parallel placement and chaining procedures, and it introduces an online learning approach enhancing the heuristic performance. Table I provides a comprehensive overview of all parameters and variables we consider in the following.

1) Main steps of the heuristic: As shown in $Algorithm$ 1, to optimize the deployment of a set of SFCs, our heuristic starts by ordering the SFCs in ascending order of the margin between their required latency and the SFC length (line 1). Indeed, the smaller the latency gap, the more critical the SFC. Next, for each SFC $q \in Q$ (loop on line 2–11), the method executes two first operations: it computes a modified $Eppstein$ representation, which enables the calculation of the k loop-free shortest paths (line 3) and sets the flag dep_q to $false$ indicating that the SFC remains undeployed (line 4). Then, while SFC q is not deployed and a k^{th} shortest path is available (line 5), the heuristic determines the k^{th} shortest path based on the computed $Eppstein$ representation (line 6). Subsequently, it checks whether the memory capacity of the path is adequate for the deployment of SFC q (line 7). If the memory capacity is found to be sufficient, the heuristic attempts the deployment of SFC q by invoking $Algorithm$ 2 with the SFC q and shortest path sp_q parameters (line 8).

Algorithm 1 Overall heuristic method for micro-service SFC placement and chaining

```
Input: Set of SFC $Q$, infrastructure $G$
Output: Placement and chaining solution for all SFCs
1:  Order SFCs according to latency gap value
2:  for all SFC $q \in Q$ do
3:      $ER_q \leftarrow EppRep(s_q, d_q, G)$
4:      $dep_q \leftarrow$ false
5:      while $\neg dep_q$ and $NeShPa(ER_q).exist()$ do
6:          $sp_q \leftarrow NeShPa(ER_q)$
7:          if $Capacity(SP_q) \leq Length(q)$ then
8:              $dep_q \leftarrow SFCDeployment(q, sp_q)$ // Algorithm 2
9:          end if
10:     end while
11: end for
```

2) Shortest Path Computation: In a micro-service SFC placement and chaining scenario, determining only one shortest path per SFC may be insufficient due to bounded node memory. This necessitates the identification of the k shortest paths where, if the $(k-1)^{th}$ path lacks sufficient memory capacity, we deploy the SFC on the k^{th} path, maintaining flexibility and robustness. To optimally address this, we leverage the $Eppstein$ algorithm [21], an efficient solution for graphs without negative loops, which fits with our context where edge latency cannot be negative. $Eppstein$ algorithm complexity is $O(m + n\log(n) + k\log(k))$, with m, n, and k representing the number of edges, nodes, and calculated paths, respectively. The algorithm operates according to the following steps: (1) it starts with $Dijkstra$'s algorithm calculating the shortest paths between each node and destination; (2) it forms a unique graph, which allows a direct transit to the subsequent node, or a detour to an auxiliary node, with latency variation; (3) it outputs the shortest path, while empty auxiliary node sets represent direct paths. For each node on the shortest path, (4) it forms sets of shortest paths and adds feasible paths to a heap. Finally, (5) it extracts the minimum-cost set from the heap, computes their shortest path, and repeats until all paths are explored.

When graphs contain positive loops, shortest paths can be infinite due to possible loop traversals on each path calculation. Since repeated node traversal does not affect the path's micro-service deployment capacity, we need loop-free paths. Consequently, we modified the $Eppstein$ algorithm at two points to exclusively produce k shortest loop-free paths: one at step (2) to remove edges if the auxiliary node creates a loop with the original path, and at step (3) to eliminate looping paths. These modifications ensure paths are loop-free, maximizing $Eppstein$ algorithm's efficiency in our context. In our method, the $EppRep()$ function oversees the operation of the modified steps (1-2), while the $NeShPa()$ function manages the execution of the modified steps (3-5).

3) Load Balanced SFC Deployment: The deployment of SFCs produced by Algorithm 2 consists in optimising the placement of micro-services to maximise the number of SFCs deployed, while respecting the prescribed latency and balancing the load on the network nodes.

Our load balancing strategy, inspired by the $Waterfilling$ algorithm [6], aims to fairly distribute remaining space for post-SFC deployment on nodes. However, two challenges arise: (1) memory allocation indivisibility, which necessitates a strategy for managing integer division and leftover fractions; (2) the potential resource distribution imbalance due to micro-services parallelization, which demands an integrated approach for system balance maintenance. To achieve this balanced deployment, we propose to use four metrics: available space as ; the margin mr ; the margin per node mpn and the residual margin (rm), which are described in Table I.

The proposed procedure, described in Algorithm 2, employs an iterative method to traverse two lists: the list of micro-services associated with SFC q, indexed by m, and the list of nodes within the specified path p, indexed by n. Initially, the algorithm considers the first micro-service and node (line 1 and 2). While any SFC remains undeployed and nodes persist within the path, the heuristic executes the ensuing steps (line 7-16). First, it checks whether the current node can host the current micro-service (line 7). This assessment entails verifying if the available space surpasses the mpn metric. If the node proves to be sufficient, the algorithm invokes the $Algorithm$ 3 (step 9) to deploy the micro-service m onto node n. Subsequently, if the deployment is successful (line 10), it transits to the next micro-service while maintaining its position at the current node (step 11). Conversely, if the node cannot host the micro-service, the algorithm progresses to the next node without transitioning to the next micro-service (step 14). By implementing this procedure, the algorithm facilitates balanced micro-service deployment on path p by evenly distributing the remaining space among the nodes. One can notice that rm will be used by $Algorithm$ 3.

Algorithm 2 SFC Deployment

```
Inputs: SFC $q$, path $p$
Output: Deployment of SFC $q$
1:  int $n \leftarrow 0$ // index for iterating over path $p$
2:  int $m \leftarrow 0$ // index for iterating over SFC $q$
3:  int $as \leftarrow AvaiSpace(p)$
4:  int $mr \leftarrow as$ - $Length(q)$
5:  int $mpn \leftarrow Quotient(marge/Length(p))$
6:  int $rm \leftarrow Remainder(margin/Length(p))$
7:  while $m \leq Length(q)$ and $n \leq Length(p)$ do
8:      if $ca_n > mpn$ then
9:          $dep_q \leftarrow$ µ$ServicesPlacement(q, p, m, n)$ // Algorithm 3
10:         if $dep_q$ then
11:             $m \leftarrow m + 1$
12:         end if
13:     else
14:         $n \leftarrow n + 1$
15:     end if
16: end while
```

4) Micro-services Placement and Internal Parallelization: To leverage the internal parallelization of micro-services for latency reduction, we propose a strategy, described in Algorithm 3, that begins by placing a micro-service m on node n (line 1), then assess whether the preceding one m^- can feasibly operate in parallel with m. If so and if not already operating in parallel (line 3), the algorithm checks if m^{-1} is deployed on n (line 4) and if so, it integrates m^{-1} to the parallelizable set PM_m (line 5). If m^{-1} is not deployed on n, it verifies if (i) there is enough residual margin rm that could be used to avoid unbalancing the deployment too much, or (ii) if SFC q is critical, meaning that the parallelization is crucial to satisfy the latency constraint. Then the algorithm checks also if path p has enough capacity to move m^- to node n (line 6). Indeed, the parallelizing m with m^{-1} involves the migration to n, which leads to imbalances in the load deployment. In cases where the conditions are met, the algorithm moves m^- to n (line 7) and integrates m^- to PM_m (line 8). In the other case when m is parallelizable with m^{-1} and the latter is already operating in parallel with another(s) micro-service(s), the algorithm checks if all micro-services within $PM_{m^{-1}}$ are capable of running in parallel with m (line 10). If so, and m^- has already been deployed on n (line 11), the algorithm integrates m into the existing group $PM_{m^{-1}}$ of parallelizable micro-services of m^- (line 12). If m^- is not placed on the same node (line 13), the algorithm performs the same checks as line 6, but with the set of micro-services operating in parallel with m^-, contained in PM_{m^-}. In cases where the conditions are met, the algorithm moves them to n including m^- (line 16) and integrates m to PM_{m^-} (line 18).

Algorithm 3 µServices Placement

```
Inputs: SFC $q$, path $p$, micro-services $m$, node $n$
Output: optimized deployment of micro-service $m$
1:  $Place(m, n)$
2:  if $Para(m, m^-)$ then
3:      if $PM_{m^{-1}} = \varnothing$ then
4:          if $Contains(n,m^-)$ then
5:              $PM_m \leftarrow PM_m \cup m^{-1}$
6:          else if $rm > 0$ or ($IsCritical(q)$ and ($ca_p$ - $Length(q)$)) $> 0$ then
7:              $Move(m^-,n)$
8:              $PM_m \leftarrow PM_m \cup m^{-1}$
9:          end if
10:     else if $\forall\ \mu_i \in PM_{m^{-1}}$, $Para(m, \mu_i)$ then
11:         if $Contains(n,m^-)$ then
12:             $PM_{m^{-1}} \leftarrow PM_{m^{-1}} \cup m$
13:         else
14:             if $rm > Length(PM_{m^-})$ or ($IsCritical(q)$ and ($ca_p$ - $Length(q)$) $> Length(PM_{m^-})$ then
15:                 for all $\mu_j \in PM_{m^-}$ do
16:                     $Move(\mu_j,n)$
17:                 end for
18:                 $PM_{m^{-1}} \leftarrow PM_{m^{-1}} \cup m$
19:             end if
20:         end if
21:     end if
22: end if
```

C. Ordering the SFC Processing

A notable drawback of heuristic methods over the exact one relies in their sequential SFCs deployment, which does not optimize the entire SFC set concurrently as in mathematical models. This reveals that the processing order has significant impact on the solution performance. Consequently, we employed an online learning process to optimally order the SFC processing, as described in Algorithm 4.

The heuristic starts by uniformly assigning for each SFC in Q a consistent score "s" and it computes a modified *Eppstein* Representation (line 1-2). Then, at the start of each iteration $it \in NbIt$, whose upper bound has been empirically assessed to half of the number of SFCs per instance (line 5), the heuristic deploys them in ascending order of scores unlike latency gap value as detailed in Section 1 (line 6). After deployment, it checks each SFC's latency compliance (line 8). If an SFC meets latency requirements, its score is increased, thus lowering its priority (line 9). If it does not, its score is reduced, thus raising its priority (line 11). Over multiple iterations, the solution progressively improves. However, if an SFC consistently fails to meet latency requirements despite its priority, its score is intentionally increased after two consecutive decreases, downgrading it (line 12-13). This ensures the heuristic does not prioritize SFCs that persistently violate latency specifications.

Algorithm 4 Online learning approach for micro-service SFC placement and chaining

```
    Inputs: set of SFCs, infrastructure
    Output: deployment of all SFCs
 1: for all SFC $q \in Q$ do
 2:     $sc_q \leftarrow s$ // assigns identical score to all SFCs
 3:     $ER_q \leftarrow EppRep(s_q, d_q, G)$
 4: end for
 5: while $it \leq NbIt$ do
 6:     // Deploy all SFCs as per Algorithm 1 except that the
        ordering is based on the $s$ score instead of the latency
        gap value
 7:     for all SFC $q \in Q$ do
 8:         if $el_q \leq rl_q$ then
 9:             $sc_q \leftarrow sc_q + \Delta_s$ // score downgrading
10:         else
11:             $sc_q \leftarrow sc_q - \Delta_s$ // score improvement
12:             if $Rank(q) = 1$ and $TwiceDecr(q)$ then
13:                 $sc_q \leftarrow sc_q + N * \Delta_s$ // major deterioration score
14:             end if
15:         end if
16:     end for
17:     $it \leftarrow it + 1$
18: end while
```

IV. Evaluation

To assess the performance of our heuristic method under diverse scenarios, we implemented it in C++. The implementation consists of 2300 lines of code accessible at: https://www.mosaico-project.org/outcomes for replication purposes. To validate our implementation, we performed comprehensive checks, ensuring that (1) there are no circuits, (2) all micro-services related to each request are deployed, (3) the sequence of micro-services is respected, (4) memory resource consumption is respected, (5) latency computation is accurate, and (6) parallelism and mutualization are correctly implemented in accordance with their respective tables. All our experiments were conducted on an 11th generation Intel Core i7-1165g7@2.80GHz 1.69GHz computer, with 16GB of RAM and operating on Windows 10 Professional Education.

A. Evaluation Scenarios

Our performance evaluation scenarios aim at (1) understanding the performance of our model in realistic situations and (2) comparing it with the optimal solution ($CPLEX$) generated with the model we proposed in [5] and having the same optimization objective, which is to maximize the number of SFCs respecting latency constraints. We also evaluate the relevance of the different improvements we exposed above, namely: Heuristic without parallelization, online learning approach and load balancing (*H*); Heuristic managing parallelization (*H-P*); Heuristic managing parallelization and load balancing (*H-PB*); (*H-PLB*) being the last and standing for our selected candidate, which manages online learning approach in addition to the version (*H-PB*).

Parameter	Range or value
Topology	DFN-Verein European Telco
VNFs	Firewall, NAT, Traffic monitor, IPS
Micro-services	Read (Rd), Header Classifier (HC), Modifier (Md), Alert (Al), Drop (Dp), Check IP Header (CIH), HTTP Classifier (HC), Count URL (CU), Payload Classifier (PC), Output (Out)
SFC latency	5-10ms according to the SFC
Link latency	1ms
Micro-services proc. latency	1ms
SFC length	5-14 micro-services
Node capacity	3-10 instances of micro-services
k	10
sc_q	set to the number of SFCs in the scenario
Δ_s	1
N	set to the number of SFCs in the scenario

TABLE II: Evaluation parameters

All the parameters we considered in our evaluation are summarized in Table II and motivated subsequently. The implemented topology, extracted from the SNDlib[1] library, is that of the DFN-Verein European telco. We have partitioned it by selecting only some Point of Presence (PoP) for a given region. Then, each region has been split into two layers: one node acting as a regional PoP connected to other regional PoP according to the telco topology. Additionally, this node serves as an aggregation point for a few local nodes connected to it through a regional loop forming a ring sub-topology. The different SFCs we consider reflect those that can be found in the dedicated literature [31], each of them being splitable into micro-services. Their splits are technically realistic and derived from the literature [13], [14]. As the core benefits of micro-services, we have considered the mutualization and parallelization tables illustrated in [22]. We have considered the mutualization and parallelization tables illustrated in [22] as the core benefits of micro-services. The initial number of shortest paths to be computed, denoted as k, is set to 10. In case the algorithm is unable to deploy the SFC on one of these paths, it calculates the next 10. This process continues until a deployment is possible or no more shortest paths are available. As for sc_q, it is defined as the number of instances per scenario. This enables a homogeneous ranking of the SFCs. Variable Δ_s is set to 1, which is enough to prioritize one SFC over another. Finally, N is also defined as the number of SFCs per scenario. This definition is crucial to cause a significant deterioration in the ranking when necessary. All the results presented subsequently are the mean of eight repetitions bounded with 95% confidence intervals.

B. Result Analysis

In our evaluation, we use several metrics to assess the heuristic's performance and its variants. Primary indicators include the quantity of SFCs exceeding the required latency and the mean latency per scenario. We also study the average optimality gap, representing the heuristic solution deviation from the optimal. We estimate the computation duration by modifying the number of SFC instances per scenario and

[1]Survivable fixed telecommunication Network Design – http://sndlib.zib.de/

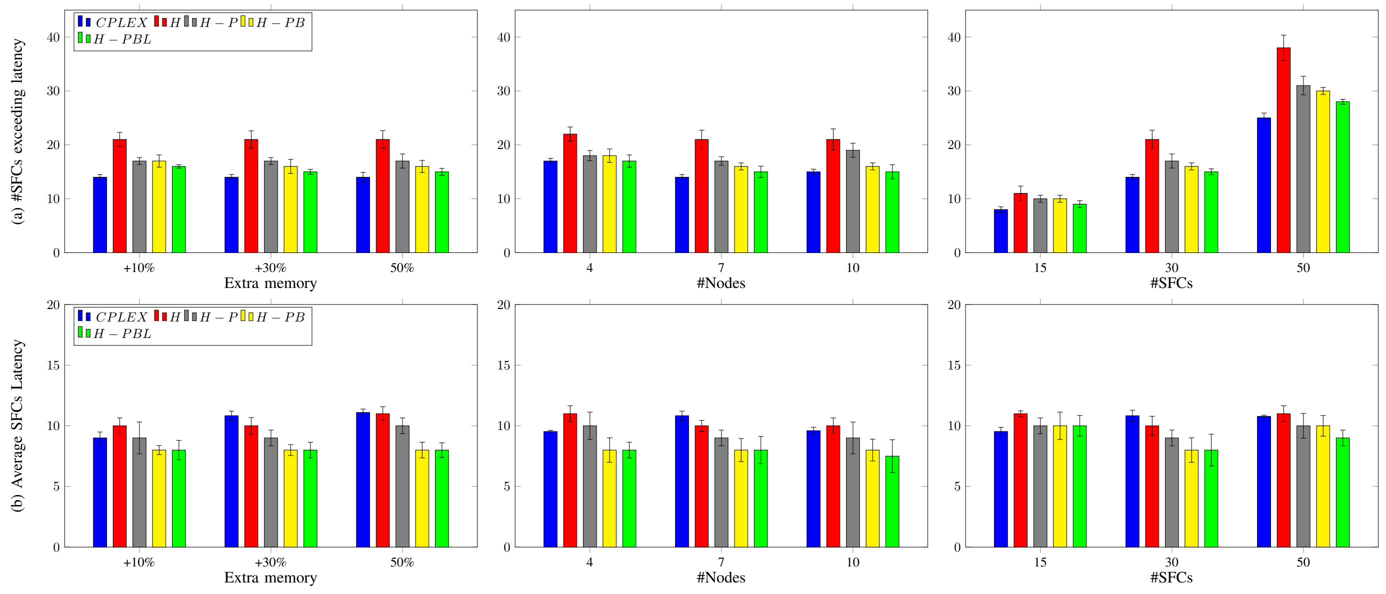

Fig. 1: Histogram of exceeded SFC latency and average SFC latency under CPLEX and heuristic approaches, with varying extra memory (left), infrastructure nodes (middle), and SFC per instance (right)

comparing the results to the exact approach's ($CPLEX$) computation duration. Additionally, we scrutinize the computation time by adjusting the number of nodes while keeping the SFCs instances constant. Finally, we assess load balancing with the Jain index, an acknowledged metric in the related literature [32], [33]. It quantifies deployment load balance with an index varying from 0 (unbalanced) to 1 (balanced).

1) Number of SFCs Exceeding Latency: Reviewing all scenarios of Figure 1.a highlights that, as expected, the exact $CPLEX$ method outperforms others, specifically the H approach, which lacks optimization and neglects micro-service peculiarities. The $H - P$ variant shows marked improvement due to effective parallelization during deployment. The load-balancing version, $H - PB$, optimizes performance in 6 out of 9 scenarios. The remaining scenarios, constrained by limited memory infrastructure, maintain performance levels, since the impact of load balancing decreases. However, when the infrastructure is not overloaded, load balancing distributes available space efficiently across the network, facilitating SFC deployment on the shortest paths, hence respecting latency. Finally, online learning approach ($H - PBL$) enhances the heuristic performance by countering its sequential limitations on all scenarios. In terms of metric variation, with more extra memory, the performance slightly improves due to increased placement and parallelization possibilities. However, when more nodes maintain the same total capacity, the performance of $H - P$ deteriorates as the possibility of parallelization decreases because of fixed memory capacities, which means that less space per node is available. Nevertheless the performance increases for $H - PBL$ thanks to the load balancing approach. Concerning the SFC number variation, we note that the gap between the exact approach $CPLEX$ and heuristic versions (H) and ($H - P$) notably widens when SFCs increases. However, our heuristic ($H - PBL$) maintains a consistent difference with $CPLEX$, proving its robustness regardless the number of scenarios.

2) Average Latency: As depicted in Figure 1.b, this analysis reveals a paradoxical behavior. Indeed, the exact approach ($CPLEX$) performs better when looking at the number of SFCs that exceed latency, especially when compared to our heuristic approach. However, when considering the average execution times for all latencies, CPLEX has the poorest performance. This phenomenon is explained by the fact that in the exact approach, when an SFC exceeds the latency, the model does not limit its exceeding value. By contrast, the heuristic optimizes each SFC by trying to minimize its latency, thereby enabling a more uniform deployment in terms of latency compliance. We notice that the average latency improves with the $H - PBL$ approach compared to the $H - PB$ approach. This improvement is especially evident in scenarios with 10 nodes and those with 50 SFCs. The reason is that the benefits of SFC ordering are more pronounced in these specific cases.

3) Optimality Gap: To refine the performance analysis of the solutions generated by our heuristic method, we evaluate the optimality gap as summarized in Table III. This metric measures the distance between the solutions generated by our heuristic and the optimal solution produced by the exact method. Our method achieves a latency gap of 11%, which is significantly commendable as compared to the literature, where the latency gap typically ranges between 13% and 14% [34], [35]. Interestingly, each optimization, whether it is parallelization, online learning, or load balancing, contributes to its own margin of improvement. However, parallelization reduces the most the optimality gap (over one half), which is due to its strong tied to micro-services approach.

4) Computation Time as a Function of SFC Number: Figure 3.a illustrates the evolution of computation time according

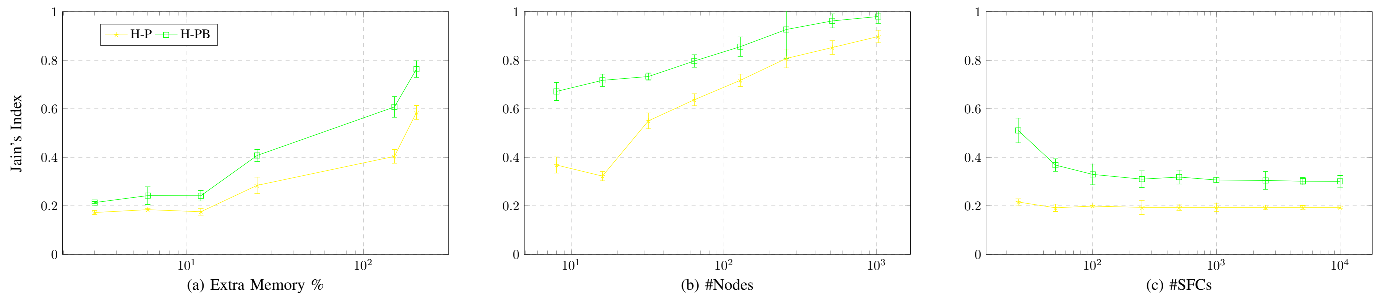

Fig. 2: Jain index for micro-services deployments as a function of (a) the extra allocated memory, (b) number of nodes and (c) number of SFCs

	H	$H-P$	$H-PB$	$H-PBL$
Optimality gap (%)	43	19	17	11

TABLE III: Optimality gap for different versions of heuristics

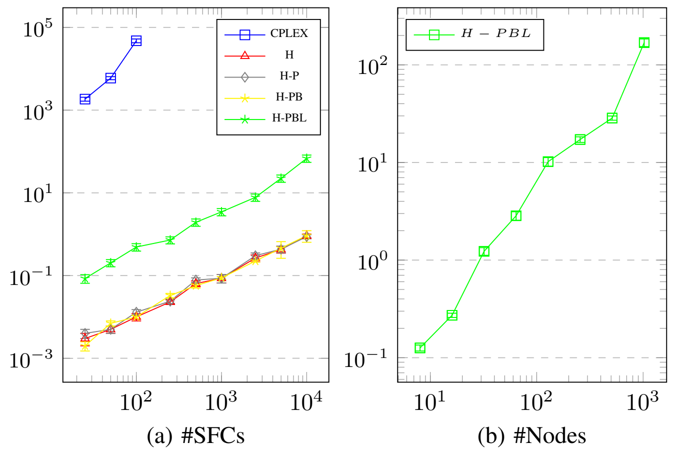

Fig. 3: Computation time (seconds) as a function of the number of (a) SFCs and (b) nodes

to the SFC number, ranging from 25 to 10,000, considering different versions of our heuristic as well as the exact method $CPLEX$. For the latter, the results are limited in range due to the exponential increase in computation time. The evolution of the computation time follows a logarithmic shape as the size of the instances increases. It is worth noting that the heuristic approach $H-PLB$ is on average 20,000 times faster than the exact method $CPLEX$. We can also observe that two distinct categories of computation time also emerge: the H, $H-P$ and $H-PB$ approaches, which present a computation time, on average, 40 times faster than the $H-PBL$ approach. This difference is due to the integration of online learning in $H-PBL$, which requires multiple deployment iterations to optimize the deployment order. As for the impact of parallelization in the $H-P$ version and load balancing in the $H-PB$ version, they seem to have a minimal, if any, impact on the computation time as compared to the H and $H-PL$ versions, respectively.

5) Computation Time as a Function of Node Number: Figure 3.b presents the evolution of the computation time as a function of the infrastructure size, defined by the number of nodes, from 8 to 1024. For this representation, the number of SFCs is kept constant, set at 30. Additionally, the available space is also constant, set to +150% of the number of micro-services. We observe here a clear increase in computation time, which is due to the modified $Eppstein$ algorithm used for computing the k shortest paths, whose complexity depends on the number of nodes in the infrastructure. It is worth noting that although this increase is linked to the size of the infrastructure, our approach outperforms that of the exact method of three orders of magnitude, being about 20,000 times faster for the scenario involving 128 nodes. Even when we are on very large infrastructure instances (512 or 1024 nodes) the execution time, roughly a few tens of seconds, proves to be acceptable for an operational deployment and highly scalable.

6) Load Balancing Quality: Figure 2 illustrates the Jain's fairness index for the two approaches $H-P$ and $H-PBL$ according to the Extra memory, Node number and SFC number. A clear distinction in terms of load balancing is observed in favor of the $H-PBL$ approach for all scenarios. Nevertheless, we note a convergence of the index in cases where the scenarios are restricted: (i) when the extra memory is low, and (ii) when the number of SFCs increases. Indeed, in this scenario where the infrastructure is highly constrained, load balancing becomes a challenging task, especially considering the stringent latency requirements for all SFCs. This limits our algorithm ability to balance, thus prioritizing latency compliance. Finally, as the number of nodes rises, Jain's indices of both approaches converge. Indeed, an infrastructure, composed of significantly large node numbers and a static capacity, can still achieve more balanced deployment without specific techniques.

V. Conclusion and Future Work

The relevance of the micro-services approach in LL applications has been acknowledged in literature, emphasizing its benefits such as deployment flexibility, scalability and reduced latency. However, traditional monolithic placement and chaining algorithms no longer fit with these features and novel dedicated solutions have to be proposed. As such, in this paper, we presented a heuristic method based on four key processes: the optimal computation of the k shortest paths, parallelization during deployment, load balancing, and the optimization of the SFCs processing order. The combination of these optimizations enable our solution to reach, on average, 1.1 times the optimum. Our evaluation demonstrated a significant improvement in computation time, being up to 20,000 times faster than the exact approach. It also showed a linear correlation between computation time and the number of nodes and SFCs, underscoring our approach effectiveness on larger instances. Regarding load balancing, when the network is not over-stressed, it improves performance by at least 2 points, prevents congestion and makes it easier to scale up. Overall, being lightweight and deterministic, our method is highly suitable for a real deployment scenario. Our future research investigates metaheuristic methods to move closer to optimality by leveraging the various solutions generated by different heuristic versions and cross-referencing them to produce more efficient offspring solutions.

Acknowledgment

This work is partially funded by the French ANR MOSAICO project, No ANR-19-CE25-0012 and the European Regional Development Fund FEDER, No OPE-2020-0031.

References

[1] S. Nádas, G. Gombos, F. Fejes, and S. Laki, "A congestion control independent l4s scheduler," in *Proceedings of the Applied Networking Research Workshop*, 2020, pp. 45–51.

[2] B. Briscoe, K. D. Schepper, O. Albisser, O. Tilmans, N. Kuhn, and G. Fairhurst, "Implementing the ' prague requirements ' for low latency low loss scalable throughput (l 4 s)," 2018.

[3] J. Sun, Y. Zhang, F. Liu, H. Wang, X. Xu, and Y. Li, "A survey on the placement of virtual network functions," *JNCA*, vol. 202, p. 1033, 2022.

[4] H. Vural, M. Koyuncu, and S. Guney, "A systematic literature review on microservices," in *ICCSA 2017*, 2017, pp. 203–217.

[5] H. Magnouche, G. Doyen, and C. Prodhon, "Leveraging microservices for ultra-low latency: An optimization model for service function chains placement," in *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, 2022, pp. 198–206.

[6] Z.-Q. Luo and J.-S. Pang, "Analysis of iterative waterfilling algorithm for multiuser power control in digital subscriber lines," *EURASIP JASP*, vol. 2006, pp. 1–10, 2006.

[7] S. R. Chowdhury, M. A. Salahuddin, N. Limam, and R. Boutaba, "Re-architecting nfv ecosystem with microservices: State of the art and research challenges," *Network*, vol. 33, no. 3, pp. 168–176, 2019.

[8] G. Liu, Y. Ren, M. Yurchenko, K. K. Ramakrishnan, and T. Wood, "Microboxes: High performance nfv with customizable, asynchronous tcp stacks and dynamic subscriptions," in *SIGCOMM*, 2018, p. 504–517.

[9] A. Bremler-Barr, Y. Harchol, and D. Hay, "Openbox: A software-defined framework for developing, deploying, and managing network functions," in *SIGCOMM*. ACM, 2016.

[10] L. S. Foundation. Data plane development kit. [Online]. Available: https://www.dpdk.org/

[11] D. Shadija, M. Rezai, and R. Hill, "Microservices: Granularity vs. performance," 12 2017, pp. 215–220.

[12] M. Nekovee, S. Sharma, N. Uniyal, A. Nag, and R. Nejabati, "Towards ai-enabled microservice architecture for network function virtualization," in *2020 ComNet*, 2020, pp. 1–8.

[13] Z. Meng, J. Bi, H. Wang, C. Sun, and H. Hu, "Micronf: An efficient framework for enabling modularized service chains in nfv," *JSAC*, vol. 37, no. 8, pp. 1851–1865, 2019.

[14] S. Chowdhury, A. Rahman, H. Bian, T. Bai, and R. Boutaba, "A disaggregated packet processing architecture for network function virtualization," *JSAC*, vol. 38, no. 6, 2020.

[15] H. Nabli, "An overview on the simplex algorithm," *Applied Mathematics and Computation*, vol. 210, pp. 479–489, 2009.

[16] G. Forney, "The viterbi algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.

[17] L. Askari, F. Hmaity, and M. Tornatore, "Virtual-network-function placement for dynamic service chaining in metro-area networks," 05 2018, pp. 136–141.

[18] A. Hirwe and K. Kataoka, "Lightchain: A lightweight optimisation of vnf placement for service chaining in nfv," in *IEEE NetSoft Conference and Workshops (NetSoft)*, 2016, pp. 33–37.

[19] A. Gadre, A. Anbiah, and K. Sivalingam, "Centralized approaches for virtual network function placement in sdn-enabled networks," *EURASIP JWCN*, vol. 2018, 08 2018.

[20] S. M. Kumari and N. Geethanjali, "A survey on shortest path routing algorithms for public transport travel," *Global Journal of Computer Science and Technology*, vol. 9, pp. 73–76, 2010.

[21] D. Eppstein, "Finding the k shortest paths," *SIAM Journal on computing*, vol. 28, no. 2, pp. 652–673, 1998.

[22] Y. Zhang, B. Anwer, V. Gopalakrishnan, B. Han, J. Reich, A. Shaikh, and Z.-L. Zhang, "Parabox: Exploiting parallelism for virtual network functions in service chaining," in *SOSR*. ACM, 2017, pp. 143–149.

[23] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "Nfp: Enabling network function parallelism in nfv," in *SIGCOMM*. ACM, 08 2017, pp. 43–56.

[24] S. Xie, J. Ma, and J. Zhao, "Flexchain: Bridging parallelism and placement for service function chains," *TNSM*, vol. 18, no. 1, pp. 195–208, 2021.

[25] I.-C. Lin, Y.-H. Yeh, and K. C.-J. Lin, "Toward optimal partial parallelization for service function chaining," *IEEE/ACM Transactions on Networking*, vol. 29, no. 5, pp. 2033–2044, 2021.

[26] P.-C. Lin, Y.-D. Lin, C.-Y. Wu, Y.-C. Lai, and Y.-C. Kao, "Balanced service chaining in software-defined networks with network function virtualization," *Computer*, vol. 49, no. 11, pp. 68–76, 2016.

[27] M.-T. Thai, Y.-D. Lin, P.-C. Lin, and Y.-C. Lai, "Towards load-balanced service chaining by hash-based traffic steering on softswitches," *JNCA*, vol. 109, pp. 1–10, 2018.

[28] F. Carpio, S. Dhahri, and A. Jukan, "Vnf placement with replication for loac balancing in nfv networks," in *IEEE international conference on communications (ICC)*. IEEE, 2017, pp. 1–6.

[29] C. You *et al.*, "Efficient load balancing for the vnf deployment with placement constraints," in *IEEE ICC 2019*, 2019, pp. 1–6.

[30] A. Zamani, B. Bakhshi, and S. Sharifian, "An efficient load balancing approach for service function chain mapping," *Computers & Electrical Engineering*, vol. 90, p. 106890, 2021.

[31] M. C. Luizelli, W. L. Da Costa Cordeiro, L. S. Buriol, and L. P. Gaspary, "A fix-and-optimize approach for efficient and large scale virtual network function placement and chaining," *Computer Communications*, vol. 102, no. C, pp. 67–77, 2017.

[32] M. Dianati, X. Shen, and S. Naik, "A new fairness index for radio resource allocation in wireless networks," in *IEEE Wireless Communications and Networking Conference, 2005*, vol. 2, 2005, pp. 712–717 Vol. 2.

[33] G. Bartoli, F. Chiti, R. Fantacci, and B. Picano, "An efficient resource allocation scheme for applications in lr-wpans based on a stable matching with externalities approach," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 6, pp. 58–59, 2019.

[34] F. Bari, S. R. Chowdhury, R. Ahmed, R. Boutaba, and O. C. M. B. Duarte, "Orchestrating virtualized network functions," *IEEE Transactions on Network and Service Management*, vol. 13, no. 4, pp. 725–739, 2016.

[35] M. M. Tajiki, S. Salsano, L. Chiaraviglio, M. Shojafar, and B. Akbari, "Joint energy efficient and qos-aware path allocation and vnf placement for service function chaining," *IEEE Transactions on Network and Service Management*, vol. 16, no. 1, pp. 374–388, 2019.